Information and Software Technology 196 (2026) 108140

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Advancing fuzzing with unbiased random generator and Feistel network-based mutations

Sadegh Bamohabbat Chafjiri [a,*], Phil Legg [b], Jun Hong [b], Michail-Antisthenis Tsompanas [b]

[a] Birmingham City University, Steamhouse Building, Birmingham, B4 7RQ, UK
[b] University of the West of England, Coldharbour Lane, Bristol, BS16 1QY, UK

ARTICLE INFO

Keywords:
Fuzzing
AFL++
Feistel-besed mutation
Unbiased random generator
Permuted Congruential Generator

ABSTRACT

Context: This research tackles challenges in traditional fuzzing, such as limited coverage, instability, and inefficiency in bug discovery. We propose two novel models and their combination to enhance mutation processes and improve its reliability through unbiased randomisation, building on cryptographic techniques from our prior work. To our knowledge, we are the first to apply this approach to AFL++, extending Feistel-inspired mutation and high-performance randomisation to generate high-quality test cases, with potential to attract attention in the fuzzing community.

Objectives:

- Integrate and assess Feistel-inspired mutations' impact on AFL++ performance, focusing on code coverage and stability.
- Integrate the *Permuted Congruential Generator* (PCG) into AFL++ and evaluate its performance compared to traditional random number generators (RNGs).
- Evaluate a hybrid model combining Feistel and PCG randomness for better stability and coverage.

Methods: We enhance AFL++ with algorithmic improvements and RNGs modifications. Our models include:

- CAFL++ (*Cryptographic*-AFL++): Integrates Feistel-inspired transformations for improved coverage.
- PCGAFL++: Refines the AFL's RNG with PCG to reduce bias.
- CPCGAFL++: Combines Feistel-inspired swaps and PCG-based RNG for a robust fuzzing approach.

Performance was analysed using metrics like Code Coverage and the Vargha-Delaney_A12 statistic across 20 Fuzzbench targets, and bug discovery on three targets.

Results: Our models showed significant improvements over AFL++. CAFL++ outperformed AFL++ in 75% of test targets, offering better code coverage and stability. PCGAFL++ surpassed AFL++ in 60% of targets by enhancing randomness, resulting in more efficient fuzzing. CPCGAFL++ demonstrated improved stability and enhanced bug discovery performance, while achieving code coverage comparable to AFL++. These results highlight the key improvements introduced by our two models for fuzz testing.

Conclusion: Our models advance fuzzing by improving code coverage and stability. Integrating Feistel-inspired swaps and PCG-based RNG overcomes traditional fuzzing limitations, offering a more efficient and reliable method. These models represent a step forward in fuzzing techniques, influencing both academic research and industrial practices.

1. Introduction

In recent years, fuzzing has emerged as a vital automated methodology in software testing, playing a crucial role in enhancing the robustness and reliability of software systems. The primary objective of fuzzing is to uncover and address unforeseen test cases that can lead to significant security vulnerabilities. By supplying a diverse array of inputs, often random, unexpected, or malformed, fuzzing provides a thorough mechanism for test the system's responses, to assess the resilience of the software against anomalous or malformed inputs.

The origins of fuzzing date back to 1989, when Miller et al. introduced a groundbreaking random testing tool designed to identify vulnerabilities in Unix utilities [1–3]. This foundational work established the principles of fuzzing and laid the groundwork for subsequent methodologies such as [4–6]. The 1990s saw further advances, with the development of increasingly sophisticated tools that improved

* Corresponding author.
E-mail address: sadegh.chafjiri@bcu.ac.uk (S. Bamohabbat Chafjiri).

https://doi.org/10.1016/j.infsof.2026.108140
Received 22 January 2025; Received in revised form 16 February 2026; Accepted 31 March 2026

vulnerability detection within Unix systems. By the late 1990s and early 2000s, significant innovations led to the emergence of numerous fuzzing methodologies, particularly in network protocol testing and web application security assessment [7,8].

Despite these advancements, early fuzzing models were often limited in precision, particularly when tasked with complex code structures found in contemporary software. For instance, blind fuzzing techniques [9–12], characterised by indiscriminate random input generation, often lack the depth necessary to effectively explore input formats. While they may uncover vulnerabilities in systems sensitive to unexpected data, their approach is generally superficial. In contrast, guided fuzzing [13–17] represents a significant advancement in the field, utilising insights into structured input formats and execution patterns to enable a more comprehensive exploration of code paths. Techniques such as grammar-based input generation [18], symbolic execution [19], and evolutionary algorithms have markedly enhanced code coverage and improved vulnerability detection rates. These improvements are achieved through sophisticated methods like taint analysis and execution path optimisation [20–23].

Over the past decade, significant advancements in fuzzing methodologies have gained considerable attention from both academia and industry [24–27]. Structured fuzzing, which generates inputs that adhere to specific formats, combined with feedback-driven strategies, has represented a leap forward in fuzz testing. These advanced methodologies have enabled a more comprehensive and targeted exploration of software vulnerabilities, addressing the complexities inherent in modern software systems.

Within this evolving landscape, mutation-based fuzzing [28–30] emerges as a pivotal technique, distinguished by its systematic approach to navigating complex input spaces. By employing a combination of bit, byte, and block-level operations, alongside well-defined seed and byte scheduling strategies, mutation-based fuzzing significantly enhances both the robustness and efficiency of fuzz testing. As software systems continue to increase in complexity, mutation-based fuzzing is positioned to play a crucial role in advancing software security testing. This is achieved through the integration of systematic input exploration [31], prioritisation techniques [32], adaptive mutation strategies [33], and feedback-driven adjustments [34].

In this context, we delve into mutation-based fuzzing methodologies, with a particular emphasis on recent advancements in cryptographic-inspired mutation [35]. Notably, our investigation also explores the impact of unbiased random generation techniques, a first in this framework, to the best of our knowledge. These innovations promise to significantly enhance fuzzing effectiveness by improving input diversity and unpredictability, ultimately broadening the scope of vulnerability discovery in complex software applications. The specific contributions of our research are outlined as follows:

- **Feistel-inspired Mutation in Fuzzing:** We explore the application of cryptogra-phic-inspired mutations, specifically the Feistel Network (FN) model within `AFL++` (`CAFL++`), demonstrating its adaptation as a swap function for generating high-entropy, diverse inputs. `AFL++` is used as a baseline and stepping stone for developing new methods due to its well-established framework, extensive community support, and modularity, making it an ideal platform for integrating and testing innovative approaches like FN-based mutations.
- **Development of an Unbiased Random Generator:** This paper introduces a framework based on the Permuted Congruential Generator (`PCG`) [36], providing an unbiased alternative to traditional `RNGs` in fuzzing. This framework (`PCGAFL++`) minimises bias in random input generation, improving code coverage and increasing the likelihood of identifying rare vulnerabilities.
- **Hybrid Approach:** We propose a hybrid model (`CPGAFL++`) that combines Feistel-inspired mutations from `CAFL++` with Permuted Congruential Generator (`PCG`) from `PCG-AFL++`, assessing whether this strategy yields more thorough software testing compared to traditional methods.

Through these contributions, we aim to establish a more robust and adaptable framework for mutation-based fuzzing, ultimately enhancing the resilience and security of modern software systems.

2. Motivation and research questions

Fuzzing effectiveness depends heavily on the diversity of generated test inputs and the quality of randomness guiding their selection. However, many widely used mutation engines, such as those in `AFL++`, are limited to shallow transformations (e.g., simple bit flips, arithmetic on fixed-width integers, or basic byte substitutions on swap functions) [20,37]. These operations often fail to produce sufficiently varied or structurally distinct inputs, resulting in poor exploration of deep or rare program states.

To address this gap, we ground our study in the concept of *diversifying* mutation within the swap function, which we define as the degree to which mutations introduce meaningful structural variation and expand the space of explored program execution paths. We adopt the Feistel-inspired transformation within a selective swap strategy. Its impact on mutation is driven by the parameterised flexibility of the underlying swap function, facilitating a broader and more diverse space of structural variations. We motivate this effect conceptually and evaluate it quantitatively through our proposed mutation metrics.

A second limitation in modern fuzzers is biased randomness. Many input-selection mechanisms inadvertently cluster test cases around common patterns, reducing the chance of exercising under-explored or rare program behaviours [36]. Using an unbiased pseudorandom generator can mitigate these effects and improve both state-space coverage and the discovery of nuanced vulnerabilities.

Motivated by these challenges, this work investigates whether Feistel-inspired swap transformations can imrove fuzzing coverage and whether integrating `AFL++`'s `RNG` with an unbiased `PCG`-based random generator yields a more effective fuzzing model that more reliably uncovers subtle software flaws. Furthermore, we evaluate the combined approach to determine whether it improves overall fuzzing performance or, conversely, whether the additional computational overhead may negate potential benefits and lead to degraded efficiency. These considerations lead us to the following research questions:

- How does integrating Feistel-inspired arithmetic operations impact the diversity and effectiveness of input mutations in fuzzing?
- What are the effects of using an unbiased `PCG`-based random generator on code coverage and the detection of rare vulnerabilities compared to traditional `RNGs` used in `AFL++`?
- Can a hybrid approach combining Feistel-inspired mutations and unbiased `PCG`-based random generation outperform the baseline `AFL++` in terms of mutation diversity and testing thoroughness?

2.1. Methodological motivation and background of mutation strategy

This subsection provides a rationale for the inclusion of the Feistel-inspired Swap function and PCG in our approach, highlighting the motivations for their selection and their anticipated contributions to the methodology.

2.1.1. Feistel-inspired swap

In this paper, the integration of Feistel-inspired swap into arithmetic operations builds directly on the solutions presented in our previous work on cryptography-inspired mutations [35]. The Feistel-inspired swap represents a well-known cryptographic structure called Feistel network [38] where an input block is divided into two halves. One half undergoes transformation through a round function before being merged with the other half using a reversible operation, typically XOR. Specifically, we continue and extend the Feistel-swap approach, previously demonstrated to outperform state-of-the-art fuzzers in targeted evaluations. Unlike prior work, which implemented this technique as a

custom mutation within Honggfuzz [39], our methodology integrates it natively into AFL++ in a more lightweight manner, allowing broader applicability. In this evaluation, we focus on the impact on code coverage, and we further evaluate this mutation strategy on a significantly expanded set of 20 FuzzBench benchmarks [40], thereby testing its generalisability and robustness.

The methodological motivation for choosing Feistel-inspired transformations lies in their structured yet flexible manipulation of input data. The Feistel structure, originally designed for symmetric-key cryptographic algorithms, enables controlled transformation of data through recursive bitwise swaps and permutations, and has shown improved results in [35] compared to the Substitution–Permutation Network (SPN) [41]. In addition, in the context of fuzzing, these characteristics are valuable and may help address some limitations: they enable mutations that are structurally consistent, often preserving partial input validity while introducing meaningful variance. For instance, Rawat et al. [42] highlight the limitations of random mutations in traversing deep program paths, motivating the search for more strategic input manipulations. By introducing targeted swaps within the input's memory layout, Feistel-inspired mutations can preserve execution-relevant substructures while still injecting entropy to explore new program states.

In addition to the example discussed above, other fuzzing techniques span a wide range of methodologies, which may warrant further review, including evolutionary-based approaches [43], greybox fuzzing [21], grammar-based methods [16], state-guided, and hybrid strategies [44], all of which may benefit from our mutation design. State-guided fuzzing, in particular, emphasises the importance of understanding and traversing internal program states to uncover vulnerabilities. Our adoption of Feistel-inspired swaps aligns with emerging directions across diverse fuzzing methodologies. These mutations can introduce structured variations that may help explore program state transitions more effectively, which are often crucial for uncovering hard-to-detect vulnerabilities.

Fuzz testing typically involves several distinct stages: importing inputs via the instrumentation driver, applying mutations to generate diverse samples, executing the fork/exec loop, monitoring process termination, and collecting execution feedback. This feedback loop, where coverage data such as jump destinations and path maps are recorded, is foundational to the effectiveness of greybox fuzzing. Our mutation strategy integrates seamlessly into this loop, leveraging coverage feedback to guide the application of Feistel-inspired swaps that are more likely to perturb relevant execution paths.

Recent advances in coverage-guided fuzzing highlight the effectiveness of mutation strategies that account for deeper software state models. These findings suggest that augmenting traditional mutators with more adaptive swap operations, such as the Feistel-inspired swap, can substantially enhance fuzzing depth and precision. Our initial results [35] demonstrated that such swaps enhanced the performance of the Honggfuzz mutator, particularly for programs with complex internal state transitions. However, our preliminary evaluation in that paper was limited to a narrow set of targets and did not fully utilise a real-world benchmarking suite with a broader range of targets.

To address these limitations, we expand the evaluation within the FuzzBench framework [40], a large-scale, reproducible benchmarking platform grounded in real-world fuzzing campaigns and widely used in academic and industrial research. This broader experimental scope allows us to assess the impact of Feistel-inspired mutations across diverse target programs. We aim to generate new empirical insights into their effect on both code coverage and the discovery of unique bugs. Additionally, by integrating the mutation strategy into AFL++, a widely adopted, extensible fuzzing platform [43], we ensure that the methodology is practical, reproducible, and applicable to real-world fuzzing campaigns.

Overall, the methodological choices presented here reflect a shift towards more structured fuzzing strategies. By combining the deterministic advantages of Feistel-inspired swaps with the adaptive mechanisms of AFL++, our approach contributes to the development of more intelligent and efficient fuzzers.

2.1.2. PCG

The PCG family [36] is a class of pseudorandom number generators designed to provide statistically strong output while maintaining a small state size and high performance. PCGs combine a simple linear congruential step for state evolution with a non-linear permutation function applied to the state before output. This separation of state transition and output transformation yields several advantages, most notably improved statistical quality and efficient implementation.

Despite the advantages of PCG generators, many widely used fuzzers — including AFL++ — still rely on simple linear congruential generators (LCGs) for randomness [45]. This is problematic because fuzzing is inherently stochastic, and its effectiveness depends critically on the quality of random decisions made during input generation and mutation. AFL++ employs the traditional C library rand()/srand(), an LCG family known to exhibit structural weaknesses formalised by Marsaglia's theorem [46]. Such weaknesses can introduce biased or low-entropy randomness, potentially reducing mutation diversity and limiting the fuzzer's exploratory power. Prior work has shown that the randomness characteristics of fuzzing inputs can substantially influence fuzzing effectiveness. However, the specific impact of embedded RNG within the fuzzer itself — and how such internal randomness shapes fuzzing behaviour — has not been systematically investigated. For instance, Klees et al. [47] demonstrate that fuzzing results can vary widely across runs due solely to random seed selection, underscoring the need to better control stochastic effects. Similarly, Schloegel et al. [48] observe that fuzzers are highly sensitive to probabilistic input-generation strategies. More recently, Feng et al. [49] argue that uncontrolled randomness can hinder directed fuzzers by slowing bug discovery. Additionally, while biased randomness has caused inefficiencies or vulnerabilities in other computational domains, such as Monte Carlo simulations, machine learning, and cryptography [50–55], its direct impact on fuzzing has not been empirically studied. In fuzzing, randomness governs critical operations such as mutation selection, input generation, and scheduling of test cases; bias in the underlying pseudorandom generator could skew these operations, potentially reducing input diversity or limiting path exploration. Together, this evidence suggests the need to systematically investigate bias in random number generation and assess whether it may represent an overlooked factor influencing fuzzing performance, providing further motivation for our proposed PCG design.

In addition, we selected PCG because it outperforms conventional pseudorandom number generators, such as LCG by providing statistically superior uniformity, improved equidistribution, and higher-quality randomness across bit positions, all while maintaining excellent computational efficiency [36]. O'Neill's comprehensive evaluation highlights PCG's ability to eliminate structural artifacts, such as low-order bit patterns, that commonly affect older generators. Moreover, PCG consistently achieves stronger statistical performance on empirical randomness test suites like TestU01. These advantages make PCG particularly well-suited for fuzzing applications, where biased or correlated random inputs can significantly impair code coverage and exploration diversity. By reducing such biases, PCG facilitates more reliable and effective path exploration, ultimately enhancing the robustness of fuzzing strategies.

In the following sections, this study first provides a comprehensive overview of existing literature on different categories of software fuzzing, culminating in a discussion of state-guided fuzzing and mutation strategies for memory swap, particularly with a focus on Feistel-based transformations, as well as randomisation techniques such as finite state machines (FSMs) in Section 3. Section 4 delves into the specific methodologies employed in our research. The experimental setup used to evaluate the performance of these enhanced fuzzing strategies is detailed in Section 5, which includes descriptions of the test environments and metrics. Section 6 presents the results of our experiments, and Section 7 analyses the rationale behind the effectiveness of the proposed methods in terms of coverage and efficiency. Finally, Sections 8 and 9 propose future work in this domain, discuss the potential impact of advancements on the future development and summarise our findings.

3. Related work

Fuzz testing typically involves several stages: importing the fuzzing input into the instrumentation driver, applying mutations to generate diverse inputs, executing the fork/execute mechanism, and awaiting process termination. During termination, the instrumentation driver records data such as jump destinations, execution path maps, and outcomes from the fuzzing target. Finally, the instrumentation driver validates and stores test cases while feeding back insights to refine subsequent input generation. These stages provide a comprehensive framework that supports the integration of various advanced strategies to enhance the effectiveness of the process, as outlined in below subsections.

This literature review critically analyses the progression of fuzzing techniques, emphasising the methodological advancements that have informed the development of the proposed Feistel-inspired mutation strategies. The principal methodologies discussed include greybox, coverage-directed, taint analysis, grammar-based, generative, hybrid, and evolutionary fuzzing [42,43], all of which contribute to improved vulnerability detection. Additionally, state-guided fuzzing and novel methods underscore the importance of understanding software states for effective vulnerability discovery. The review advocates for the adoption of innovative mutation strategies based on Feistel-inspired swaps, aimed at further enhancing code coverage in state-based fuzzing.

3.1. Greybox fuzzing

Greybox fuzzing acts as a pivotal intermediary between whitebox and blackbox testing methodologies, leveraging program structural knowledge while maintaining a degree of abstraction. Recent advancements in greybox fuzzing reflect its increasing sophistication. For instance, Ankou enhances greybox fuzzing by refining test case evaluation through improved fitness functions, which employ diverse combinations of execution data to address limitations observed in earlier fuzzers [56]. Coverage-based Greybox Fuzzing (CGF) enhances testing efficiency by specifically targeting low-frequency paths and optimising seed energy based on path exploration probabilities [21]. Directed Greybox Fuzzing (DGF) adopts a focused approach by concentrating on predefined targets during the fuzzing process and utilising simulated annealing for input generation [5]. These advancements illustrate the evolution of greybox fuzzing, highlighting the importance of tailored strategies to enhance fuzzing efficacy.

3.2. Coverage-directed fuzzing

Coverage-directed fuzzing has emerged as a critical strategy for improving vulnerability discovery. Techniques such as Classfuzz and CSI-Fuzz exemplify this approach, optimising fuzzing processes through advanced instrumentation and path tracking [57,58]. The integration of methodologies like the MO_{PT} mutation scheduling scheme enhances fuzzing capabilities by employing Particle Swarm Optimisation (PSO) to select optimal mutation operators, thereby increasing detection rates [29,59]. Recent advancements demonstrate a data-driven fuzzing strategy that incorporates preprocessing analysis and coverage-driven seed selection, framing seed scheduling as an integer linear programming (ILP) problem to provide objective measures of effectiveness [60].

3.3. Taint analysis

Taint analysis has become an integral component of fuzzing, facilitating the tracking of sensitive data flow to identify vulnerabilities. Innovations such as Angora leverage scalable byte-level taint tracking in conjunction with context-sensitive branch counting to enhance input generation [61]. Matryoshka effectively manages control and taint flow dependencies to explore critical application paths [62]. REDQUEEN offers a lightweight alternative to traditional taint tracking, utilising input-to-state correspondence during execution [63]. Furthermore, BuzzFuzz employs dynamic taint tracing to generate new fuzzed inputs while preserving the original structure, thereby deepening program code exploration [64].

3.4. Grammar-based and generative fuzzing techniques

Grammar-based and generative fuzzing techniques significantly enhance the reliability of fuzz testing. DEEPFUZZ integrates grammar-based fuzzing with generative modelling, successfully detecting bugs that traditional methods may overlook [65]. Skyfire utilises a data-driven approach to learn syntax and semantic rules for generating inputs likely to expose complex bugs [66]. Additionally, Constraint Logic Programming (CLP) generates well-typed programs for statically typed languages, effectively testing type checkers [67]. These innovations highlight the potential of these techniques to improve fuzz testing efficacy, particularly in identifying vulnerabilities within complex software systems [68].

3.5. Hybrid fuzzing techniques

Hybrid fuzzing, which includes concolic execution, integrates diverse methodologies to enhance the efficiency and depth of vulnerability analysis. For instance, Driller combines mutation-based fuzzing with selective concolic execution, optimising resource allocation to improve code coverage and detect embedded vulnerabilities [19]. FLAX, a hybrid fuzzer designed for JavaScript, integrates dynamic trace generation with taint analysis to target client-side validation vulnerabilities [69]. QSYM merges symbolic emulation with native execution through dynamic binary translation, improving emulation speed and resource conservation [70], and employs a hierarchical scheduler to optimise fuzzing prioritisation and scheduling through fine-grained coverage metrics and dynamic scheduling, thereby enhancing vulnerability discovery by ensuring comprehensive software exploration. Recent models, such as EcoFuzz, utilise an adversarial Multi-Armed Bandit model for adaptive resource allocation, facilitating enhanced path exploration in fuzzing [71]. Another notable approach is HD-Fuzz [72], which improves firmware fuzzing by incorporating hardware dependency awareness through hybrid modelling.

3.6. Evolutionary fuzzers

Evolutionary fuzzers enhance exploration in complex software applications without necessitating prior knowledge of input formats. VUzzer utilises control- and data-flow analysis to autonomously discern application properties, thereby improving adaptability [42]. The AFL family, including AFLNET, extends adaptability to protocol fuzzing through state-feedback-driven guidance [73]. CollAFL minimises path collisions to enhance tracking accuracy while preserving unique paths [22]. FairFuzz specifically targets rare branches using a dynamic mutation mask to focus on seldom-executed paths, thereby improving coverage [37].

3.7. State-guided fuzzing and mutation strategies for memory swap

State-guided fuzzing employs advanced techniques to enhance vulnerability analysis, which is critical for protocols that rely on cryptographic algorithms such as DTLS and TLS [74,75]. Tools like PULSAR and SNOOZE incorporate automated state inference, making them invaluable for closed-source environments that depend on cryptographic schemes for security [76,77]. Techniques developed by Pacheco and Steelix enhance scalability by integrating random testing with binary instrumentation [78,79]. These enhancements enable fuzzing to handle more complex codebases while maintaining precision in vulnerability discovery. A notable extension of these efforts includes techniques that

employ mini-simulations and attribute grammars, modelling input and behaviour through finite state machines (FSMs), which, as a component of cryptographic systems, particularly in stream ciphers like SNOW [80], function as sequential state machines, to systematically scramble initial seeds and generate a vast array of test cases [81]. Recent studies have explored the integration of state-based and Markov chain analysis for enhanced input generation, incorporating Monte Carlo techniques [82]. Methods such as FSMs and Monte Carlo are becoming increasingly prevalent in cryptography-focused methodologies and, more recently, in fuzzing research. FSMs and other theories, such as Monte Carlo methods, frequently provide structured yet adaptable models for simulating diverse operational scenarios, making them highly relevant for robust fuzzing analysis, especially in cases where state analysis is critical. In particular, state-guided fuzzing combined with memory swap-focused mutation strategies can enhance the precision and depth of testing, especially in complex memory management scenarios. As discussed in recent literature [35], these approaches examine how the Feistel-inspired swap can optimise input variation and enhance overall performance of HonggFuzz mutator [39] in AFL++ [43] in bug detection, but this study was limited to specific targets.

This paper aims to build upon Feistel-inspired mutation strategies directly within the AFL++ swap function, rather than using a custom mutator, by incorporating Feistel network structures. By implementing these strategies and expanding the scale of the experiment within the Fuzzbench [40] benchmarking framework, we seek to assess their effectiveness across diverse targets at a significantly larger scale than what was done in [35]. Our goal is to provide additional insights into how these fuzzing techniques can enhance both robustness and efficiency in AFL++, with a particular focus on code coverage. Furthermore, we investigate the number of unique bugs in certain cases within Fuzzbench. These advancements underscore a trend towards increasingly intelligent and adaptable fuzzing methods, marking a promising direction for improved vulnerability detection and software resilience.

4. Methodology

In this section, we want to delve into the methodology to explore the development and evaluation process of our solution. the fuzzing methodology is centred around enhancing the AFL++ fuzzing framework by implementing specific improvements to its core components, aimed at increasing the effectiveness of bug detection and coverage generation. These modifications involve both algorithmic improvements and changes to the underlying random number generation mechanism.

4.1. Key components of the methodology

In this methodology section, we outline the key enhancements introduced to the AFL++ fuzzing framework to improve its ability to detect vulnerabilities, particularly those related to arithmetic errors and randomness. The methodology consists of three main components: the integration of Feistel-inspired swaps to detect arithmetic relationships, the implementation of an improved RNG using the PCG, and the overall structural changes to the AFL++ codebase to support these modifications. Each of these improvements is designed to make the fuzzing process more effective by increasing the likelihood of uncovering subtle bugs that might be missed by traditional fuzzing techniques. In the following subsections, we delve into the specifics of these key components and their implementation within the fuzzing framework.

4.1.1. Arithmetic relationship detection with Feistel-inspired swaps

In the initial stage, new swap functions are integrated in Cryptographic-inspired AFL++, named CAFL++ and invoked during the fuzzing process to evaluate whether one numerical value can be interpreted as an arithmetic modification of another, based on specified byte lengths. The core analysis compares a novel implementation based on Feistel transformations with a baseline model using simple swapping techniques. This comparison aims to assess the effectiveness of Feistel transformations in enhancing the detection of arithmetic relationships within AFL++'s fuzzing framework. The key enhancement to the fuzzing process focuses on refining the `could_be_arith` function in `afl-fuzz-one.c`, leveraging swap transformations from the baseline model to optimise mutation operations. Traditional fuzzing methods only use basic swapping techniques to explore input mutations, but we introduce Feistel-inspired swap transformations. Based on an if-condition, which will be explained later, the function either invokes the baseline swap or the newly integrated swap, depending on the condition. These new transformations are specifically designed to detect and exploit arithmetic relationships using a Feistel-inspired swap pattern (rather than a simple swap) within the program being tested. By incorporating this technique, the fuzzing process becomes more capable of triggering bugs related to arithmetic errors, which are often difficult to uncover using conventional methods.

4.1.2. Improved random number generation (RNG) with PCG

In the second stage, we examine the PCG used in PCGAFL++ and compare it to the RNG in AFL++, which is based on the traditional `srandom` function. Both methods are reseeded periodically, but the structure and statistical properties of the RNGs differ, which can influence fuzzing efficiency and effectiveness.

4.1.3. Implementation pathway

The fuzzing methodology is implemented across several key files within the AFL++ codebase, including `afl-fuzz.h`, `afl-fuzz-one.c`, and `types.h`. Each of these files plays a crucial role in supporting the overall fuzzing pipeline, from RNG integration to mutation application. Additionally, we implemented a hybrid version of the Feistel-inspired swap and PCG models to compare its consistency with the baseline model. By mapping the changes across these components, the methodology ensures a cohesive approach to fuzzing that balances performance improvements with bug detection capabilities.

Fig. 1 presents a visual summary of the modifications introduced to the AFL++ fuzzing framework. It highlights enhancements to the `could_be_arith` function in `afl-f- uzz-one.c`, introducing Feistel-inspired swap transformations to detect arithmetic relationships, in contrast to the baseline swapping techniques. It also illustrates the integration of the PCG into `afl-fuzz.h`, replacing the traditional `srandom` RNG with a xorshift-enabled model, and provides a comparison of their structural and statistical properties. Additionally, it outlines the implementation pathway across key files, including `afl-fuzz.h`, `afl-fuzz-one.c`, and `types.h`. In place of the standard `srandom` RNG, the framework integrates the PCG into `afl-fuzz.h` [83,84]. The PCG is a modern, high-quality RNG that offers improved statistical properties and greater randomness compared to the traditional xorshift model. This substitution enhances the randomness in the generation of test inputs, providing a broader and more diverse set of mutations during fuzzing.

In the following subsections, we provide a detailed explanation of the development and evaluation of each code snippet integrated into AFL++. Section 4.2 elaborates on the enhancements made to the `could_be_arith` function in `afl-fuzz-one.c` and the use of `types.h` to support Feistel-based swap operations; Section 4.3 discusses the implementation of the PCG in `afl-fuzz.h` and its impact on randomness. To ensure compatibility with AFL++'s original behaviour, the existing swap functions are preserved alongside the new Feistel-based macros, allowing for the dynamic selection of transformation methods during fuzzing.

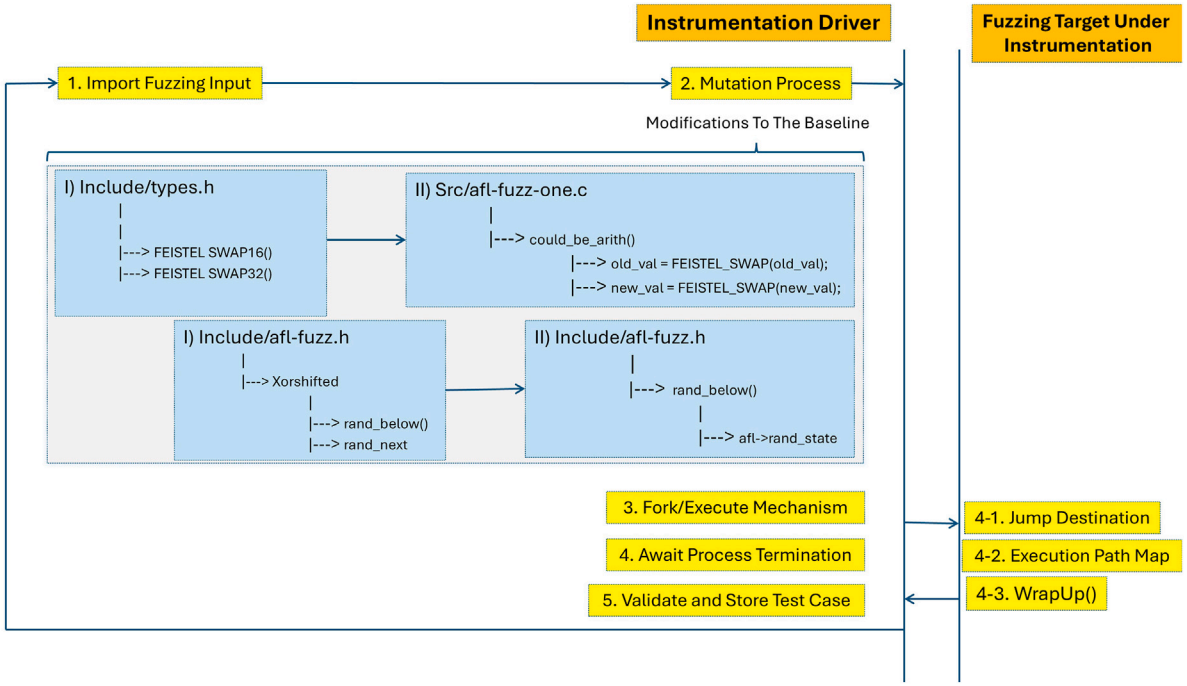


Fig. 1. Key components of the methodology.

Algorithm 1 Feistel-inspired Transformations for 16-bit and 32-bit Values

Input:
  x : integer (16-bit or 32-bit)
Output:
  y : transformed integer

  # Part (a): 16-bit Feistel Swap)
function FEISTEL_SWAP16(x)
  $left \leftarrow x \gg 8$
  $right \leftarrow x \ \& \ 0xFF$
  $newLeft \leftarrow right$
  $newRight \leftarrow left \oplus G(right)$
  return $(newLeft \ll 8)|newRight$

  # Part (b): 32-bit Feistel Swap)
function FEISTEL_SWAP32(x)
  $left1 \leftarrow x \gg 24$
  $left2 \leftarrow (x \gg 16) \ \& \ 0xFF$
  $right1 \leftarrow (x \gg 8) \ \& \ 0xFF$
  $right2 \leftarrow x \ \& \ 0xFF$
  $newLeft1 \leftarrow right2$
  $newLeft2 \leftarrow right1$
  $newRight1 \leftarrow left2 \oplus F(right2)$
  $newRight2 \leftarrow left1 \oplus F(right1)$
  return $(newLeft1 \ll 24)|(newLeft2 \ll 16)|(newRight1 \ll 8)|newRight2$

4.2. Feistel transformations

In AFL++, the rationale for using swap in `types.h` revolves around optimising memory management and improving efficiency, particularly in byte-order swaping, managing internal data structures like queues, fuzzing cases, or seeds. In the `could_be_arith` function, the `SWAP16` and `SWAP32` macros perform byte-order swaps (endianness swaps) rather than any container or memory management operations. This distinction is essential when comparing or analysing the effects of mutations involving multi-byte values (such as 16-bit or 32-bit data). The purpose of `SWAP16` and `SWAP32` operations is to determine whether the difference between $value_{old}$ and $value_{new}$ can be attributed to a byte-swap mutation instead of a simple arithmetic adjustment.

In the proposed Feistel transformation functions, an initial condition first checks for a byte-swap mutation. If this mutation does not sufficiently explain the data transformation, a second stage is introduced. This stage uses a simple swap function combined with a hybrid of byte-swap and Feistel transformations at the 16- and 32-bit levels.

Feistel-inspired transformations introduce additional operations, such as logical AND, thereby enabling a broader range of mutation patterns within swap operations and supporting more complex bit-level manipulations than a simple swap mechanism. This approach is to enhance code coverage and aids in uncovering complex data relationships, especially useful for fuzzing complex protocols.

4.2.1. Key enhancements of Feistel transformations

Algorithm 1 introduces two Feistel-based transformation functions, `FEISTEL_SWAP16` and `FEISTEL_SWAP32`, that extend the baseline model's swap operations in AFL++. Additionally, non-Linear Functions `G(x)` and `F(x)` introduced within swapping mechanisms, these transformations improve detection of complex patterns, while preserving baseline behaviours for simpler checks.

Algorithm 2 Non-Linear Bitwise Transformation Functions

Input:
  x : 64-bit integer
Output:
  y : transformed integer

  # Part (a) - $\gg k$ denotes logical right shift by k bits and & denotes the bitwise AND
function $G(x)$
  $y \leftarrow x \ \& \ (x \gg 4)$
  return y

  # Part (b)
function $F(x)$
  $y \leftarrow x \ \& \ (x \gg 8)$
  return y

FEISTEL_SWAP16: Operates on 16-bit data by dividing it into two 8-bit segments, swapping them, and applying a G(x) (see Algorithm 2, part (a)) transformation to the right segment, creating a non-linear bit pattern.

FEISTEL_SWAP32: Extends this to 32-bit data by applying F(x) (see Algorithm 2, part (b)) transformation on inner segments, introducing additional complexity suited for cryptographic and mutation contexts.

4.2.2. Feistel-enhanced arithmetic mutation detection

The updated could_be_arith function (Algorithm 3) extends previous arithmetic adjustment detection by applying selective Feistel-based permutations alongside conventional block-wise comparisons. This dual strategy efficiently determines whether $value_{new}$ can be interpreted as a permissible arithmetic mutation of $value_{old}$ under a bounded threshold k, for integer values of byte length $L \in \{2,4\}$. Key conventions used in the algorithm include the absolute arithmetic difference $\delta = |w_{\text{old}} - w_{\text{new}}|$ for 16 bits or $\delta = |v_{\text{old}} - v_{\text{new}}|$ for 32 bits.

The proposed procedure classifies the transformation as *arithmetic* if and only if there exists exactly one contiguous unit (byte, 16-bit word, or 32-bit value) whose absolute difference does not exceed k. The algorithm returns early if any block or word satisfies the arithmetic threshold k, improving efficiency.

The verification is performed hierarchically across granularities and repeated under byte-order reversal to ensure endianness robustness. Consequently, the method soundly captures bounded additive or subtractive mutations while excluding multi-byte or non-linear modifications.

4.2.3. Integration of Feistel transformations

The solution introduces a use_feistel flag that allows conditional application of Feistel transformations. The use_feistel parameter controls whether Feistel-based transformations (FEISTEL_SWAP16, FEISTEL_SWAP32) or standard endianness swap functions (SWAP16, SWAP32) are applied. This design enables the fuzzing engine to dynamically select mutation strategies based on the target, adding decision-making for complex patterns without altering the original function's purpose. This approach is particularly effective for multi-byte numeric inputs, cross-endian programs, and structured protocols where integer values can appear in different byte orders. This model balances simple delta checks with Feistel-adjusted comparisons, capturing subtle arithmetic mutations across both block and full-word scales.

4.2.4. Prototype model and real-world applications

Our model serves as a prototype for detecting arithmetic operations across a wide range of applications. Its primary focus is on scrambling data patterns using complex swapping techniques, which enhances its ability to detect and adapt to subtle, intricate changes in the data. This capability makes the model particularly effective in scenarios where sensitivity to fine-grained mutations is crucial. Our model serves as a prototype for detecting arithmetic operations across a wide range of applications. Its primary focus is on scrambling data patterns through complex swapping techniques, enhancing its ability to identify and adapt to subtle, intricate changes in the data. This heightened sensitivity is not limited to specific patterns but can also be applied to real-world challenges, such as scrambler attacks in eavesdropping. These attacks exploit "over-simplistic implementations" of system components, highlighting the model's potential to address vulnerabilities in practical scenarios [85,86]. A scrambler attack in this context exploits weaknesses in the randomisation process of communication systems by using a simplified pseudo-random number generation method. By overhearing a single communication, an attacker can exploit this predictability to breach privacy without detection. These situations can be analysed through our prototype model, where complex mutation strategies are crucial for effectively protecting privacy.

4.2.5. Impact on fuzzing strategies and benefits for applications

The integration of the Feistel-inspired swap methodology into fuzzing strategies introduces a significant advancement over traditional blind fuzzing techniques by enabling targeted modifications to scrambled data. Instead of randomising data entirely in one iteration, this approach applies the Feistel-inspired swap to alter half of the data at each step. This focused scrambling simulates realistic error scenarios, such as corrupted data packets or unexpected data types, which are common in eavesdropping attempts or scrambler attacks. By generating inputs that are mostly valid but subtly altered, this method enhances fuzzing effectiveness, uncovering vulnerabilities in areas such as error handling, data parsing, buffer boundaries, and data type validation, issues often missed by conventional fuzzing.

Algorithm 3 Feistel-Enhanced Arithmetic Mutation Detection (Incremental)

Input:
  v_{old}: original integer value
  v_{new}: mutated integer value
  L: byte-length of the value ($L \in \{2,4\}$)
  k: maximum allowed arithmetic delta
  $use\_feistel$: whether to apply Feistel permutation
Output: `true` if mutation can be explained under Feistel-adjusted arithmetic; otherwise `false`

/* 16-bit block Feistel adjustment */
for each 16-bit block position i **do**
  Extract word w_{old} and w_{new} at position i
  if $w_{old} \neq w_{new}$ **then**
    $\delta \leftarrow |w_{old} - w_{new}|$
    if $\delta \leq k$ **then**
      return `true`
    end if
    /* Feistel or simple swap */
    if $use\_feistel$ **then**
      $w_{old} \leftarrow FEISTEL\_SWAP16(w_{old})$
      $w_{new} \leftarrow FEISTEL\_SWAP16(w_{new})$
    else
      $w_{old} \leftarrow SWAP16(w_{old})$
      $w_{new} \leftarrow SWAP16(w_{new})$
    end if
    $\delta \leftarrow |w_{old} - w_{new}|$
    if $\delta \leq k$ **then**
      return `true`
    end if
  end if
end for

if $L = 4$ **then**
  /* 32-bit Feistel adjustment */
  $\delta \leftarrow |v_{old} - v_{new}|$
  if $\delta \leq k$ **then**
    return `true`
  end if
  if $use\_feistel$ **then**
    $v_{old} \leftarrow FEISTEL\_SWAP32(v_{old})$
    $v_{new} \leftarrow FEISTEL\_SWAP32(v_{new})$
  else
    $v_{old} \leftarrow SWAP32(v_{old})$
    $v_{new} \leftarrow SWAP32(v_{new})$
  end if
  $\delta \leftarrow |v_{old} - v_{new}|$
  if $\delta \leq k$ **then**
    return `true`
  end if
end if
return `false`

Moreover, this approach is particularly beneficial for fuzzing applications that demand heightened data complexity. By combining simple byte-swaps with other bit-wise or byte-wise transformations, the methodology introduces a layer of analytical depth. The use of Feistel transformations, such as FEISTEL_SWAP16 or FEISTEL_SWAP32, extends beyond basic swaps to perform non-trivial operations, enhancing sensitivity to complex mutations. This enables the detection of subtle transformation patterns that traditional fuzzing tools, like baseline AFL++, may fail to identify. Although this complexity may increase

the rate of false positives, it broadens the spectrum of mutation patterns and strengthens the detection of vulnerabilities tied to arithmetic and scrambling operations. To the best of our knowledge, such complex transformation capabilities are not present in the baseline AFL++, making this integration a notable innovation in fuzzing methodology.

4.2.6. Quantifying the impact of Feistel swaps on mutation complexity

AFL++ predominantly relies on deterministic swap mutations that exchange the left and right bytes of 16-bit words and the left and right halves of 32-bit words. These swap operations are strictly localised: they only reorder existing bytes within a fixed word boundary and do not introduce any new arithmetic relationships or cross-bit interactions. As a result, swap mutations modify only a small, structurally isolated portion of the input at a time, limiting their ability to explore paths that depend on coordinated multi-byte or non-linear arithmetic conditions. While effective for shallow input–state exploration, these limited-variation mutations often fail to produce the coordinated multi-bit changes required to reach program paths involving non-linear or cross-bit relationships, such as integer overflows, carry-propagation effects, or modulo-arithmetic edge cases. Mathematically, if x is a seed input and $S(x)$ is the AFL++ swap, the expected mutation complexity can be expressed as

$$C(S) = \mathbb{E}[d(x, S(x))] \tag{1}$$

where $d(\cdot,\cdot)$ measures structural or Hamming distance. Because simple swaps affect a limited number of bits, $C(S)$ is low, and many program paths, especially those requiring complex or non-linear input changes, may remain unexplored.

To address this, Feistel-inspired non-linear transformations $F(x)$ and $G(x)$, which combine swaps with XOR and bitwise operations (Algorithm 2) and are selectively guided by input differences and block size, can be formalised as:

$$M_{\text{selective}}(x,y) = \begin{cases} \text{identity or arithmetic adjustment}, & \text{if diffs} = 1 \text{ and } |x-y| \le k \\ \text{FEISTEL}(x), & \text{if use\_feistel} = \text{true and } |x-y| > k \\ \text{SWAP}(x), & \text{otherwise} \end{cases} \tag{2}$$

The method separates small arithmetic modifications from structural transformations, applying linear arithmetic, Feistel permutation, or swap operations as appropriate, thereby preserving AFL++ behaviour while increasing mutation complexity for deeper path exploration.

$$C(H) = p \cdot C(S) + (1-p) \cdot C(F) \tag{3}$$

By increasing both structural diversity and Hamming distance of mutated inputs, this strategy improves coverage and the likelihood of discovering vulnerabilities that simple deterministic swaps alone would miss.

4.3. Mitigating RNG bias in AFL++ through PCG integration

In this section, we first discuss the methodological challenge of module bias in fuzzing and then present our approach to address this issue, highlighting the advantages of combining `xorshift` with rotation.

4.3.1. Modulo bias problem

The original `rand_below()` function in AFL++ aims to generate random numbers uniformly across a given range. However, simply using the modulo operator (`random_number % limit`) introduces **bias** if the RNG's output is not evenly divisible by the limit. This results in some numbers being generated more frequently than others, which is undesirable in fuzzing since it can skew the input selection. For example, if the `limit` is 100 and the RNG produces values between 0 and 255, numbers between 0 and 55 would have slightly higher probabilities of being selected due to the nature of modulo operation. This leads to a biased random distribution.

Algorithm 4 Remove Modulo Bias

Input:
  L : upper bound of desired range
Output:
  y : unbiased random integer in $[0, L-1]$
$M \leftarrow 2^{64}$ # Total number of possible outputs
$T \leftarrow M - (M \bmod L)$ # Largest multiple of L below M to eliminate incomplete residue
repeat
  $x \leftarrow$ NextRandom() # 64-bit RNG output using PCG (output of PCG-RN (RXS-M-XS))
until $x < T$ # Reject values in upper tail to prevent modulo bias
$y \leftarrow x \bmod L$ # Safely map accepted value into $[0, L-1]$
return y # Return unbiased random integer

4.3.2. Unbiased RNG methodology

Our methodology introduces specific bitwise operations, namely `xorshift` [87] and `rotate`, designed to enhance randomness and uniformity, especially when generating random bits from the state. Subsequently, the `rand_next` function applies the PCG's specialised permuted output function, `RXS-M-XS`, which promotes uniform randomness even as the state advances. Our primary focus is on bias detection and uniformity in randomness, both of which are essential in fuzzing applications to ensure thorough exploration of the input space. AFL++ utilises a simpler RNG based on the `srandom` function. Once seeded, the RNG updates the state as follows:

```
rand_state = srandom(seed)
```
(4)

This RNG lacks the sophisticated transformation function present in the PCG algorithm, which may introduce biases in its output due to its relatively simple randomisation mechanism.

As a result, some code paths may be exercised more frequently while others remain underexplored, reducing overall coverage and potentially causing bugs to be missed. Our solution, PCGAFL++, however, implements the PCG-RNG algorithm, known for generating lower-bias random numbers during state updates. To achieve this, we designed the `rand_below()` in AFL++ with two-Stage process to remove bias.

Stage 1: Generate a random number

In the first stage, a raw random number is generated using `rand_next()`. However, directly applying the modulo operation at this point can still introduce bias. Therefore, we need to ensure that the generated number falls within a range that is uniformly divisible by the `limit`. In AFL++, the parameter `limit` defines the upper bound of the target interval for bounded random number generation. In our formalisation, this parameter is denoted as L.

Stage 2: Remove modulo bias

In the second stage presented in Algorithm 4, modulo bias is eliminated through rejection sampling. Random values are repeatedly generated until one falls within the acceptable interval.

Bias elimination for bounded generation

To generate a uniformly distributed random integer in the interval $[0, L-1]$, direct application of the modulo operation may introduce bias when 2^{64} is not divisible by L. To avoid this issue, rejection sampling is applied.

Let $M = 2^{64}$ and $T = M - (M \bmod L)$. Only values $y < T$ are accepted, and the final bounded random number is computed as

$$r = y \bmod L. \tag{5}$$

This ensures uniform distribution without bias, even when L does not divide 2^{64} evenly. In the modified version of `rand_next()`, the PCG is used as the RNG. The PCG implementation works in two main steps:

Step 1: State transition

The internal state of the generator is updated using a linear congruential transformation:

$$s \leftarrow s_{\text{old}} \cdot a + c \pmod{2^{64}}, \tag{6}$$

where a and c are fixed constants defining the generator, $\cdot$ denotes multiplication and + denotes addition. In this linear congruential step, the current state s_{old} is multiplied by a fixed constant a and incremented by a constant c. The result is reduced modulo 2^{64} to ensure that the state remains within the 64-bit space.

Step 2: Output permutation (RXS-M-XS)

After updating the state, a permutation function is applied to the previous state value in order to improve statistical quality. The transformation consists of a xorshift followed by a data-dependent rotation, as demonstrated by the Algorithm 5.

Algorithm 5 RXS-M-XS Transformation and PCG−RNG

Input:
  s : current internal state (64-bit integer)
  L : upper bound of desired range
Output:
  r : random integer in $[0, L-1]$
  s' : updated internal state

  # Part (a): Handle trivial range
if $L \leq 1$ **then**
  return $0, s$
end if

  # Part (b): Threshold for modulo bias)
$M \leftarrow 2^{64}$
$T \leftarrow M - (M \bmod L)$

  # Part (c): RXS-M-XS iteration — $\ll k$ denotes logical left shift by k bits and $\oplus$ denotes the bitwise XOR and $|$ denotes the bitwise OR
repeat
  {State transition (linear congruential step)}
  $s_{\text{old}} \leftarrow s$
  $s \leftarrow s_{\text{old}} \cdot a + c$
  {Output permutation (RXS-M-XS) }
  $x \leftarrow \left((s_{\text{old}} \gg 18) \oplus s_{\text{old}}\right) \gg 27$
  $\rho \leftarrow s_{\text{old}} \gg 59$
  $y \leftarrow (x \gg \rho) \mid (x \ll ((-\rho) \bmod 32))$
until $y < T$

  # Part (d): Final output
$r \leftarrow y \bmod L$
$s' \leftarrow s$
return r, s'

Therefore, the RXS-M-XS permutation introduces non-linearity and improves the distributional properties of successive outputs by reducing observable correlations.

The RNG periodically refreshes its internal state to maintain entropy and avoid predictability, ensuring that the produced random values remain statistically robust over time. In the baseline approach, the state is directly initialised with a fixed seed, which can help prevent cumulative bias if the RNG begins to exhibit predictable patterns. Our model, however, shows promise in reducing bias over time due to the properties of `xorshift` [87].

While both the baseline and PCG-based implementations apply comparable strategies to minimise modulo bias, PCGAFL++ leverages the PCG generator, which is renowned for its improved randomness properties and reduced bias relative to standard RNGs. As a result, PCGAFL++ is likely to offer a more unbiased and statistically uniform distribution in comparison to traditional LCGs or other basic RNGs, making it a preferable choice in scenarios where unbiased random numbers are paramount, particularly in fuzzing applications.

5. Experiment setup

This section details the experimental setup, including the testing environment, target selection, and performance metrics. A controlled framework evaluates our proposed fuzzesr against AFL++, ensuring consistent conditions with defined hardware and software. Diverse real-world targets assess adaptability, while core metrics, code coverage, bug discovery, and comparative analysis, highlight each fuzzer's strengths across scenarios.

5.1. Testing environment

We used the Fuzzbench dataset, a benchmarking tool popular in the fuzzing community, to evaluate state-of-the-art fuzzers on 20 real-world benchmarks and three target variants. Each configuration ran multiple trials (at least 5, as defined by the FuzzBench default configuration) over 24 h, measuring code coverage and unique bugs. Experiments were conducted on a Kali Linux VM with 16 GB RAM, hosted on VMware Workstation 17 running Windows 11 Education. The workstation provided 4 CPU cores and 16 GB RAM. The choice of using at least five trials reflects the computational constraints of the original experimental environment and aligns with the practical resource limitations commonly encountered by researchers without access to high-performance clusters. Consequently, at least five trials represented a feasible and realistic balance between methodological rigour and available hardware.

5.2. Selected targets

We chose diverse targets across domains such as general-purpose applications, network protocols, data processing, media handling, and system management to evaluate our fuzzer against the state-of-the-art AFL++. Table 1 categorises these targets and highlights their selection rationale, demonstrating the fuzzer's efficacy and versatility across varied applications. Each target was selected for its relevance and representation of real-world software in fuzz testing. They emphasise the proposed fuzzer's strengths by tackling category-specific challenges, including complex input formats, structured data management, and adherence to security protocols.

5.3. Fuzzer performance metrics

We evaluated fuzzer performance using key metrics to assess code exploration and bug discovery efficacy. Code Coverage and Sample Statistics provided a baseline for exploring target program code, while the Relative Code Coverage metric offered finer granularity. The *Vargha-Delaney_A12 statistic* measured effectiveness in uncovering unique program paths. By combining absolute and relative metrics with additional case-specific analyses, our study delivers a comprehensive evaluation of fuzzer performance.

5.3.1. Code coverage

The primary metric, *code coverage statistics*, measures the range of code branches reached, reflecting each fuzzer's thoroughness in exploring the codebase. *Relative code coverage* compares each fuzzer's median performance to the experiment's maximum observed coverage, which may not reach 100%, as it is benchmarked against the highest

achieved coverage rather than an absolute scale. Relative coverage for each trial is calculated as:

$$\text{trial\_relative\_coverage} = \frac{\text{trial\_coverage}}{\text{experiment\_max\_coverage}} \quad (7)$$

This approach provides an averaged comparative view of each fuzzer's effectiveness across multiple test cases, with the rankings ordered by a "FuzzerMean" metric to underscore relative performance.

5.3.2. Vargha-Delaney_A12 statistic

To further refine our analysis, we incorporate the *Vargha-Delaney_A12 statistic*, a probabilistic metric used to assess the likelihood that one fuzzer consistently outperforms another in terms of code coverage across multiple trials, within a 4×4 matrix. An *A12* score of 1 indicates that the fuzzer being evaluated (fuzzer 1) outperforms all others in every comparison. In this context, the 'row' in the matrix refers to the fuzzer being evaluated (fuzzer 1), and the 'column' refers to the other fuzzers being compared against (for example, fuzzer 2), as shown in the matrix of probabilities, where the rows and columns represent different fuzzers.

When the score satisfies $p > q$, where p and q represent probabilities derived from the comparison:

- p: The probability that the fuzzer in the row (fuzzer 1) achieves higher code coverage than the fuzzer in the column (fuzzer 2) across trials.
- q: The probability that the fuzzer in the column (fuzzer 2) achieves higher code coverage than the fuzzer in the row (fuzzer 1).

If $p > q$, it implies that fuzzer 1 is more likely to outperform fuzzer 2 consistently over multiple trials. Conversely, if $p = q$, the two fuzzers perform similarly, and if $p < q$, fuzzer 2 has a higher likelihood of outperforming fuzzer 1. This statistic provides a nuanced view of relative performance by considering the distribution of results rather than just aggregate metrics like averages. This metric is crucial for evaluating the comparative stability and performance reliability of each fuzzer across repeated runs.

5.3.3. Additional metrics for specific targets

For selected targets, such as the "harfbuzz_hb-shape-fuzzer_17863b", "bloaty_fuzz_tar- get_52948c" and "libxml2_xml_e85b9b" the Fuzzbench dataset provides additional metrics, which are not universally available across all targets. These expanded metrics include *unique bug count* and *bug coverage*, along with the *Vargha-Delaney A12 measure* applied to bug discovery rates. Together, these measures enrich the evaluation framework by allowing analysts to assess both the depth of code coverage and the fuzzer's bug-finding efficiency and success likelihood relative to other fuzzers. For these specialised targets, we use this expanded set of metrics to enable a comprehensive comparison of fuzzers' capabilities in bug detection and overall robustness.

6. Performance analysis

In this section, we conduct a comprehensive quantitative and qualitative analysis to gain a more accurate understanding of fuzzer performance. This is achieved through statistical analysis using metrics such as the mean, standard deviation, and the *Vargha-Delaney_A12 statistic*. These measures provide statistical and visual insights into the central tendency and variability of performance, along with the effect size in comparative evaluations. Additionally, relative performance is evaluated by examining relative ranks, enabling an objective comparison of the fuzzers.

6.1. Code coverage statistics

Table 2 summarises the performance rankings of the evaluated fuzzers across all targets. It highlights how frequently each fuzzer outperformed the baseline, AFL++, and provides a breakdown of the number of times each fuzzer secured specific ranks, from 1st to 4th place. The top-performing fuzzers in each position, based on the highest frequency they achieved in 1st, 2nd, and 3rd places, are highlighted in green, while the worst-performing fuzzer, based on the most frequent 4th-place, is highlighted in red.

CAFL++ consistently outperformed AFL++, securing the top spot in three-quarters of the total targets, indicating its significant improvements over AFL++ in most scenarios. Although CAFL++ shows some variability in performance compared to AFL++ on a few targets, this does not diminish its ability to consistently secure first-place rankings on most targets.

PCGAFL++ strikes a balance between performance and stability. Despite some fluctuations in its results, it outperformed AFL++ on three-fifths of the targets, securing second place. While it shows potential, further improvements are needed to match the performance of CAFL++.

AFL++, historically significant, consistently ranks lower than the other fuzzers. It was placed in 4th place in just under half of the targets due to its lower average performance and higher variability, though it still performs solidly in certain cases (6 targets).

In contrast, CPCGAFL++ offers more stability, securing 1st and 2nd ranks on a total of 11 targets, compared to AFL++'s 8 and PCGAFL++'s 6, with lower variability and greater consistency, ranking 4th only 6 times compared to AFL++'s 8. However, it tends to lag behind in peak performance relative to CAFL++, offering a balanced but less aggressive performance.

In summary, CAFL++ and PCGAFL++ outperformed AFL++ in more than half of the 20 experiments. CPCGAFL++'s consistent performance further supports its viability as an effective choice for fuzzing tasks.

These results underscore the superior and more consistent performance of the integrated solutions compared to AFL++ across a wide range of targets. For more details, please see Table A.5 in Appendix A and Fig. B.2 in Appendix B, which provide a more detailed breakdown of each fuzzer's performance through Coverage Statistics (mean, standard deviation, min, 25%, median, 75%, max) and its trend over 24 h. These statistics confirm the trends observed in the rankings.

6.2. Relative code coverage

Table 3 presents a comparative analysis of the relative performance of each fuzzer. In this context, a fuzzer that does not occupy the top rank (green entries) is highlighted in shades of blue if its performance exceeds 95%. The intensity of the blue colour increases as its performance approaches 99% and remains uncoloured if its performance falls below this threshold. The number of uncoloured entries serves as an important metric for identifying the least effective fuzzers in the competition across all target environments.

From this perspective, among the fuzzers exhibiting minimal uncoloured relative coverage, namely CAFL++, PCGAFL++, and CPCGAFL++, CPCGAFL++ stands out as the highest-ranking fuzzer. It consistently achieves the most top ranks (as indicated by green entries) and demonstrates the most stable performance in terms of code coverage. In the second position, CAFL++ maintains 13 top ranks, while also securing three second-place and a single third-place rank. PCGAFL++ ranks third, with 16 instances exceeding 95%, including 8 first-place, 5 second-place, and 3 third-place.

AFL++, on the other hand, emerges as the least effective fuzzer because, in the cases where it does not secure the top rank (indicated by green), its relative coverage falls below 95% on six targets, with only two targets exceeding 95%. This represents lower performance

Table 1
Selected fuzzer targets with categories and reasons.

Category	Target	Reason for selection
General-purpose	bloaty_fuzz_target	A good general-purpose target with a focus on binary size reduction.
	bloaty_fuzz_target_52948c	Specific variant for binary size reduction analysis, helping to study number of unique bugs.
Networking	curl_curl_fuzzer_http	Represents network protocol fuzzing, particularly HTTP, a critical area in security.
	openthread_ot-ip6-send-fuzzer	Related to IoT and network protocols, showing diversity in network fuzzing.
Graphics	freetype2_ftfuzzer	A widely used library for font rendering, significant in graphics and UI applications.
	harfbuzz_hb-shape-fuzzer	Targets text shaping in the HarfBuzz library, crucial for rendering complex text.
	harfbuzz_hb-shape-fuzzer_17863b	Another variant of the HarfBuzz fuzzer, designed to analyse the number of distinct bugs in the text shaping functionality.
Data formats	jsoncpp_jsoncpp_fuzzer	Targets JSON parsing, fundamental in modern web applications and APIs.
	libxml2_xml	Targets XML parsing, crucial for data interchange formats and configuration files.
	libxml2_xml_e85b9b	An additional fuzzer for XML parsing, designed for analysing the number of unique bugs in this common data format.
	libxslt_xpath	Focuses on XPath in XSLT processing, important for XML data transformations.
	woff2_convert_woff2ttf_fuzzer	Targets font file conversion, relevant in web and graphic design.
Media processing	lcms_cms_transform_fuzzer	Focuses on colour management for images, essential for accurate media representation.
	libjpeg-turbo_libjpeg_turbo_fuzzer	Focuses on JPEG encoding/decoding, relevant for media applications.
	libpng_libpng_read_fuzzer	Targets PNG image handling, crucial for media processing.
	openh264_decoder_fuzzer	Targets H.264 decoding, essential for video streaming and playback.
	stb_stbi_read_fuzzer	Targets image loading in the stb library, widely used in games and multimedia apps.
	vorbis_decode_fuzzer	Focuses on audio decoding, significant for media applications.
Database	sqlite3_ossfuzz	A popular embedded database system, representing database interactions.
Compression	zlib_zlib_uncompress_fuzzer	A well-known compression library, important for many applications dealing with data compression.
Scripting language	php_php-fuzz-parser_0dbedb	Tests PHP code parsing, crucial for web applications and server-side scripting.
Geospatial	proj4_proj_crs_to_crs_fuzzer	Focuses on coordinate system transformations, essential for geospatial applications.
Text processing	re2_fuzzer	Targets regular expression handling, vital for text parsing and processing.

Table 2
Fuzzer ranking performance, comparison against AFL++, and Trade-offs.

Fuzzer	Frequency of outperforming baseline	1st place	2nd place	3rd place	4th place	Trade-offs across fuzzers
CAFL++	15	10	5	3	2	Excels in peak performance but shows some variability.
PCGAFL++	12	2	4	10	4	Offers a middle ground but requires further enhancements for top-tier competitiveness.
AFL++	–	4	4	4	8	While occasionally effective, ranks the lowest overall due to its reduced average performance and higher variability.
CPCGAFL++	9	4	7	3	6	Prioritises stability in securing first and second ranks compared to the baseline, making it a dependable choice for varied targets.

compared to the other fuzzers. In contrast, the other fuzzers have only three instances of performance below 95% when not in the top position. Therefore, the number of uncoloured entries provides a clear indication of the relative effectiveness of these fuzzers.

6.3. Vargha-Delaney A12 statistic

This section assesses the *Vargha-Delaney A12 statistic*, providing a comparative analysis of the probability that a given fuzzer outperforms another. The summarised *A12* values from the pairwise Vargha-Delaney measure of effect size are displayed in the tables across all targets presented in Fig. C.3 in Appendix C. Cells shaded green indicate the likelihood of the fuzzer in the corresponding row surpassing the fuzzer in the column.

The frequency analysis reveals that the following fuzzers achieve a success rate of higher than 50% compared to the baseline AFL++ across all targets: CAFL++ in 14 instances, CPCGAFL++ in 13 instances, and PCGAFL++ in 12 instances. For CPCGAFL++, specific cases, such as targets, libjpeg-turbo_ibjpeg_turbo_fuzzer, met the 50% threshold under certain conditions; however, these were not included in the overall statistics.

Breaking this down further, the number of targets each fuzzer outperformed others (calculated by counting rows with green entries per fuzzer) is as follows: CAFL++ leads with eight wins, followed by CPCGAFL++ with five wins, AFL++ with four wins, and PCGAFL++ with three wins.

When evaluating the combined performance, all three fuzzers collectively outperform the baseline AFL++ in nine cases (identified by either the first columns with three green entries or the first rows with three red entries in Fig. C.3). Conversely, in five instances, these three alternative fuzzers collectively underperform against the baseline AFL++. This evaluation confirms that CAFL++, in particular, stands out as the most consistently successful fuzzer, with CPCGAFL++ and PCGAFL++ also demonstrating strong consistency in their likelihood of surpassing the baseline fuzzer on more than half of the targets.

6.4. Total number of unique bugs

A few target variants, such as harfbuzz_hb-shape-fuzzer_17863b, bloaty_fuzz_tar- get_52948c, and libxml2_xml_e85b9b, yielded valuable insights, including the discovery of unique bugs. However, these features were not available for all Fuzzbench benchmarks previously evaluated in Table A.5. Table 4 presents the total number of unique bugs identified by each fuzzer across different benchmarks. Each row corresponds to a specific benchmark, while the columns represent the performance of different fuzzing techniques, namely AFL++, CAFL++, PCGAFL++, and CPCGAFL++. The cells indicate the count of unique bugs detected by each fuzzer, with the

Table 3
Relative coverage summary for fuzzing targets. The table shows each fuzzer's median relative performance, sorted by "FuzzerMean". Green indicates the highest coverage, and blue (spectrum) represents coverage above 95%. Note: Relative coverage is calculated as trial coverage/max coverage, so the highest performance may be below 100%.

Fuzz target	AFL++	CAFL++	CPCGAFL++	PCGAFL++
bloaty_fuzz_target	92.00	97.00	97.00	96.00
curl_curl_fuzzer_http	97.00	98.00	99.00	98.00
libjpegturbo_libjpeg_turbo_fuzzer	99.00	99.00	99.00	99.00
libpng_libpng_read_fuzzer	95.00	95.00	95.00	95.00
freetype2_ftfuzzer	92.00	99.00	93.00	92.00
libxml2_xml	99.00	99.00	99.00	96.00
mbedtls_fuzz_dtlsclient	98.00	97.00	98.00	98.00
sqlite3_ossfuzz	95.00	78.00	71.00	75.00
vorbis_decode_fuzzer	99.00	99.00	99.00	99.00
woff2_convert_woff2ttf_fuzzer	94.00	97.00	97.00	96.00
zlib_zlib_uncompress_fuzzer	97.00	97.00	97.00	97.00
re2_fuzzer	99.00	98.00	99.00	98.00
jsoncpp_jsoncpp_fuzzer	100.00	100.00	100.00	99.00
harfbuzz_hb-shape-fuzzer	99	98.00	97.00	97.00
lcms_cms_transform_fuzzer	76	78	92.00	79.00
libxslt_xpath	96.00	99.00	96.00	97.00
openh264_decoder_fuzzer	98.00	98.00	98.00	98.00
openthread_otip6sendfuzzer	86.00	87.00	87.00	87.00
proj4_proj_crs_to_crs_fuzzer	91.00	91.00	89.00	87.00
stb_stbi_read_fuzzer	95.00	95.00	96.00	96.00

Table 4
Total unique bugs found on each benchmark (green background = most unique bugs).

Benchmark	AFL++	CAFL++	PCGAFL++	CPCGAFL++
harfbuzz_hb-shape-fuzzer_17863b	3	3	3	1
bloaty_fuzz_target_52948c	0	0	0	1
libxml2_xml_e85b9b	0	0	0	0

green background highlighting the highest-performing fuzzer(s) for each benchmark in terms of unique bug discovery.

For targets where the new fuzzers achieved improved coverage, such as `bloaty_fuzz_ target_52948c`, CPCGAFL++ emerged as the only fuzzer capable of identifying a unique bug.

Interestingly, on `harfbuzz_hb-shape-fuzzer_17863b`, despite lower coverage, CAFL++ and PCGAFL++ demonstrated competitive performance in identifying bugs relative to AFL++. Notably, AFL++ also detected the same number of bugs. In contrast, for `libxml2_x- ml_e85b9b`, none of the evaluated fuzzers succeeded in uncovering any bugs.

These results suggest that there is no straightforward correlation between coverage and the discovery of unique bugs within our experiments. Nonetheless, at least one of the proposed fuzzers exhibited performance comparable to or exceeding that of the baseline. These findings underscore the potential efficacy of our fuzzing strategies, particularly in scenarios where traditional coverage metrics may not directly align with bug discovery outcomes.

7. Discussion

In this section, we conduct a comparative analysis and explain why two of our proposed integrations, the Feistel-Enhanced Approach and the PCG Model, outperform the baseline, and why the Combined Mechanism (CPCGAFL++) performs similarly to the baseline in terms of coverage results.

7.1. Feistel-enhanced approach vs. baseline

The Feistel-enhanced solution represents a significant advancement in code coverage over the baseline model by combining both direct and transformed data comparisons across most targets. By improving relative code coverage by up to 7% compared to the baseline AFL++ on most targets (Table 3), the Feistel-enhanced solution further strengthens the model's capacity to identify complex arithmetic relationships in scrambled or non-linear data. The inclusion of non-linear Feistel transformations, together with additional validation checks, enables the detection of subtle arithmetic patterns that might otherwise be overlooked by linear methods. Importantly, the approach retains the baseline's capacity to handle both basic and complex data patterns effectively while also capturing more complex relationships between values.

While the Feistel-enhanced approach is generally more effective than the baseline model at detecting intricate relationships and covering more branches in code, it did not perform as well on few targets. These targets may contain specific vulnerabilities that the Feistel-enhanced method is unable to track as effectively as the baseline. Such vulnerabilities are often more easily uncovered by fuzzers like AFL++, which explores input spaces systematically to identify common corruption bugs. Additionally, some issues are less straightforward and require fuzzing strategies tailored to more complex vulnerabilities, such as constraint-based fuzzing or symbolic execution. In cases where the targets involve smaller, more structured datasets, such as SQL queries, mutation-based fuzzing can yield better results, making fuzzers like AFL++ more suited to exploring edge cases and identifying vulnerabilities that may elude the Feistel-enhanced approach. This suggests that different fuzzing techniques and models may be more appropriate for discovering specific types of vulnerabilities, and a one-size-fits-all approach may not be sufficient in all scenarios.

7.2. PCG model vs. baseline

PCG-based random number generation works effectively for fuzzing by providing more uniform and less biased random inputs, resulting in up to 16% more efficient exploration of the code compared to the baseline on most targets. This uniformity helps ensure that the random numbers generated do not favour specific inputs or paths, allowing

the fuzzing process to cover more branches in the code. The reseeding strategy, particularly using sources like /dev/urandom, introduces additional variety in the input space, enabling tools like PCG-based approach to escape local maxima more easily and continue exploring new areas of the code. By generating a diverse set of inputs, PCG-based approach demonstrates more aggressive exploration, leading to higher code coverage and the discovery of unique paths. This is supported by statistical tests, which confirm that PCG-based approach's improvements are consistent and not due to random fluctuations. The PCG-based approach enhances the fuzzing process by ensuring that random numbers are evenly distributed, eliminating biases, and improving the reliability of the RNG. This leads to better performance and more thorough exploration of the codebase, ultimately improving fuzzing efficiency.

7.3. Discussion on the combined mechanism (CPCGAFL++)

While CPCGAFL++ integrates both the Feistel-swap mechanism and the PCG-based random number generator, the results in Section 6 indicate that it does not consistently outperform either CAFL++ or PCGAFL++ individually. Although combining multiple techniques is often expected to yield superior performance, this expectation is not always met in fuzzing. As discussed in the introduction, prior work has highlighted the strong sensitivity of fuzzing performance to randomness and its interaction with mutation and scheduling strategies [47–49]. The behaviour observed in CPCGAFL++ is consistent with these findings and suggests that integrating multiple randomness-aware mechanisms can reintroduce or amplify the same stochastic effects identified in earlier studies, rather than eliminating them. Consequently, such combinations do not necessarily yield additive improvements and may remain highly dependent on program characteristics and seed selection.

In addition to the above, our research does not claim that unbiased randomness is universally optimal, as coverage metrics may vary across different runs and its effectiveness appears to be target-dependent. Wang et al. [88] noted that various coverage metrics in grey-box fuzzing offer unique benefits, and Zhao et al. [89] recapped Wang et al.'s work by stating that "no universal coverage metric outperforms all others", even when combined.

Likewise, Poncelet et al. [90], in their concluding remarks, stated that "no single fuzzing tool outperforms all the others or is able to consistently expose the bugs that exist or, worse, that it has previously discovered itself". Similarly, Sun et al. [91] highlighted that simplistic integration of fuzzing optimisations can introduce conflicting behaviours or additional overhead, potentially leading to neutral or even degraded performance. Recent works [92,93] further emphasise that fuzzing efficiency depends on numerous factors, including program characteristics, feedback configurations, and parameter tuning, making improvements from combined approaches highly context-sensitive.

In our scenario, several reasons may explain why CPCGAFL++ does not exceed its individual components:

- **Interaction effects.** The Feistel-swap and PCG-based RNG may disrupt each other's assumptions about randomness and mutation patterns, resulting in suboptimal search dynamics.
- **Computational overhead.** Combining both mechanisms adds extra processing and memory costs, reducing effective mutation throughput over time.
- **Target-specific limitations.** Each mechanism may already capture the most beneficial patterns for a given target, leaving little room for additional gains.
- **Parameter misalignment.** Parameters optimised for standalone mechanisms may not remain effective when combined, causing an imbalance between exploration and exploitation.

Thus, CPCGAFL++'s lack of dominance aligns with established findings in the fuzzing literature: integrating multiple enhancements can introduce complex interactions and diminishing returns rather than assured performance improvements. However, this does not imply that CPCGAFL++ offers no unique advantages. For example, as shown in Table 4, CPCGAFL++'s distinctive exploration strategy enabled it to uncover a vulnerability in bloaty_fuzz_target_52948c that neither CAFL++ nor PCGAFL++ could detect individually. This observation reflects that while combining mechanisms can yield benefits in specific scenarios, improved results across all aspects are not necessarily guaranteed.

8. Future work

Based on our findings, several avenues for future research and development emerge to further enhance fuzzer performance and effectiveness across diverse scenarios:

- **Enhancing Stability and Peak Performance:** While CAFL++ and CPCGAFL++ demonstrate promising stability and peak performance, their variability under certain conditions suggests the need for further refinement. Future work could explore adaptive algorithms that dynamically balance stability and performance, particularly for targets with diverse or unpredictable behaviour.
- **Optimising PCGAFL++:** The hybrid version of CAFL++ and PCGAFL++ demonstrates improved stability across a wide range of targets, outperforming the baseline in stability even in cases where its performance falls short. While it successfully balances performance and stability, there is still room for improvement to achieve top-tier performance on all targets. Future work should focus on refining this hybrid approach, particularly by exploring advanced reseeding strategies and integrating it with other techniques like symbolic or constraint-based fuzzing, to further enhance its overall effectiveness and competitiveness.
- **Exploring Non-Linear Transformations:** The Feistel-enhanced approach has demonstrated the potential of non-linear transformations for detecting complex patterns. Expanding this concept with advanced machine learning techniques or heuristic-driven transformations could further enhance its ability to uncover subtle vulnerabilities in highly obfuscated or non-linear data.
- **Improving Adaptability for Edge Cases:** Certain edge cases, such as small and highly structured datasets, require tailored fuzzing techniques. Future work could focus on adaptive frameworks that detect such edge cases during runtime and apply appropriate strategies, such as mutation-based fuzzing or specialised heuristics, to maximise coverage and bug discovery.

  By addressing these areas, future research can build upon the strengths of the current approaches while overcoming their limitations, leading to more versatile and effective fuzzing tools capable of handling the complexities of modern software systems.

9. Conclusion

In this paper, we propose three new integrated approaches, CAFL++, PCGAFL++, and CPCGAFL++, and evaluate them against the baseline, AFL++. Our findings indicate that CAFL++ significantly outperforms AFL++ across most targets, securing first-place rankings in the majority of tests. PCGAFL++, which ranks second, shows promise with balanced performance, though further improvements are needed to match CAFL++'s peak capabilities. CPCGAFL++, on the other hand, offers notable stability, consistently securing first and second-place rankings more often than the baseline, with lower variability and fewer instances of fourth-place finishes. However, AFL++ remains competitive in some targets, placing it and CPCGAFL++ in nearly equal positions.

The analysis of relative coverage and the Vargha-Delaney A12 statistic supports these conclusions, highlighting the stability of both CPCGAFL++ and CAFL++, and demonstrating that all three new fuzzers outperform AFL++ in many cases. Furthermore, the discovery

Table A.5
Fuzzer coverage statistics.

Fuzz target	Fuzzer	Mean	Std	Min	25%	Median	75%	Max	Experiment findings
bloaty_fuzz_target	CAFL++	5836.6	75.909815	5757	5809	5822	5833	5962	CAFL++ provides the best overall performance, with the highest mean and median, though it comes with higher variability. CPCGAFL++ is a close second, with a slightly lower mean but the best consistency in results. PCGAFL++ and AFL++ show lower overall performance, with AFL++ ranking the lowest due to both lower average performance and higher variability. Thus, based on the analysis, the ranking is as follows: CAFL++, CPCGAFL++, PCGAFL++, AFL++.
	CPCGAFL++	5800.8	32.736829	5752	5784	5812	5825	5831	
	PCGAFL++	5761.4	51.383850	5696	5727	5775	5780	5829	
	AFL++	5497.4	64.103822	5431	5436	5503	5539	5578	
curl_curl_fuzzer_http	CPCGAFL++	10 421.2	72.685624	10 300	10 408	10 456	10 464	10 478	CPCGAFL++ emerges as the best-performing fuzzer, with the highest mean, median, and good consistency, making it the top choice for this target. CAFL++ delivers strong performance with impressive consistency, but it lags slightly behind CPCGAFL++ in terms of mean and median values. PCGAFL++ exhibits more variability in its results, with a wider range, but it still performs reasonably well in comparison. AFL++ shows the least consistency and the lowest performance, with the smallest mean and median, making it the least favourable option for this target. Thus, the fuzzers rank as follows: CPCGAFL++, CAFL++, PCGAFL++, AFL++.
	CAFL++	10 400.0	25.465663	10 375	10 381	10 390	10 422	10 432	
	PCGAFL++	10 392.25	101.200708	10 280	10 355	10 381.5	10 418.75	10 526	
	AFL++	10 264.0	59.987499	10 210	10 227	10 236	10 290	10 357	
libjpeg-turbo_libjpeg_turbo_fuzzer	CAFL++	2554.4	0.894427	2554	2554	2554	2554	2556	The fuzzers exhibit nearly indistinguishable performance, prompting us to focus on their consistency to determine the best performer. CAFL++ stands out as the top performer, demonstrating the highest mean and median values along with the best consistency. CPCGAFL++ and PCGAFL++ show similar levels of consistency but slightly lower mean and median values. AFL++, while still competitive, ranks last due to slightly less consistency, despite marginally better mean and median values. These differences, however, are negligible. The rankings, with a focus on consistency for libjpeg-turbo_libjpeg_turbo_fuzzer, are: CAFL++, CPCGAFL++, PCGAFL++, AFL++
	CPCGAFL++	2553.6	0.894427	2553	2553	2553	2554	2555	
	PCGAFL++	2553.4	0.894427	2553	2553	2553	2553	2555	
	AFL++	2553.8	1.643168	2552	2553	2553	2555	2556	
libpng_libpng_read_fuzzer	CAFL++	1993.2	13.141537	1970	1996	1999	1999	2002	CAFL++ offers a balance of strong average performance, stability, and consistency, making it the top choice. PCGAFL++ has the highest average performance but suffers from variability. AFL++ is the most consistent but has weaker overall performance. CPCGAFL++ lags behind in both performance and consistency. The rankings for libpng_libpng_read_fuzzer are: CAFL++, PCGAFL++, AFL++, CPCGAFL++.
	PCGAFL++	2005.0	46.276344	1965	1980	1997	1999	2084	
	AFL++	1987.4	10.163661	1979	1980	1981	1998	1999	
	CPCGAFL++	1982.6	20.218803	1952	1979	1980	2000	2002	
freetype2_ftfuzzer	CAFL++	9664.0	373.237190	9129	9411	9925	9926	9929	CAFL++ excels in mean and median performance, making it the best overall. CPCGAFL++ shines in consistency and stability, making it a solid second choice. PCGAFL++ performs well but is less consistent than CPCGAFL++. AFL++ consistently underperforms compared to the others, with the lowest mean and highest variability. The rankings for freetype2_ftfuzzer are: CAFL++, CPCGAFL++, PCGAFL++, AFL++.
	CPCGAFL++	9425.0	163.180881	9242	9338	9363	9534	9648	
	PCGAFL++	9418.8	327.199786	9244	9248	9291	9309	10 002	
	AFL++	9169.6	385.393695	8672	8918	9211	9399	9648	
libxml2_xml	AFL++	15 299.8	49.806626	15 253	15 256	15 285	15 348	15 357	AFL++ delivers the best overall performance, combining high average scores with excellent consistency and stability. CAFL++ comes close but is slightly less consistent, making it second. CPCGAFL++ is a solid performer but falls behind in both mean and stability. PCGAFL++ underperforms significantly, showing high variability and weaker results overall. The rankings for libxml2_xml are: AFL++, CAFL++, CPCGAFL++, PCGAFL++.
	CAFL++	15 244.6	146.932978	14 982	15 303	15 307	15 311	15 320	
	CPCGAFL++	15 199.8	108.520505	15 017	15 187	15 247	15 262	15 286	
	PCGAFL++	14 909.0	327.670566	14 616	14 627	14 782	15 232	15 288	
mbedtls_fuzz_dtlsclient	CPCGAFL++	2671.0	16.416455	2651	2661	2668	2684	2691	CPCGAFL++ is the clear leader, with the best overall performance and highest consistency. AFL++ provides competitive results but is less consistent and stable. PCGAFL++ shows decent performance but suffers from variability. CAFL++ underperforms in terms of mean and median, though it remains relatively consistent. The rankings for mbedtls_fuzz_dtlsclient are: CPCGAFL++, AFL++, PCGAFL++, CAFL++.
	AFL++	2661.2	27.224989	2623	2648	2663	2679	2693	
	PCGAFL++	2656.2	29.592229	2628	2634	2655	2661	2703	
	CAFL++	2650.0	23.054284	2629	2630	2644	2665	2682	
sqlite3_ossfuzz	AFL++	17 264.4	488.982413	16 774	16 883	17 101	17 719	17 845	AFL++ is the clear winner, excelling in all key metrics and maintaining consistent, stable performance. CAFL++ is a strong second but lags significantly behind AFL++. PCGAFL++ and CPCGAFL++ show similar trends, with CPCGAFL++performing the worst due to its low mean and high variability. The rankings for sqlite3_ossfuzz are: AFL++, CAFL++, PCGAFL++, CPCGAFL++.
	CAFL++	13 744.0	826.063860	12 705	13 162	13 948	14 088	14 817	
	PCGAFL++	13 239.0	896.512409	12 198	12 601	13 393	13 483	14 520	
	CPCGAFL++	12 593.6	975.245508	11 593	11 755	12 773	12 836	14 011	

(*continued on next page*)

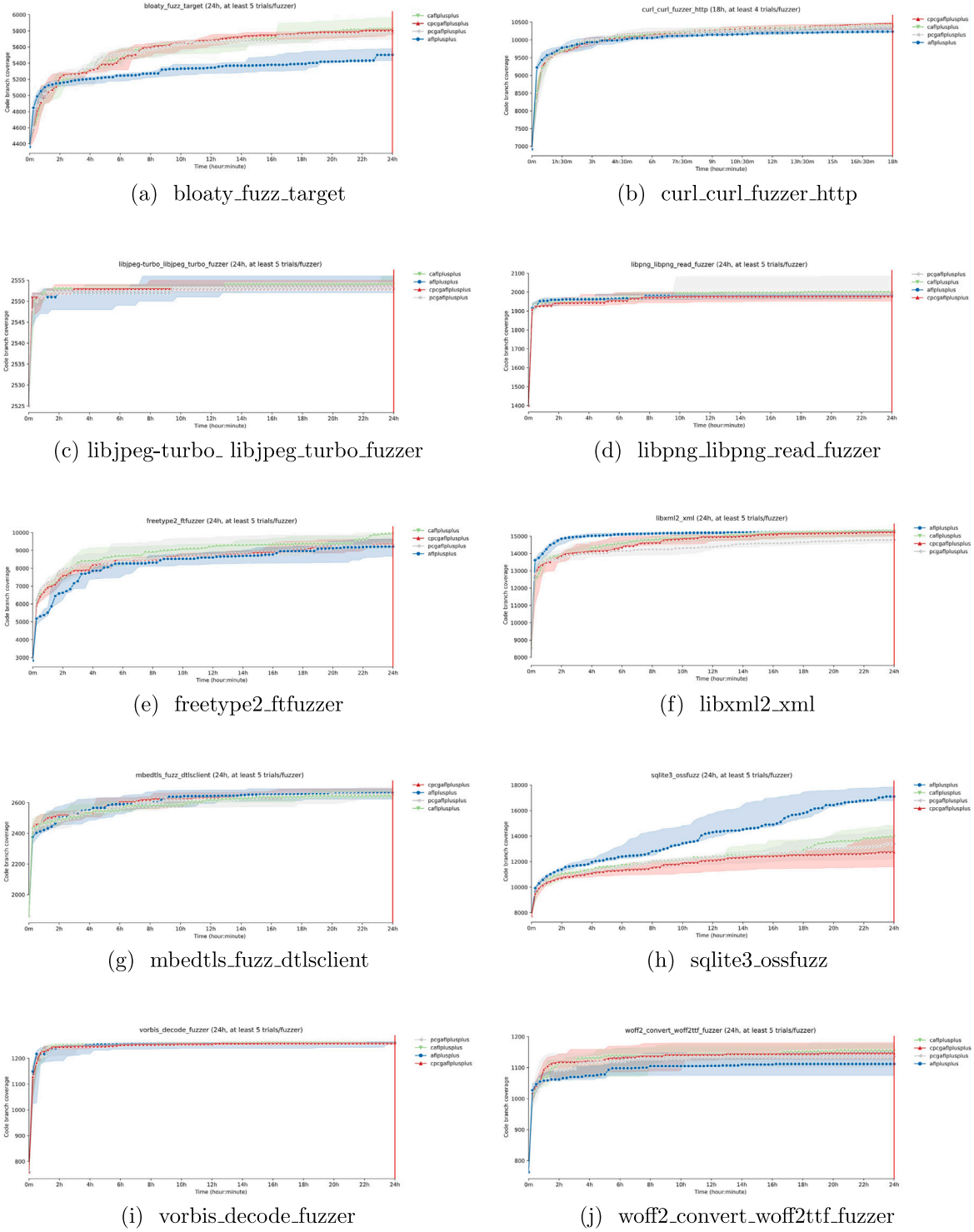
Table A.5 (*continued*).

vorbis_decode_fuzzer	PCGAFL++	1261.4	3.286335	1256	1261	1262	1264	1264	PCGAFL++ takes the lead with superior mean and median performance,
	CAFL++	1260.6	2.607681	1258	1258	1261	1262	1264	despite being slightly less consistent compared to the second-ranked
	AFL++	1258.6	3.781534	1253	1257	1260	1260	1263	CAFL++. CAFL++, on the other hand, delivers the highest consistency and
	CPCGAFL++	1258.8	4.324350	1254	1255	1259	1262	1264	stability, though it lags marginally in terms of mean and median performance. AFL++ and CPCGAFL++ perform closely, with CPCGAFL++ showing slightly more variability, placing it last. The rankings for vorbis_decode_fuzzer are: PCGAFL++, CAFL++, AFL++, CPCGAFL++.
woff2_convert_ woff2ttf_fuzzer	CAFL++	1154.0	13.693064	1142	1143	1154	1155	1176	CAFL++ is the clear leader with the best performance across all metrics.
	CPCGAFL++	1151.4	19.475626	1127	1145	1147	1158	1180	CPCGAFL++ is competitive but less consistent. PCGAFL++ and AFL++ fall
	PCGAFL++	1140.0	30.033315	1103	1125	1135	1155	1182	behind due to lower averages and higher variability. The rankings for
	AFL++	1110.2	28.778464	1075	1094	1112	1118	1152	woff2_convert_woff2ttf_fuzzer are: CAFL++, CPCGAFL++, PCGAFL++, AFL++.
zlib_zlib_uncompress _fuzzer	CAFL++	457.4	0.894427	457	457	457	457	459	CAFL++ delivers the best overall performance with a balance of high
	AFL++	457.0	1.224745	456	456	457	457	459	average, low variability, and strong stability. AFL++ has marginally lower
	PCGAFL++	458.4	2.509980	456	457	457	460	462	averages but offers good consistency and range. Although their means
	CPCGAFL++	458.6	5.319774	455	456	457	457	468	are slightly improved, their inconsistency resulted in rankings of third and fourth, respectively. The rankings for zlib_zlib_uncompress_fuzzer are: CAFL++, AFL++, PCGAFL++, CPCGAFL++.
harfbuzz_hb-shape- fuzzer	CPCGAFL++	10 376.8	80.530739	10 284	10 338	10 372	10 388	10 502	CPCGAFL++ is highly consistent and delivers the best overall performance.
	CAFL++	10 339.4	121.389044	10 237	10 241	10 286	10 424	10 509	CAFL++ shows strong performance, but it is less consistent compared to
	PCGAFL++	10 291.8	119.103736	10 109	10 252	10 317	10 362	10 419	CPCGAFL++. PCGAFL++ is balanced, but this fuzzer does not stand out in
	AFL++	10 088.2	848.121277	8595	10 203	10 521	10 558	10 564	terms of either performance or consistency. AFL++ is the least competitive, with low mean performance and high variability. For the harfbuzz_hb-shape-fuzzer target, the ranking is as follows:based on performance metrics is: CPCGAFL++, CAFL++, PCGAFL++, AFL++.
jsoncpp_jsoncpp _fuzzer	CAFL++	520.0	0.000000	520	520	520	520	520	The fuzzers demonstrate virtually identical performance, prompting us
	AFL++	519.8	0.447214	519	520	520	520	520	to emphasise their consistency to identify the best performer. Among
	CPCGAFL++	519.8	0.447214	519	520	520	520	520	them, CAFL++ is the most consistent and performs identically on every run, making it the top performer
	PCGAFL++	519.2	0.836660	518	519	519	520	520	here. AFL++ and CPCGAFL++ have very similar performance in terms of mean, median, and variability, but AFL++ has a slight edge in the mean. PCGAFL++ performs slightly worse in of all metrics and shows the most fluctuation in results. For the jsoncpp_ jsoncpp_fuzzer target, the ranking based on performance metrics is: CAFL++, AFL++, CPCGAFL++, PCGAFL++.
re2_fuzzer	AFL++	2852.4	14.397917	2837	2844	2846	2864	2871	AFL++ leads in terms of both mean and median performance, with
	CPCGAFL++	2837.8	22.509998	2811	2817	2845	2856	2860	moderate variability and range. CPCGAFL++ comes close in terms of
	PCGAFL++	2835.8	7.120393	2827	2830	2837	2842	2843	median but has more variability and a larger range than AFL++.
	CAFL++	2833.2	30.044966	2787	2828	2831	2858	2862	PCGAFL++ shows the lowest variability and range, making it very stable, though it has a slightly lower mean and median. CAFL++ has the most variability and a larger range, which results in lower overall performance compared to the other fuzzers. For the re2_fuzzer target, the ranking based on performance metrics is: AFL++, CPCGAFL++, PCGAFL++ CAFL++.
lcms_cms_transform _fuzzer	PCGAFL++	1612.6	163.790415	1461	1509	1534	1703	1856	PCGAFL++ leads in terms of mean and has a moderate range and
	CPCGAFL++	1557.8	442.789679	937	1244	1791	1905	1912	variability, placing it first. CPCGAFL++ has the highest median but suffers
	CAFL++	1516.8	55.445469	1428	1513	1517	1555	1571	from high variability and a large range, which affects its overall
	AFL++	1486.2	417.344821	879	1334	1475	1809	1934	performance ranking. CAFL++ shows excellent consistency (low variability and range), but its mean and median are slightly lower than the top two fuzzers. with the lowest mean and median, as well as the largest range AFL++ shows the weakest overall performance, and high variability. For the lcms_cms_transform_fuzzer target, the ranking based on performance metrics is: PCGAFL++, CPCGAFL++, CAFL++, AFL++.

(*continued on next page*)

Table A.5 (*continued*).

openthread_ot-ip6-send-fuzzer	CAFL++	3165.4	177.140622	3025	3035	3049	3347	3371	CAFL++ performs best on average, with strong mean and median values, but has slightly more fluctuation compared to others. AFL++ follows closely in terms of mean, but its high variability and range suggest it might sometimes show more extreme results. CPCGAFL++ and PCGAFL++ are consistent in their performance, with CPCGAFL++ showing the lowest variability, but their performance is not as high as CAFL++ or AFL++. For the openthread_ot-ip6send-fuzzer target, the fuzzers rank as follows: CAFL++, AFL++, CPCGAFL++,PCGAFL++.
	AFL++	3115.4	210.514370	2998	3019	3031	3038	3491	
	CPCGAFL++	3044.6	10.922454	3034	3036	3041	3053	3059	
	PCGAFL++	3034.8	10.473777	3018	3031	3041	3042	3042	
libxslt_xpath	CAFL++	10 647.6	118.822557	10 463	10 607	10 676	10 729	10 763	CAFL++ exhibits the strongest overall performance, with the highest mean and median values, along with the best consistency. CPCGAFL++ performs better than AFL++ in mean, but it exhibits higher variability. While AFL++ is consistent, its mean and median values are lower than those of PCGAFL++. In contrast, CPCGAFL++ shows the highest variability, the lowest mean, and inconsistent performance. In conclusion, CAFL++ stands out with the best overall performance, while AFL++ excels in consistency. However, despite a reasonable median, CPCGAFL++ is the least reliable due to its higher variability. For the libxslt_xpath target, the fuzzers rank as follows: CAFL++, PCGAFL++, AFL++, CPCGAFL++.
	PCGAFL++	10 502.6	94.030846	10 380	10 444	10 500	10 593	10 596	
	AFL++	10 407.4	62.078982	10 324	10 372	10 408	10 456	10 477	
	CPCGAFL++	10 257.8	349.988857	9634	10 365	10 414	10 436	10 440	
openh264_decoder_fuzzer	CAFL++	9296.80	40.319970	9240	9285	9303	9304	9352	CAFL++ leads with the best performance, showing both the highest mean and median and the lowest variability. PCGAFL++ shows strong performance with slightly more variability than CAFL++. AFL++ shows reasonable performance but with higher variability, which makes it less consistent. CPCGAFL++ ranks last in both mean and median and also has higher variability, but its performance is still comparable to the others in terms of range. For the openh264_decoder_fuzzer target, the fuzzers rank as follows: CAFL++,PCGAFL++, AFL++, CPCGAFL++.
	PCGAFL++	9293.20	56.131097	9230	9258	9287	9315	9376	
	AFL++	9271.75	111.008633	9144	9232.5	9264	9303.25	9415	
	CPCGAFL++	9230.00	96.049466	9115	9144	9254	9314	9323	
proj4_proj_crs_to_crs_fuzzer	AFL++	5361.0	472.62	4735.0	5080.0	5353.0	5774.0	5863.0	AFL++ remains the best overall due to its highest mean and strong performance at the upper percentiles, despite a slightly lower median. CPCGAFL++ is ideal for those prioritising consistency, as it has the lowest standard deviation and high stability across the entire range with better mean than CAFL++. CAFL++ ranks third, excelling in median and upper-percentile performance but slightly lagging behind AFL++ and PCGAFL++ in overall mean performance and exhibiting more inconsistency. PCGAFL++ performs well but lags behind in both mean and median, making it the least favourable among the four. For the proj4_proj_crs_to_crs_fuzzer decoder fuzzer target, the fuzzers rank as follows: AFL++, CPCGAFL++, CAFL++, PCGAFL++.
	CPCGAFL++	5344.4	183.56	5214.0	5252.0	5253.0	5341.0	5662.0	
	CAFL++	5282.6	380.97	4698.0	5155.0	5373.0	5491.0	5696.0	
	PCGAFL++	5264.8	410.24	4742.0	5078.0	5141.0	5647.0	5716.0	
stb_stbi_read_fuzzer	CPCGAFL++	2119.4	31.020961	2087.0	2111.0	2112.0	2116.0	2171.0	PCGAFL++ has the highest mean, max, and min coverage, making it the most consistent and best overall performer. CPCGAFL++ shows strong median coverage, ranking second in mean and max, making it better for consistency. CAFL++ delivers decent performance but has lower min coverage, indicating it struggles on some runs. AFL++ has the lowest mean, median, max, and min coverage, making it the least effective fuzzer. Ranks are as follow: PCGAFL++, CPCGAFL++, CAFL++, AFL++.
	PCGAFL++	2127.00	45.216148	2086	2091.0	2109.0	2161.0	2188.0	
	CAFL++	2096.20	16.483325	2080.0	2086.0	2087.0	2113.0	2115.0	
	AFL++	2060.8	37.851024	1994	2067.0	2079.0	2082.0	2082.0	



(a) bloaty_fuzz_target (b) curl_curl_fuzzer_http

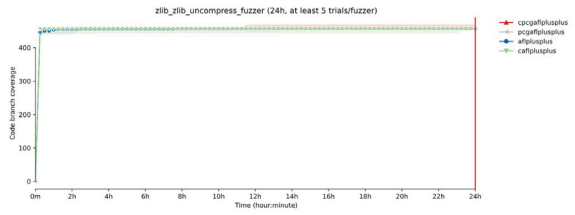
(c) libjpeg-turbo_ libjpeg_turbo_fuzzer (d) libpng_libpng_read_fuzzer

(e) freetype2_ftfuzzer (f) libxml2_xml

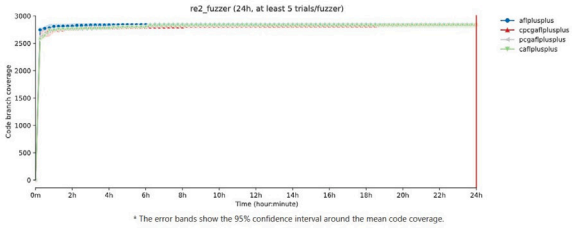
(g) mbedtls_fuzz_dtlsclient (h) sqlite3_ossfuzz

(i) vorbis_decode_fuzzer (j) woff2_convert_woff2ttf_fuzzer

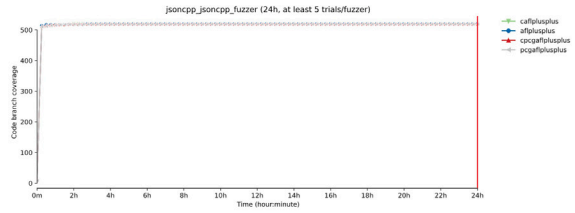
Fig. B.2. (a–j) Comparison of Code Coverage Across Targets within 24 h (part 1). An exception was observed with curl curl fuzzer http, which, despite a 24-hour intended execution, produced results for only 18 h. The premature conclusion of the run was probably attributed to a process limitation in the target's configuration or nature, confirmed through repeat testing under identical experimental conditions used for other targets. (k–t) Comparison of Code Coverage Across Targets within 24 h (part 2). Same settings as Fig. B.2 (part 1).



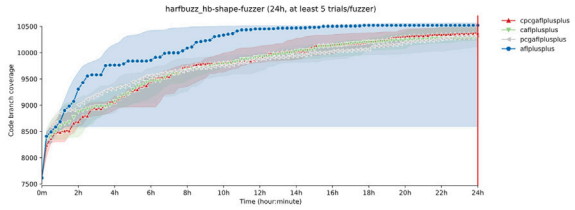
(k) zlib_zlib_uncompress_fuzzer



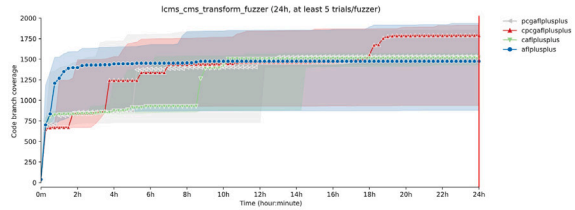
(l) re2_fuzzer



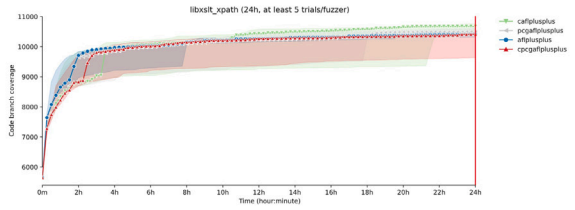
(m) jsoncpp_jsoncpp_fuzzer



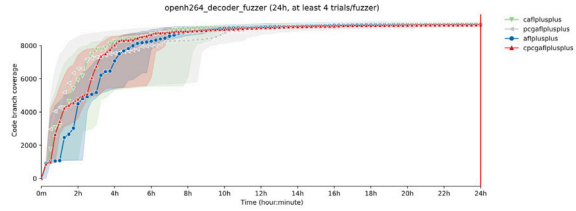
(n) harfbuzz_hb-shape-fuzzer



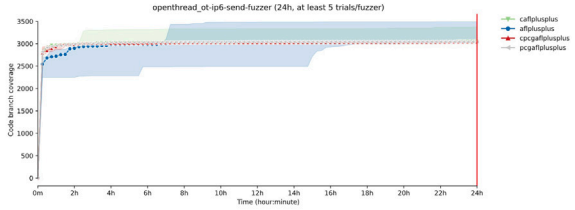
(o) lcms_cms_transform_fuzzer



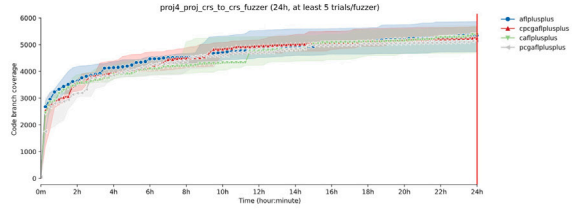
(p) libxslt_xpath



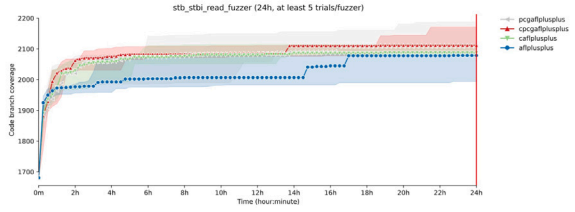
(q) openh264_decoder_fuzzer



(r) openthread_ot-ip6-send-fuzzer



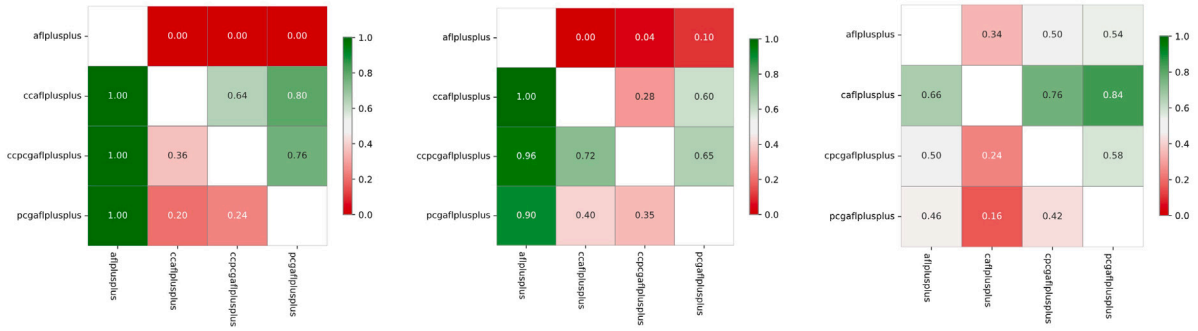
(s) proj4_proj_crs_to_crs_fuzzer



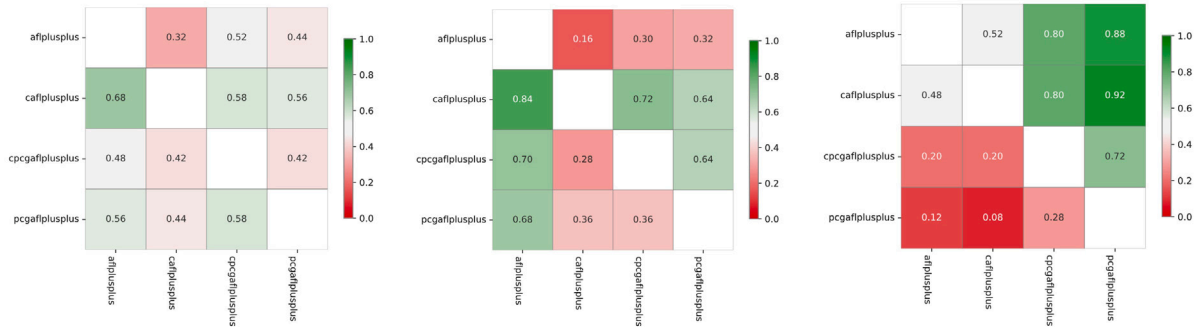
(t) stb_stbi_read_fuzzer

Fig. B.2. (*continued*).

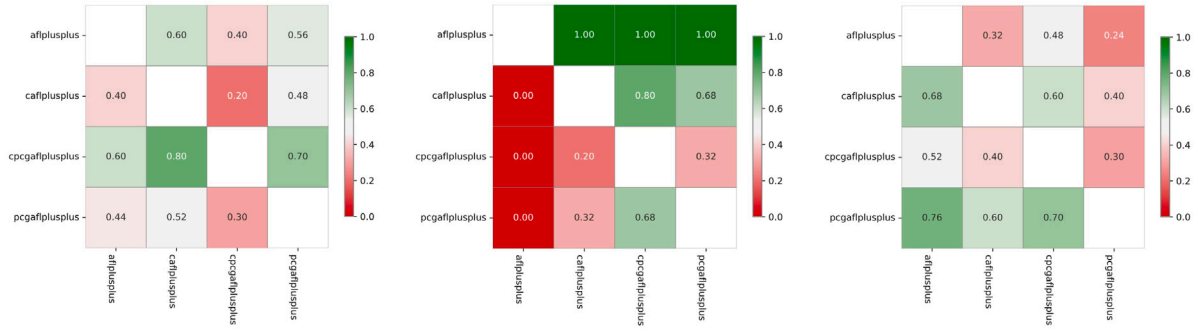
of unique bugs by these fuzzers, especially by CPCGAFL++, emphasises their practical value beyond just coverage metrics. These results show that the integrated fuzzers provide significant improvements over traditional approaches, making them highly effective tools for software testing and vulnerability discovery.



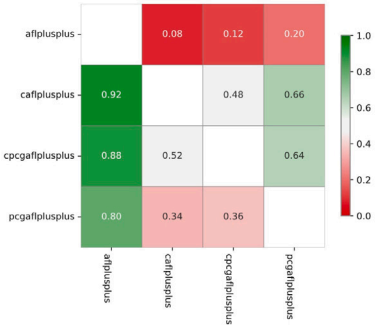
(a) bloaty_fuzz_target (b) curl_curl_fuzzer_http (c) libjpeg-turbo_libjpeg_turbo_fuzzer



(d) libpng_libpng_read_fuzzer (e) freetype2_ftfuzzer (f) libxml2_xml

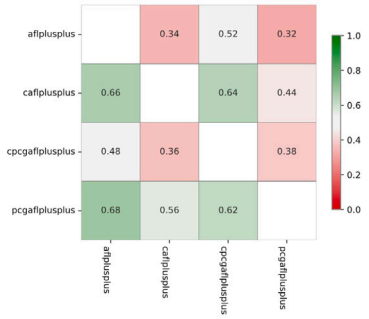


(g) mbedtls_fuzz_dtlsclient (h) sqlite3_ossfuzz (i) vorbis_decode_fuzzer

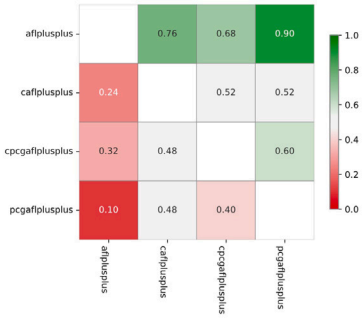


(j) woff2_convert_woff2ttf_fuzzer

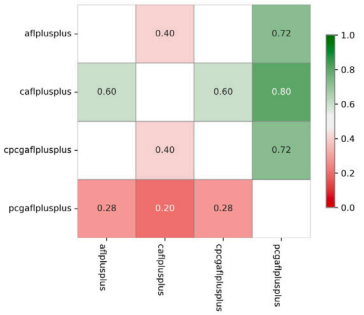
Fig. C.3. (a–j) Comparison of Vargha-Delaney A12 statistic Across Targets (part 1). (k–t) Comparison of Vargha-Delaney A12 statistic Across Targets (part 2): No values observed for CPCGAFL++ vs. AFL++ on jsoncpp jsoncpp, even after repeated experiments.



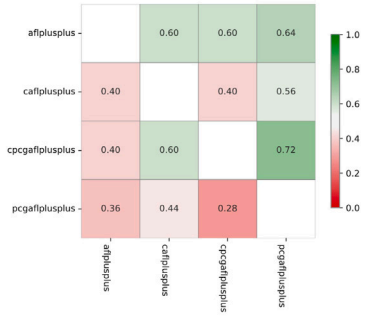
(k) zlib_zlib_uncompress_fuzzer



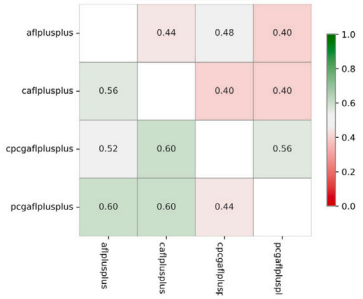
(l) re2_fuzzer



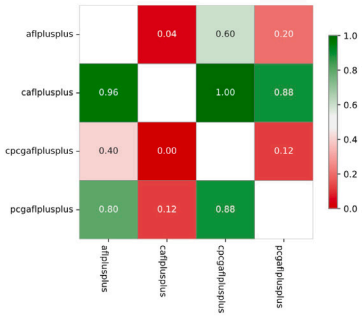
(m) jsoncpp_jsoncpp_fuzzer



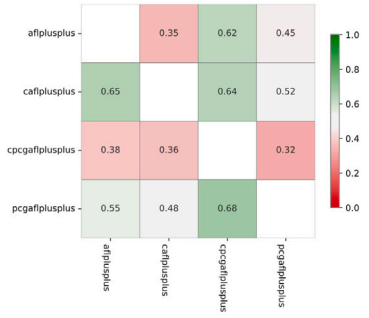
(n) harfbuzz_hb-shape-fuzzer



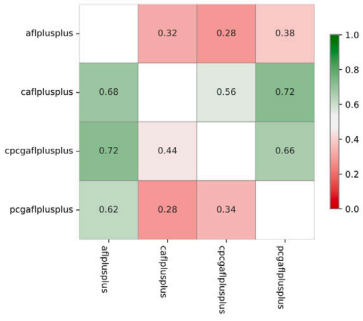
(o) lcms_cms_transform_fuzzer



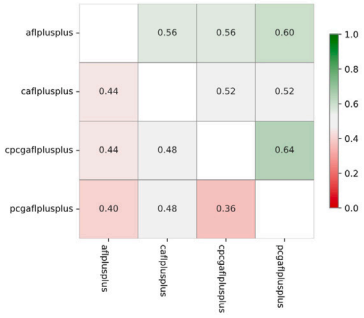
(p) libxslt_xpath



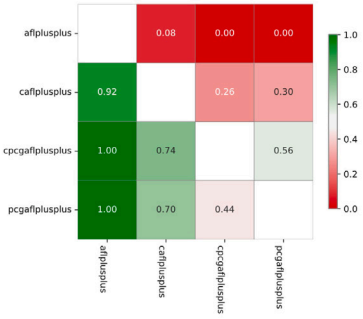
(q) openh264_decoder_fuzzer



(r) openthread_ot-ip6-send-fuzzer



(s) proj4_proj_crs_to_crs_fuzzer



(t) stb_stbi_read_fuzzer

Fig. C.3. (*continued*).

CRediT authorship contribution statement

Sadegh Bamohabbat Chafjiri: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Data curation, Conceptualization. **Phil Legg:** Writing – review & editing, Validation, Supervision, Data curation. **Jun Hong:** Writing – review & editing, Validation, Supervision, Data curation. **Michail-Antisthenis Tsompanas:** Writing – review & editing, Validation, Supervision, Data curation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research was supported by the PhD Studentship scheme within College of Arts, Technology and Engineering at the University of the West of England. The corresponding author, Sadegh Bamohabbat Chafjiri, was at University of the West of England when the research was conducted and has since joined Birmingham City University.

Appendix A. Coverage statistics on 20 Fuzzbench targets

This appendix contains supplementary details on each fuzzer's coverage statistics on different targets.

Appendix B. Comparison of code coverage across targets within 24 h

This appendix contains figures comparing code coverage across targets within 24 h.

Appendix C. Comparison of *Vargha-Delaney_A12 statistic* across targets

This appendix contains figures comparing the *Vargha–Delaney_A12* statistic across targets within 24 h.

Data availability

Data will be made available on request.

References

[1] B.P. Miller, L. Fredriksen, B. So, An empirical study of the reliability of UNIX utilities, Commun. ACM 33 (12) (1990) 32–44, http://dx.doi.org/10.1145/96267.96279.

[2] M. Woo, S.K. Cha, S. Gottlieb, D. Brumley, Scheduling black-box mutational fuzzing, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 511–522, http://dx.doi.org/10.1145/2508859.2516736.

[3] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, J. Steidl, Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services, Tech. Rep., Computer Sciences Department, University of Wisconsin-Madison, 1995.

[4] D. Molnar, P. Godefroid, M. Levin, Automated whitebox fuzz testing, in: Network and Distributed System Security Symposium, NDSS, 2008, pp. 416–426.

[5] M. Böhme, V.-T. Pham, M.-D. Nguyen, A. Roychoudhury, Directed greybox fuzzing, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 2329–2344.

[6] H.J. Abdelnur, R. State, O. Festor, KiF: a stateful SIP fuzzer, in: Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications, IPTComm '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 47–56, http://dx.doi.org/10.1145/1326304.1326313.

[7] A. Biyani, G. Sharma, J. Aghav, P. Waradpande, P. Savaji, M. Gautam, Extension of SPIKE for encrypted protocol fuzzing, in: 2011 Third International Conference on Multimedia Information Networking and Security, IEEE, 2011, pp. 343–347.

[8] L. Juranić, Using Fuzzing to Detect Security Vulnerabilities, Tech. Rep. INFIGO-TD-01-04-2006, 2006.

[9] Ombagi, Time-based blind SQL injection via HTTP headers: Fuzzing and exploitation, in: 2017 Strathmore Research Symposium, Nairobi, Kenya, 2017.

[10] J. de Ruiter, E. Poll, Protocol state fuzzing of TLS implementations, in: 24th USENIX Security Symposium, USENIX Security 15, USENIX Association, Washington, D.C., 2015, pp. 193–206, URL https://www.usenix.org/conference/usenixsecurity15/technical-ses-sions/presentation/de-ruiter.

[11] M. Sutton, A. Greene, P. Amini, Fuzzing: Brute Force Vulnerability Discovery, Addison-Wesley Professional, 2007.

[12] A. Takanen, J.D. Demott, C. Miller, Fuzzing for Software Security Testing and Quality Assurance, second ed., Artech House, 2018.

[13] I. Haller, A. Slowinska, M. Neugschwandtner, H. Bos, Dowsing for overflows: A guided fuzzer to find buffer boundary violations, in: 22nd USENIX Security Symposium, USENIX Security 13, USENIX Association, Washington, D.C., 2013, pp. 49–64, URL https://www.usenix.org/conference/usenixsecurity13/technical-ses-sions/papers/haller.

[14] Peach fuzzer, 2023, https://peachfuzzer.com/. (Last Accessed 27 May 2023).

[15] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, G. Vigna, SNOOZE: toward a stateful NetwOrk protocol fuzZEr, in: Proceedings of the Information Security Conference, 2006.

[16] P. Godefroid, A. Kiezun, M.Y. Levin, Grammar-based whitebox fuzzing, in: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2008, pp. 206–215.

[17] V.-T. Pham, M. Böhme, A. Roychoudhury, Model-based whitebox fuzzing for program binaries, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 543–553.

[18] S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, S. Asryan, Grammar-based fuzzing, in: 2018 Ivannikov Memorial Workshop, IVMEM, 2018, pp. 32–35, http://dx.doi.org/10.1109/IVMEM.2018.00013.

[19] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna, Driller: Augmenting fuzzing through selective symbolic execution, NDSS 2016, in: 23rd Annual Network and Distributed System Security Symposium, vol. 16, The Internet Society, 2016, pp. 1–16, http://dx.doi.org/10.14722/ndss.2016.23368, Publisher Copyright: © 2016 Internet Society.; 23rd Annual Network and Distributed System Security Symposium, NDSS 2016 ; Conference date: 21-02-2016 Through 24-02-2016.

[20] M. Zalewski, American fuzzy lop (AFL), 2013, https://lcamtuf.coredump.cx/afl/. (Last Accessed 27 May 2023).

[21] M. Böhme, V. Pham, A. Roychoudhury, Coverage-based greybox fuzzing as Markov chain, IEEE Trans. Softw. Eng. 45 (5) (2019) 489–506, http://dx.doi.org/10.1109/TSE.2017.2785841.

[22] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, Z. Chen, CollAFL: Path sensitive fuzzing, in: 2018 IEEE Symposium on Security and Privacy, SP, 2018, pp. 679–696, http://dx.doi.org/10.1109/SP.2018.00040.

[23] S. Yan, C. Wu, H. Li, W. Shao, C. Jia, PathAFL: Path-coverage assisted fuzzing, in: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, in: ASIA CCS '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 598–609, http://dx.doi.org/10.1145/3320269.3384736.

[24] M. Felderer, M. Büchler, M. Johns, A.D. Brucker, R. Breu, A. Pretschner, Chapter one - security testing: A survey, in: A. Memon (Ed.), in: Advances in Computers, vol. 101, Elsevier, 2016, pp. 1–51, http://dx.doi.org/10.1016/bs.adcom.2015.11.003.

[25] C. Miller, Z.N.J. Peterson, Analysis of Mutation and Generation-Based Fuzzing, Tech. Rep., 2007.

[26] X. Zhu, S. Wen, S. Camtepe, Y. Xiang, Fuzzing: A survey for roadmap, ACM Comput. Surv. 54 (11s) (2022) 1–36, http://dx.doi.org/10.1145/3512345.

[27] X. Wei, Z. Yan, X. Liang, A survey on fuzz testing technologies for industrial control protocols, J. Netw. Comput. Appl. 232 (2024) 104020, http://dx.doi.org/10.1016/j.jnca.2024.104020, URL https://www.sciencedirect.com/science/article/pii/S1084804524001978.

[28] X. Han, Q. Wen, Z. Zhang, A mutation-based fuzz testing approach for network protocol vulnerability detection, in: Proceedings of 2012 2nd International Conference on Computer Science and Network Technology, IEEE, 2012, pp. 1018–1022.

[29] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, R. Beyah, MOPT: Optimized mutation scheduling for fuzzers, in: 28th USENIX Security Symposium, USENIX Security 19, 2019, pp. 1949–1966.

[30] Y. Koike, H. Katsura, H. Yakura, Y. Kurogome, SLOPT: Bandit optimization framework for mutation-based fuzzing, in: Proceedings of the 38th Annual Computer Security Applications Conference, ACSAC '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 519–533, http://dx.doi.org/10.1145/3564625.3564659.

[31] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, A. Zeller, Parser-directed fuzzing, in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI 2019, Association for Computing Machinery, New York, NY, USA, 2019, pp. 548–560, http://dx.doi.org/10.1145/3314221.3314651.

[32] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, P. Su, Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization, in: Proceedings 2020 Network and Distributed System Security Symposium, 2020, URL https://api.semanticscholar.org/CorpusID:211268394.

[33] S.K. Cha, M. Woo, D. Brumley, Program-adaptive mutational fuzzing, in: 2015 IEEE Symposium on Security and Privacy, 2015, pp. 725–741, http://dx.doi.org/10.1109/SP.2015.50.

[34] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, T. Holz, kAFL: Hardware-assisted feedback fuzzing for OS Kernels, in: 26th USENIX Security Symposium, USENIX Security 17, USENIX Association, Vancouver, BC, 2017, pp. 167–182, URL https://www.usenix.org/conference/usenixsecurity17/technical-ses-sions/presentation/schumilo.

[35] S. Bamohabbat Chafjiri, P. Legg, M.-A. Tsompanas, J. Hong, Improving search space analysis of fuzzing mutators using cryptographic structures, in: C. Hewage, L. Nawaf, N. Kesswani (Eds.), AI Applications in Cyber Security and Communication Networks, Springer Nature Singapore, Singapore, 2024, pp. 153–172.

[36] M.E. O'neill, PCG: A family of simple fast space-efficient statistically good algorithms for random number generation, ACM Trans. Math. Software (2014).

[37] C. Lemieux, K. Sen, Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 475–485, http://dx.doi.org/10.1145/3238147.3238176.

[38] H. Feistel, Cryptography and computer privacy, Sci. Am. 228 (5) (1973) 15–23.

[39] R. Swiecki, Honggfuzz, 2024, https://github.com/AFLplusplus/AFLplusplus/blob/stable/custom_mutators/honggfuzz/mangle.c. (Last Accessed 20 Oct 2024).

[40] J. Metzman, L. Szekeres, L. Maurice Romain Simon, R. Trevelin Sprabery, 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, in: ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1393–1403, http://dx.doi.org/10.1145/3468264.3473932.

[41] C.E. Shannon, Communication theory of secrecy systems, 1949, http://dx.doi.org/10.1002/j.1538-7305.1949.tb00928.x, Vol. 28.

[42] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, H. Bos, VUzzer: Application-aware evolutionary fuzzing, in: Network and Distributed System Security Symposium, 2017, URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/.

[43] A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse, Afl++: Combining incremental steps of fuzzing research, in: 14th USENIX Workshop on Offensive Technologies, WOOT 20, 2020.

[44] S. Jeon, J. Moon, Dr.PathFinder: hybrid fuzzing with deep reinforcement concolic execution toward deeper path-first search, Neural Comput. Appl. 34 (13) (2022) 10731–10750, http://dx.doi.org/10.1007/s00521-022-07008-8.

[45] AFL++Project, Afl-fuzz.h — Afl++ source code (stable branch), 2024, https://github.com/AFLplusplus/AFLplusplus/blob/stable/include/afl-fuzz.h. (Accessed 16 February 2026).

[46] G. Marsaglia, Random Numbers Fall Mainly in the Planes, Tech. Rep. Mathematical Note No. 582, Boeing Scientific Research Laboratories, 1968, URL https://apps.dtic.mil/sti/tr/pdf/AD0685578.pdf.

[47] G. Klees, A. Ruef, B. Cooper, S. Wei, M. Hicks, Evaluating fuzz testing, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, Association for Computing Machinery, 2018, pp. 2123–2138, http://dx.doi.org/10.1145/3243734.3243804.

[48] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, T. Holz, SoK: Prudent evaluation practices for fuzzing, in: 2024 IEEE Symposium on Security and Privacy, SP, 2024, pp. 1974–1993, http://dx.doi.org/10.1109/SP54263.2024.00137.

[49] X. Feng, X. Zhu, K. Hu, J. Wang, Y. Cao, G. Gong, J. Pan, Fuzzing: Randomness? Reasoning! efficient directed fuzzing via large language models, 2025, arXiv:2507.22065. URL https://arxiv.org/abs/2507.22065.

[50] N. Metropolis, S. Ulam, The Monte Carlo method, J. Amer. Statist. Assoc. 44 (247) (1949) 335–341, http://dx.doi.org/10.1080/01621459.1949.10483310.

[51] J.M. Hammersley, D.C. Handscomb, Monte Carlo Methods, Methuen & Co., London, 1964.

[52] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[53] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification, in: Proceedings of the IEEE International Conference on Computer Vision, ICCV, 2015, pp. 1026–1034, http://dx.doi.org/10.1109/ICCV.2015.123.

[54] A.K. Lenstra, J.P. Hughes, M. Augier, J.W. Bos, T. Kleinjung, C. Wachter, Ron was Wrong, Whit is Right, Tech. Rep. 2012/064, Cryptology ePrint Archive, 2012, URL https://eprint.iacr.org/2012/064.

[55] N. Heninger, Z. Durumeric, E. Wustrow, J.A. Halderman, Mining your Ps and Qs: Detection of widespread weak keys in network devices, in: Proceedings of the 21st USENIX Security Symposium, USENIX Association, 2012, pp. 205–220.

[56] V.J. Manès, S. Kim, S.K. Cha, Ankou: Guiding grey-box fuzzing towards combinatorial difference, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering, ICSE, 2020, pp. 1024–1036.

[57] Y. Chen, T. Su, C. Sun, Z. Su, J. Zhao, Coverage-directed differential testing of JVM implementations, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 85–99, http://dx.doi.org/10.1145/2908080.2908095.

[58] X. Zhu, X. Feng, X. Meng, S. Wen, S. Camtepe, Y. Xiang, K. Ren, CSI-fuzz: Full-speed edge tracing using coverage sensitive instrumentation, IEEE Trans. Dependable Secur. Comput. 19 (2) (2022) 912–923, http://dx.doi.org/10.1109/TDSC.2020.3008826.

[59] F. Marini, B. Walczak, Particle swarm optimization (PSO). A tutorial, Chemometr. Intell. Lab. Syst. 149 (2015) 153–165, http://dx.doi.org/10.1016/j.chemolab.2015.08.020, URL https://www.sciencedirect.com/science/article/pii/S0169743915002117.

[60] A. Rebert, S.K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, D. Brumley, Optimizing seed selection for fuzzing, in: Proceedings of the 23rd USENIX Conference on Security Symposium, 2014, pp. 861–875.

[61] P. Chen, H. Chen, Angora: Efficient fuzzing by principled search, in: 2018 IEEE Symposium on Security and Privacy, SP, 2018, pp. 711–725, http://dx.doi.org/10.1109/SP.2018.00046.

[62] P. Chen, J. Liu, H. Chen, Matryoshka: Fuzzing deeply nested branches, in: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 499–513, http://dx.doi.org/10.1145/3319535.3363225.

[63] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, T. Holz, REDQUEEN: Fuzzing with input-to-state correspondence, in: Proceedings 2019 Network and Distributed System Security Symposium, 2019, URL https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/.

[64] V. Ganesh, T. Leek, M. Rinard, Taint-based directed whitebox fuzzing, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, USA, 2009, pp. 474–484, http://dx.doi.org/10.1109/ICSE.2009.5070546.

[65] X. Liu, X. Li, R. Prajapati, D. Wu, DeepFuzz: automatic generation of syntax valid c programs for fuzz testing, in: Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, in: AAAI'19/IAAI'19/EAAI'19, AAAI Press, 2019, http://dx.doi.org/10.1609/aaai.v33i01.33011044.

[66] J. Wang, B. Chen, L. Wei, Y. Liu, Skyfire: Data-driven seed generation for fuzzing, in: 2017 IEEE Symposium on Security and Privacy, SP, 2017, pp. 579–594, http://dx.doi.org/10.1109/SP.2017.23.

[67] K. Dewey, J. Roesch, B. Hardekopf, Fuzzing the rust typechecker using CLP (T), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2015, pp. 482–493, http://dx.doi.org/10.1109/ASE.2015.65.

[68] K. Dewey, J. Roesch, B. Hardekopf, Language fuzzing using constraint logic programming, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 725–730, http://dx.doi.org/10.1145/2642937.2642963.

[69] P. Saxena, S. Hanna, P. Poosankam, D. Song, FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications, in: Proceedings of the NDSS Symposium 2010, 2010, Document Type: Reports. URL https://www.ndss-symposium.org/ndss2010/flax-systematic-discovery-client-side-validation-vulnerabilities-rich-web-applications/.

[70] I. Yun, S. Lee, M. Xu, Y. Jang, T. Kim, QSYM : A practical concolic execution engine tailored for hybrid fuzzing, in: 27th USENIX Security Symposium, USENIX Security 18, USENIX Association, Baltimore, MD, 2018, pp. 745–761, URL https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

[71] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, X. Zhou, EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit, in: 29th USENIX Security Symposium, USENIX Security 20, USENIX Association, 2020, pp. 2307–2324, URL https://www.usenix.org/conference/usenixsecurity20/presentation/yue.

[72] J. Kim, J. Yu, Y. Lee, D.D. Kim, J. Yun, HD-FUZZ: Hardware dependency-aware firmware fuzzing via hybrid MMIO modeling, J. Netw. Comput. Appl. 224 (2024) 103835, http://dx.doi.org/10.1016/j.jnca.2024.103835, URL https://www.sciencedirect.com/science/article/pii/S1084804524000122.

[73] V.-T. Pham, M. Böhme, A. Roychoudhury, AFLNET: A greybox fuzzer for network protocols, in: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification, ICST, 2020, pp. 460–465, http://dx.doi.org/10.1109/ICST46399.2020.00062.

[74] F.B. Schneider, Implementing fault-tolerant services using the state machine approach: a tutorial, ACM Comput. Surv. 22 (4) (1990) 299–319, http://dx.doi.org/10.1145/98163.98167.

[75] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, J. Somorovsky, Analysis of DTLS implementations using protocol state fuzzing, in: 29th USENIX Security Symposium, USENIX Security 20, USENIX Association, 2020, pp. 2523–2540, URL https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean.

[76] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, K. Rieck, Pulsar: Stateful black-box fuzzing of proprietary network protocols, in: B. Thuraisingham, X. Wang, V. Yegneswaran (Eds.), Security and Privacy in Communication Networks, Springer International Publishing, Cham, 2015, pp. 330–347.

[77] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, G. Vigna, SNOOZE: toward a stateful network protocol fuzZEr, in: Proceedings of the 9th International Conference on Information Security, ISC '06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 343–358, http://dx.doi.org/10.1007/11836810_25.

[78] C. Pacheco, S.K. Lahiri, M.D. Ernst, T. Ball, Feedback-directed random test generation, in: 29th International Conference on Software Engineering, ICSE'07, 2007, pp. 75–84, http://dx.doi.org/10.1109/ICSE.2007.37.

[79] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, A. Tiu, Steelix: program-state based binary fuzzing, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 627–637, http://dx.doi.org/10.1145/3106237.3106295.

[80] P. Ekdahl, T. Johansson, SNOW-a new stream cipher, in: Proceedings of First Open NESSIE Workshop, KU-Leuven, 2000, pp. 167–168.

[81] R. Kaksonen, A Functional Method for Assessing Protocol Implementation Security, VTT publications, vol. 448, Technical Research Centre of Finland, 2001.

[82] A. Shah, D. She, S. Sadhu, K. Singal, P. Coffman, S. Jana, MC2: Rigorous and efficient directed greybox fuzzing, in: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 2595–2609, http://dx.doi.org/10.1145/3548606.3560648.

[83] A. Contributors, AFL++ include directory, 2024, URL https://github.com/AFLplusplus/AFLplusplus/tree/stable/include. (Accessed 04 December 2024).

[84] A. Contributors, AFL++ source directory, 2024, URL https://github.com/AFLplusplus/AFLplusplus/tree/stable/src. (Accessed 04 December 2024).

[85] B. Bloessl, C. Sommer, F. Dressler, D. Eckhoff, The scrambler attack: A robust physical layer attack on location privacy in vehicular networks, in: 2015 International Conference on Computing, Networking and Communications, ICNC, 2015, pp. 395–400, http://dx.doi.org/10.1109/ICCNC.2015.7069376.

[86] I. Pekaric, C. Sauerwein, S. Haselwanter, M. Felderer, A taxonomy of attack mechanisms in the automotive domain, Comput. Stand. Interfaces 78 (2021) 103539, http://dx.doi.org/10.1016/j.csi.2021.103539, URL https://www.sciencedirect.com/science/article/pii/S0920548921000349.

[87] G. Marsaglia, Xorshift RNGs, J. Stat. Softw. 8 (14) (2003) 1–6, http://dx.doi.org/10.18637/jss.v008.i14, URL https://www.jstatsoft.org/index.php/jss/article/view/v008i14.

[88] J. Wang, Y. Duan, W. Song, H. Yin, C. Song, Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing, in: 22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, USENIX Association, Chaoyang District, Beijing, 2019, pp. 1–15, URL https://www.usenix.org/conference/raid2019/presentation/wang.

[89] Y. Zhao, S. Wang, J. Wang, X. Hu, X. Xia, Ensemble fuzzing with dynamic resource scheduling and multidimensional seed evaluation, 2025, arXiv:2507.22442. URL https://arxiv.org/abs/2507.22442.

[90] C. Poncelet, K. Sagonas, N. Tsiftes, So many fuzzers, so little time⋆: Experience from evaluating fuzzers on the Contiki-NG network (hay)stack, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22, Association for Computing Machinery, New York, NY, USA, 2023, http://dx.doi.org/10.1145/3551349.3556946.

[91] C. Sun, S. Chandra, Z. Zheng, Y. Jiang, Engineering a better fuzzer with synergically integrated optimizations, in: IEEE International Symposium on Software Reliability Engineering, ISSRE, IEEE, 2019, URL https://cs.uwaterloo.ca/~cnsun/public/publication/issre19/issre19.pdf.

[92] T. Zhang, Y. Wang, Z. Li, L. Cavallaro, On understanding collaborative fuzzing, in: Proceedings of the 31st ACM Conference on Computer and Communications Security, CCS, ACM, 2024, URL https://s3.eurecom.fr/docs/ccs24_zhang.pdf.

[93] Y. Jiang, R. Chen, X. Wang, On the analysis of coverage feedback in a fuzzing proprietary system, Appl. Sci. 14 (13) (2024) 5939, http://dx.doi.org/10.3390/app14135939, URL https://www.mdpi.com/2076-3417/14/13/5939.