Reality Check: Independent Evaluation of Modern Grey-Box Fuzzing Techniques

Pavel Frolikov
University of California, Irvine
Irvine, California, USA
pavel@uci.edu

Abstract

Coverage-guided grey-box fuzzing has emerged as a practical and widely-adopted technique for automated software testing. The topic has been quite popular, with numerous new techniques, each claiming significant improvements over the established baselines.

This paper presents an independent evaluation of six state-of-the-art grey-box fuzzers across seven widely-used benchmarks. Our evaluation shows that AFL++, LibAFL, and Predictive Context-Sensitive Fuzzing achieve significant improvements over AFL (97%, 97%, and 95% vs 93% mean performance), while Darwin and MOPT show no meaningful improvement over AFL (92% each), and EcoFuzz consistently underperforms at 88%. We find that improvements are highly target-dependent, with complex targets showing the largest performance variations (62-99% across fuzzers) while simpler targets show convergence.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Fuzzing, Grey-Box Fuzzing, Coverage-Guided Fuzzing, Empirical Evaluation, Benchmarking, Software Testing

ACM Reference Format:
Pavel Frolikov. 2026. Reality Check: Independent Evaluation of Modern Grey-Box Fuzzing Techniques. In *Proceedings of the 2026 ACM Secure Development Conference (SecDev '26), July 5–6, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3805773.3806004

1 Introduction

Fuzzing has become essential for vulnerability discovery, with American Fuzzy Lop [38] (AFL) serving as the de facto standard since 2013. AFL's demonstrated efficacy sparked significant research interest, which led to a wave of next-generation fuzzing techniques that promise improvements through machine learning [19], energy efficiency [11, 37], context-sensitivity [7], and enhanced mutation strategies [14, 23].

Prior work highlights the importance of independent validation for fuzzing benchmarks [5, 29]. Motivated by curiosity about the landscape of modern fuzzing, this paper seeks to answer the following question: *Have modern grey-box fuzzers made measurable improvements over the baseline of AFL?* To this end, we evaluate six modern grey-box fuzzers against the AFL baseline. These include AFL++ [14], Predictive Context-Sensitive Fuzzing (PCSF) [7], LibAFL [15], Darwin [19], MOPT [23], and EcoFuzz [37].

Our evaluation demonstrates that recent fuzzing advances show mixed success. While AFL++, LibAFL, and PCSF achieve measurable improvements over AFL's 93% baseline, reaching 97%, 97%, and 95% respectively, the remaining three fuzzers fail to deliver. Darwin and MOPT perform on par with AFL at 92% each. EcoFuzz performs significantly worse at 88%, a finding corroborated by recent independent work [29]. We also observe that fuzzer performance is highly target-dependent, with complex targets showing 37-point performance spreads while simpler targets show near-complete convergence across all fuzzers.

2 Background

Fuzzing is an automated software testing technique that systematically generates inputs to explore as many program execution paths as possible, with the goal of triggering crashes, hangs, or other abnormal conditions that may indicate the presence of bugs. The infamous 2014 HeartBleed vulnerability [10] in OpenSSL [26] occurred due to a simple bounds check error that allowed private memory exfiltration from secure servers. HeartBleed exemplifies the class of vulnerabilities that fuzzing is well-suited to detect. Since then, industry gradually built large-scale continuous fuzzing infrastructure [17], while academic research focused on advancing the state-of-the-art.

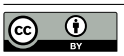
2.1 Fuzzing Strategies

Fuzzers vary in terms of how much visibility they have into the internals of the program being tested.

Black-box fuzzing: As the name suggests, such techniques [6, 13, 21, 39] treat a target as a black box; inputs are generated without any knowledge of the program's internal structure, and only external behavior – such as crashes or outputs – is observed.

White-box fuzzing: This approach [8, 16, 36] has full access to the program's code to guide test generation. Traditional white-box fuzzers rely on symbolic execution and constraint solving to explore program paths that would be nearly impossible to reach through random mutation (e.g. flipping a bit, inserting a token, or rearranging bytes) or feedback (e.g., coverage, or execution time) alone, though recent work has explored using large language models for white-box compiler fuzzing [36]. While white-box approaches can reach deeply nested or rare code paths that other methods typically miss, they come with high computational overhead and poor scalability.

Grey-box fuzzing: This approach is a hybrid of white- and black-box fuzzing. It employs lightweight instrumentation to track code



This work is licensed under a Creative Commons Attribution 4.0 International License.
SecDev '26, Montreal, QC, Canada
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2602-6/2026/07
https://doi.org/10.1145/3805773.3806004

coverage and uses that feedback to guide mutation-based input generation [1, 3, 7, 11, 14, 15, 19, 23, 34, 35, 37, 38]. This allows grey-box fuzzers to efficiently explore new execution paths without the computational overhead that makes white-box fuzzing impractical at scale. Coverage-guided grey-box fuzzing strikes a practical balance between effectiveness and efficiency, and it has become the dominant model in both academic research and large-scale industrial deployments [14, 17]. All fuzzers evaluated in this paper fall into this grey-box category.

2.2 American Fuzzy Lop (AFL)

While grey-box fuzzing has a long history [9, 25], AFL [38] brought coverage-guided fuzzing into widespread use. It combined lightweight instrumentation to track control-flow edge coverage with a mutation engine that prioritized inputs that trigger new program behavior. This mutation engine combines deterministic strategies (such as flips) with chaotic random mutations designed to explore the input space and escape coverage plateaus.

3 Methodology

We evaluate the performance of AFL (used as a baseline) and six other state-of-the-art fuzzers across seven commonly used open-source targets. They cover a wide complexity spectrum, from simple compression utilities (`zlib`, 470 edges) to `sqlite3`'s database engine with 20,360 edges, as well as parsers for structured formats such as XML (`libxml2`), XSLT (`libxslt`), and image processing libraries (`stb`, `libpng`). Table 1 lists target programs and their descriptions.

3.1 Tools Tested

Fuzzers were selected based on recency, publication at top security venues, open-source availability, and technique diversity. We focus exclusively on general-purpose coverage-guided grey-box fuzzers, excluding domain-specific approaches such as firmware, protocol, and compiler fuzzing.

3.1.1 Enhanced Mutation Strategies. **AFL++ [14]** is a current and actively maintained fork of AFL that builds upon AFL's architecture by addressing some of its shortcomings. While it preserves the core design of AFL (coverage-guided grey-box fuzzing using lightweight instrumentation and forkserver-based execution), AFL++ offers improvements that enhance performance and extensibility:

- It extends the AFL mutation engine with new operators, including multi-byte arithmetic, dictionary-based token substitution, and block overwrites.
- It supports deeper mutation stacking, allowing more mutations to be chained together per input than AFL.
- It provides a Custom Mutator API that allows researchers to implement customized mutation strategies without modifying the fuzzer's core. Several fuzzers have been ported into AFL++ using this modular interface, e.g., RedQueen [12], AFLSmart [27], and Predictive Context-Sensitive Fuzzing [7].

MOPT [23] improves fuzzing efficiency by dynamically adjusting how often different mutations are applied. Unlike AFL's fixed mutation probabilities, MOPT uses a lightweight form of Particle Swarm Optimization [20] to continuously adjust mutation likelihoods based on their past success. By adapting to the target program's behavior, MOPT automatically prioritizes effective mutations, leading to faster coverage growth without manual tuning.

3.1.2 Architectural Redesign. **LibAFL [15]** is a fuzzing framework designed for performance, composability, and scalability. Unlike monolithic fuzzers, LibAFL is designed to be modular and adaptable. Every core component (scheduler, mutator, executor) can be swapped. Its baseline fuzzer improves upon AFL++ by incorporating execution time as a signal to prioritize deeper paths. Coded in Rust [30], LibAFL uses lightweight threads and in-process execution to achieve higher throughput than traditional process-based fuzzers.

3.1.3 Context-Sensitive Approaches. **Predictive Context Sensitive Fuzzing [7]** (PCSF) is implemented as a custom mutator for AFL++ that introduces a lightweight form of context sensitivity into the fuzzing loop. PCSF distinguishes between calls to the same function based on their call sites, creating separate execution paths for each context. This helps the fuzzer explore behaviors that would otherwise look the same. For example:

Predictive Context Sensitive Fuzzing

```
def parse_header(data):
  if data.startswith(b"\x89PNG"):
    return parse_img(data)
  if data.startswith(b"\x45\x00"):
    return parse_network(data)
  return False

def load_image_file(f):
    parse_header(f.read(1024))

def handle_network_packet(pkt):
    parse_header(pkt)
```

Even though `parse_header()` is the same function, it is used in two contexts: `load_image_file()` → `parse_header()` and `handle_network_packet()` → `parse_header()`. Without context sensitivity, a fuzzer treats both call paths as the same coverage, potentially missing branches that require different input formats for each context. PCSF clones `parse_header()` for each call site, creating distinct execution paths that the fuzzer can explore independently.

3.1.4 Adaptive Learning. **Darwin [19]** is a mutation scheduling strategy built atop AFL that adapts over time by learning the most effective mutations. It uses an evolution strategy to adjust the probability of applying each mutation based on its past success. Mutations that lead to new coverage are reinforced, allowing the fuzzer to gradually prioritize those that work best without requiring any manual tuning. Darwin continuously self-adjusts based on runtime feedback.

3.1.5 Energy Optimization. **EcoFuzz [37]**, built upon AFL, aims to maximize path exploration while reducing wasted computation on redundant inputs. Fuzzing naturally plateaus over time because later stages exert more effort on already-covered paths rather than discovering new ones. EcoFuzz uses a reward-based search strategy [2] to estimate which inputs are most likely to lead to new paths, and applies an adaptive energy heuristic to decide how many mutations to generate for each input.

Table 1: Target programs used in evaluation.

Program	Target	Description	Max Edges
`sqlite3` [18]	sqlite3	SQLite database engine	20,360
`libxml2` [31]	libxml2_xml	XML parsing	15,730
`libxslt` [32]	libxslt_xpath	XSLT transformation	11,361
`openssl` [26]	openssl_x509	X.509 certificate parsing	5,834
`stb` [4]	stb_stbi_read	Image loading	2,198
`libpng` [28]	libpng_libpng_read	PNG image parsing	2,076
`zlib` [22]	zlib_zlib_uncompress	Data compression	470

3.2 Experimental Setup

All experiments were conducted on standardized Ubuntu 22.04.5 LTS virtual machines with 64 GB RAM and 16 CPU cores (13th Gen Intel i5-13600KF). Target binaries were compiled with fuzzer-specific instrumentation using their respective compilation toolchains (afl-clang-fast for AFL and AFL++, libafl_cc for LibAFL, etc.), with consistent -O2 optimization levels and identical compilation flags where possible. Fuzzers were configured using default parameters recommended by their respective authors, and used the same initial seed corpus for each target program to eliminate potential bias from different starting conditions.

3.3 Evaluation Protocol

It might seem that the best way to evaluate a fuzzer is by the number of bugs or crashes it discovers. However, code coverage is the most widely accepted and practical metric in current fuzzing research, serving as a proxy for a fuzzer's ability to find complex and rare program behaviors.

However, coverage-based evaluation has certain limitations. As Wang et al. [33] demonstrate, coverage-guided fuzzers may overlook security-critical program states that do not contribute to new coverage. Similarly, PCSF shows that traditional coverage metrics can miss important buggy states without proper context sensitivity [7].

Following Böhme et al. [5], we use coverage as the primary metric, since bug-based comparisons require prohibitively long campaigns to produce statistically meaningful results. We adopt recommendations in [5] for fuzzer evaluation, deviating only in trial count and duration, as discussed below. For each fuzzer-target pair, we performed three campaigns, each lasting five hours, totaling 735 individual fuzzing hours: 7 fuzzers × 7 targets × 3 campaigns × 5 hours.

Although [5] recommends 10-20 campaigns of 12-24 hours, the 5-hour duration aligns with industry practice (Google's OSS-Fuzz runs 5-hour jobs) and is well-justified by prior work. Fuzzers typically reach coverage plateaus within the first few hours of execution. Darwin [19] observes that coverage outcomes tend to emerge within the first 200 minutes. In cases where Darwin was not ahead by that point, it did not outperform other fuzzers later. Their standalone evaluations ran for 24 hours, and comparative experiments with other fuzzers were limited to 6-hour runs. Coverage was measured as the number of program branches reached during execution and collected at 15-minute intervals throughout each campaign using FuzzBench [24].

4 Results

4.1 Overall Performance Rankings

Our evaluation summarized in Table 2, reveals a clear performance hierarchy among tested fuzzers. Green indicates the top-performing fuzzer for each benchmark, blue indicates second place, and grey indicates third place. AFL++ emerges as the top performer scoring a 98.00% median and 97.29% mean. It is followed closely by LibAFL (97.00% and 97.14%, respectively) and PCSF (98.00% and 94.86%). These three significantly outperform the baseline AFL, which scores only 93.00% and 92.71%.

Results demonstrate that only three state-of-the-art fuzzers offer meaningful improvements over AFL. The adaptive learning approaches fall short of their promises: Darwin (93.00% median and 92.43% mean) and MOPT (93.00% median and 92.00% mean) show minimal improvement over AFL. EcoFuzz stands out as a clear under-performer with its 87.86% mean.

4.2 Performance Variation Across Targets

4.2.1 High-Variance Targets. `sqlite3` has the largest performance gaps among all targets, as shown in Table 2. It has scores ranging from 62% (EcoFuzz) to 99% (LibAFL) and 94% in the case of AFL++. `sqlite3` demonstrates LibAFL's architectural advantages, as shown in Figure 1, where its in-process execution model enables rapid early coverage growth, reaching 18,000 branches within 15 minutes compared to other fuzzers in the 11,000 range. AFL++ does eventually narrow the gap considerably.

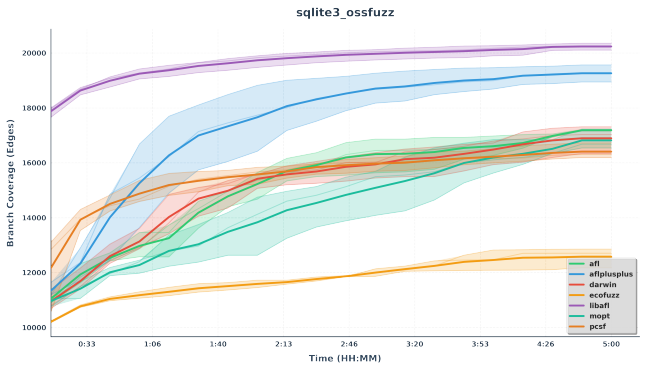


Figure 1: Coverage over time on *sqlite3*.

`libxslt_xpath` exposes how fuzzer design choices affect performance on structured input formats, as illustrated in Figure 2. AFL++ and PCSF both reach 99%, while EcoFuzz trails at 81%. AFL++'s performance stems from its enhanced mutation operators that can better handle XSLT's structured input format, while PCSF's function cloning allows the same parsing functions to be explored in different calling contexts. Conversely, EcoFuzz's energy-conserving approach may prematurely abandon exploration of parsing paths that appear redundant but represent distinct program behaviors.

4.2.2 Low-Variance Targets. Both `openssl_x509` and `zlib_zlib_uncompress` demonstrate fuzzer convergence, with all achieving similar coverage levels (highlighted in yellow in Table 2). These

Table 2: Normalized coverage scores for each fuzzer across all benchmarks. NOTE: The table shows median relative performance of each fuzzer to the maximum of the experiment. The highest relative performance may not be 100.

Benchmark	AFL++	LibAFL	PCSF	AFL	Darwin	MOPT	EcoFuzz
Median	98.00	97.00	98.00	93.00	93.00	93.00	93.00
Mean	97.29	97.14	94.86	92.71	92.43	92.00	87.86
libpng_libpng_read	96.00	96.00	97.00	93.00	93.00	93.00	93.00
libxml2_xml	98.00	98.00	99.00	95.00	96.00	96.00	96.00
libxslt_xpath	99.00	96.00	99.00	93.00	91.00	90.00	81.00
openssl_x509	99.00	99.00	99.00	99.00	99.00	99.00	99.00
sqlite3	94.00	99.00	80.00	84.00	83.00	81.00	62.00
stb_stbi_read	96.00	95.00	92.00	89.00	89.00	88.00	88.00
zlib_zlib_uncompress	99.00	97.00	98.00	96.00	96.00	97.00	96.00

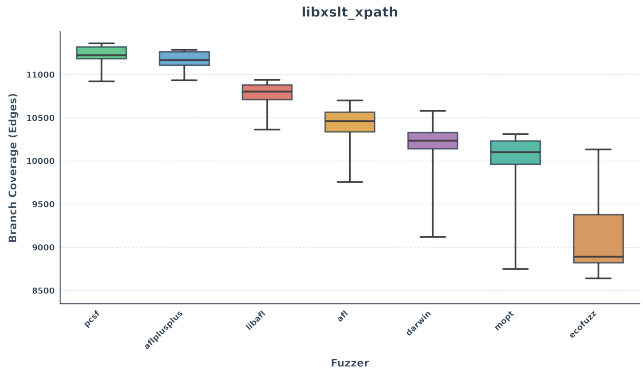


Figure 2: Final coverage on *libxslt_xpath*.

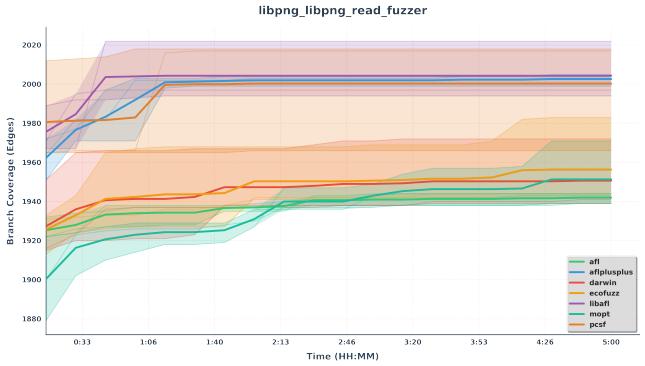


Figure 3: Coverage over time on *libpng_libpng_read*.

targets primarily follow linear processing pipelines with early rejection of invalid inputs. They lack the complex branching and varied execution paths found in larger, more general-purpose programs. In contrast to the high-variance targets in Section 4.2.1, these targets offer limited opportunities for fuzzers to differentiate themselves.

5 Takeaways

5.1 Fuzzers That Outperformed AFL

AFL++ consistently delivers on its promises, achieving the highest mean performance (97.29%) through enhanced mutations over AFL. LibAFL's architectural redesign provides clear but target dependent advantages, with its in-process execution model beating out AFL++ in some cases (such as sqlite3). PCSF demonstrated efficacy on structured parsing targets (`libxml2_xml`: 99.00%, `libxslt_xpath`: 99.00%) where context-sensitivity provides clear benefits. As shown in Figure 3, these performance differences typically emerge within the first hour of fuzzing and remain stable throughout longer runs.

5.2 Minimal Gains Over Baseline

Both Darwin and MOPT delivered results on par with the AFL baseline. Despite theoretical promises of adaptive optimization, these approaches showed minimal practical improvements across all target types. This aligns with Darwin's paper, which reported only a 1.6% increase in unique paths.

EcoFuzz achieved only an 87.86% mean performance, worse than the decade-old AFL. The energy conservation approach proved particularly detrimental on complex targets, like sqlite3 as shown in Figure 1, indicating that attempts to reduce "redundant" exploration may limit beneficial coverage paths. These findings align with Schloegel et al. [29], who independently found that EcoFuzz achieves lower code coverage than AFL on standard coverage metrics.

5.3 Recommendations

For general-purpose fuzzing, AFL++ serves as a reliable starting point, consistently delivering strong performance across diverse targets. When fuzzing high-value targets, practitioners should invest time in characterizing both program structure and behavior to select the most appropriate fuzzer.

6 Conclusion

This paper presented an independent evaluation of six modern grey-box fuzzers across seven real-world targets. Our results show

that while PCSF, AFL++, and LibAFL deliver measurable improvements over AFL, the remaining three fuzzers fail to outperform the decade-old baseline. Darwin and MOPT showed no meaningful improvement, while EcoFuzz actively underperformed.

Acknowledgments

We thank SecDev 2026 reviewers for constructive feedback. We would also like to thank Sergey Frolikov, Christoph Sendner, Gene Tsudik, and Elizabeth Twomey, for their many helpful comments and suggestions on earlier drafts of this work.

References

[1] Ammar Askar, Fabian Fleischer, Christopher Kruegel, Giovanni Vigna, and Taesoo Kim. 2025. MALintent: Coverage Guided Intent Fuzzing Framework for Android. In *Proc. Network and Distributed Systems Security*. doi:10.14722/ndss.2025.230125

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* Volume 47 (2002). doi:10.1023/A:1013689704352

[3] Andrew Bao, Wenjia Zhao, Yanhao Wang, Yueqiang Cheng, Stephen McCamant, and Pen-Chung Yew. 2025. From alarms to real bugs: multi-target multi-step directed greybox fuzzing for static analysis result verification. In *Proc. USENIX Security Symposium*.

[4] Sean Barrett. 2025. stb - single-file public domain libraries for C/C++. https://github.com/nothings/stb. Accessed: 2025-05-21.

[5] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2002. On the reliability of coverage-based fuzzer benchmarking. In *Proc. International Conference on Software Engineering*. doi:10.1145/3510003.3510230

[6] Anne Borcherding, Mark Giraud, and Johannes Häring. 2025. Bringing Light into the Darkness: Leveraging Hidden Markov Models for Blackbox Fuzzing. In *Proc. International Conference on Automation of Software Test*. doi:10.1109/AST66626.2025.00021

[7] Pietro Borrello, Andrea Fioraldi, Daniele Cono D'Elia, Davide Balzarotti, Leonardo Querzoni, and Cristiano Giuffrida. 2024. Predictive Context-sensitive Fuzzing. In *Proc. Network and Distributed Systems Security*. doi:10.14722/ndss.2024.24113

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. Conference on Operating Systems Design and Implementation*.

[9] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: Automatically generating inputs of death. Volume 12, Number 2 (2008). doi:10.1145/1455518.1455522

[10] Codenomicon and OpenSSL Team. 2014. The Heartbleed Bug. https://heartbleed.com/. CVE-2014-0160.

[11] Ayse Irmak Ercevik, Aidan Dakhama, Melane Navaratnarajah, Yazhuo Cao, and Leo Fernandes. 2025. Fuzz Smarter, Not Harder: Towards Greener Fuzzing with GreenAFL. In *Proc. Symposium on Search-Based Software Engineering*.

[12] Andreas Falkenberg, Manuel Egele, Mathias Payer, and Thorsten Holz. 2019. RedQueen: Fuzzing with Input-to-State Correspondence. In *Proc. Network and Distributed System Security Symposium*. doi:10.14722/ndss.2019.23371

[13] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. In *Proc. ACM Conference on Computer and Communications Security*. doi:10.1145/3460120.3484543

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *Proc. USENIX Workshop on Offensive Technologies*.

[15] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proc. ACM Conference on Computer and Communications Security*. doi:10.1145/3548606.3560602

[16] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. Network and Distributed System Security Symposium*.

[17] Google. 2025. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Accessed: 2025-05-21.

[18] D. Richard Hipp. 2024. SQLite - A self-contained, high-reliability, embedded, full-featured, public-domain SQL database engine. https://github.com/sqlite/sqlite. Accessed: 2025-05-21.

[19] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. 2023. DARWIN: Survival of the Fittest Fuzzing Mutators. In *Proc. Network and Distributed System Security Symposium*. doi:10.14722/ndss.2023.23159

[20] J. Kennedy and R. Eberhart. 1995. Particle swarm optimization. In *Proc. International Conference on Neural Networks*.

[21] Kyungtae Kim, Sungwoo Kim, Kevin R. B. Butler, Antonio Bianchi, Rick Kennell, and Dave (Jing) Tian. 2023. Fuzz The Power: Dual-role State Guided Black-box Fuzzing for USB Power Delivery. In *Proc. USENIX Security Symposium*.

[22] Jean loup Gailly and Mark Adler. 2025. zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://github.com/madler/zlib. Accessed: 2025-05-21.

[23] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proc. USENIX Security Symposium*.

[24] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proc. ACM joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. doi:10.1145/3468264.3473932

[25] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* Volume 33, Number 12 (1990). doi:10.1145/96267.96279

[26] OpenSSL Project. 1998. OpenSSL: The Open Source toolkit for SSL/TLS. https://www.openssl.org/. Version as cited or used; check https://www.openssl.org/source/ for specific releases.

[27] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru-Razvan Căciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* Volume 47, Number 8 (2021). doi:10.1109/TSE.2019.2941681

[28] Glenn Randers-Pehrson. 2025. libpng - the official PNG reference library. https://github.com/glennrp/libpng. Accessed: 2025-05-21.

[29] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing . In *Proc. IEEE Symposium on Security and Privacy*.

[30] The Rust Project Developers. 2023. The Rust Programming Language. https://www.rust-lang.org.

[31] Daniel Veillard. 2025. libxml2 - the XML C parser and toolkit of Gnome. https://gitlab.gnome.org/GNOME/libxml2. Accessed: 2025-05-21.

[32] Daniel Veillard. 2025. libxslt - the XSLT C library developed for the GNOME project. https://gitlab.gnome.org/GNOME/libxslt. Accessed: 2025-05-21.

[33] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Proc. Network and Distributed System Security Symposium*. doi:10.14722/ndss.2020.24422

[34] Haoyu Xiao, Ziqi Wei, Jiarun Dai, Bowen Li, Yuan Zhang, and Min Yang. 2025. HouseFuzz: Service-Aware Grey-Box Fuzzing for Vulnerability Detection in Linux-Based Firmware. In *Proc. IEEE Symposium on Security and Privacy*. doi:10.1109/SP61157.2025.00213

[35] Jifan Xiao, Peng Jiang, Zixi Zhao, Ruizhe Huang, Junlin Liu, and Ding Li. 2025. Robust, efficient, and widely available greybox fuzzing for COTS binaries with system call pattern feedback. In *Proc. USENIX Conference on Security Symposium*.

[36] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. Whitefox: White-box compiler fuzzing empowered by large language models. In *Proc. ACM on Programming Languages*. doi:10.1145/3689736

[37] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *Proc. USENIX Security Symposium*.

[38] Michal Zalewski. 2013. American Fuzzy Lop (AFL). https://lcamtuf.coredump.cx/afl/technical_details.txt.

[39] Yonghao Zou, Jia-Ju Bai, Zu-Ming Jiang, Ming Zhao, and Diyu Zhou. 2025. Blackbox Fuzzing of Distributed Systems with Multi-Dimensional Inputs and Symmetry-Based Feedback Pruning. In *Proc. Network and Distributed System Security Symposium*. doi:10.14722/ndss.2025.241912

Received 2026-02-05; accepted 2026-03-19