The Impact of Magma: A Ground-Truth Fuzzing Benchmark

Ahmad Hazimeh*
EPFL; BugScale
ahmad@hazimeh.dev

Adrian Herrera†
Australian National University; Interrupt Labs
adrian.herrera02@gmail.com

Srividya Subramanian§
ETHZ; EPFL
srividya.ssa@gmail.com

Thaqiya Aman¶
Presidency University Bangalore; EPFL
thaqiyaaman@gmail.com

Sara Vaccino
EPFL
sara.vaccino@epfl.ch

Qiang Liu‡
EPFL
cyruscyliu@gmail.com

Mathias Payer
EPFL
mathias.payer@nebelwelt.net

***Abstract*—Magma is an open-source and ground-truth fuzzing benchmark that enables uniform fuzzer evaluation and comparison. Magma was originally released with a research paper published at ACM SIGMETRICS 2021. This short paper explains the motivation, the design, and the impact of Magma, with a description of extensions to the original benchmark.**

***Index Terms*—fuzzing benchmark, bug reaching, bug triggering**

1. Introduction

Fuzzing is a widely-used dynamic bug discovery technique that feeds randomly generated inputs to a target program to trigger bugs, which is successful in finding bugs in open-source [1] and commercial off-the-shelf [2, 55, 3] software. This success has resulted in an explosion of new techniques claiming to improve bug-finding performance [44]. To highlight improvements, these techniques are typically evaluated across a range of metrics, including: (i) crash counts; and/or (ii) code-coverage profiles. While these metrics provide some insight into a fuzzer's performance, they are insufficient for use in fuzzer comparisons.

The simplest fuzzer evaluation method is to count and compare the number of crashes triggered by fuzzers on the same target. Unfortunately, crash counts often inflate the number of actual bugs in the target [27]. Moreover, deduplication techniques (e.g., coverage profiles, stack hashes) are imprecise because they fail to accurately identify the root cause of these crashes [27, 8].

Code-coverage profiles are another performance metric commonly used to evaluate and compare fuzzing techniques. Intuitively, covering more code correlates with finding more bugs. However, previous work [27] has shown that there is a weak correlation between coverage-deduplicated crashes and ground-truth bugs, implying that higher coverage does not necessarily indicate better fuzzer effectiveness.

The deficiencies of existing performance metrics call for a rethinking of fuzzer evaluation practices. In particular, the performance metrics used in these evaluations must accurately measure a fuzzer's ability to achieve its main objective: *finding bugs*. Furthermore, the targets that are used to assess how well a fuzzer meets this objective must be realistic and exercise diverse behavior. This allows a practitioner to have confidence that a given fuzzing technique will yield improvements in *finding real-world bugs*.

To satisfy these criteria, we present *Magma*, a ground-truth fuzzer benchmark based on real programs with real bugs. Magma consists of *seven* widely-used open-source libraries and applications, totaling 2 MLOC (see Section 2.1). For each Magma program, we manually analyze security-relevant bug reports and patches, reinserting defective code back into these seven programs (in total, *118* bugs were analyzed and reinserted) (see Section 2.2). Additionally, each reinserted bug is accompanied by a light-weight *oracle* that detects and reports if the bug is *reached* (i.e., the bug code location is executed) or *triggered* (i.e., the bug condition is satisfied) (see Section 2.3). This distinction between reaching and triggering a bug—in addition to a fuzzer's ability to detect a triggered bug—presents a new opportunity to evaluate a fuzzer across multiple dimensions (again, focusing on ground-truth bugs). Magma is open-source and available at https://hexhive.epfl.ch/magma/.

2. The Magma Artifact

2.1. Target Selection

Magma contains seven targets, which we summarize in Table 1. In addition to these seven *targets* (i.e., the codebases into which bugs are injected), Magma also includes 25 *drivers* (i.e., executable programs that provide a command-line interface to the target) that exercise different functionality within the target. Inspired by Google OSS-Fuzz [1], these drivers are sourced from the original target codebases (as drivers are best developed by domain experts).

**Work completed prior to the author joining BugScale.*
†Work completed prior to the author joining Interrupt Labs.
§Work completed when the author was an exchange student at EPFL.
¶Work completed when the author was an intern at EPFL; currently at the University of Galway, Ireland.
‡Corresponding Author

TABLE 1: The targets, driver programs, bug counts, and evaluated features incorporated into Magma.

Target	Drivers	Version	File Type	Bugs	Magic Values	Recursive Parsing	Compression	Checksums	Global State
libpng	`read_fuzzer`, `readpng`	1.6.38	PNG	7	✓	✗	✓	✓	✗
libtiff	`read_rgba_fuzzer`, `tiffcp`	4.1.0	TIFF	14	✓	✗	✓	✗	✗
libxml2	`read_memory_fuzzer`, `xml_reader_for_file_fu` `xmllint`	2.9.10	XML	18	✓	✓	✗	✗	✗
poppler	`pdf_fuzzer`, `pdfimages`, `pdftoppm`	0.88.0	PDF	22	✓	✓	✓	✓	✗
openssl	`asn1`, `asn1parse`, `bignum`, `bndiv`, `client`, `cms`, `conf`, `crl`, `ct`, `server`, `x509`	3.0.0	*Binary blobs*	21	✓	✗	✓	✓	✓
sqlite3	`sqlite3_fuzz`	3.32.0	SQL queries	20	✓	✓	✗	✗	✓
php	`exif`, `json`, `parser`, `unserialize`	8.0.0–dev	*Various*	16	✓	✓	✗	✗	✗

Magma's seven targets were selected for their diversity in functionality (summarized in Table 1), allowing Magma to evaluate the code exploration capabilities of fuzzers. Inspired by benchmarks in other fields [7, 52, 25, 51], we apply *Principal Component Analysis* (PCA) to quantify this diversity. PCA is a statistical analysis technique that transforms an N-dimensional space into a lower-dimensional space while preserving variance as much as possible [48]. As shown in Figure 1, unsurprisingly, the four LAVA-M targets are tightly clustered over the first four principal components, since the LAVA-M targets are all sourced from coreutils and hence share the same codebase. In contrast, both the CGC and Magma provide a wide-variety of targets. For example, *openssl*—which contains a large amount of cryptographic and networking code—appears distinct from the main clusters in Figure 1.

2.2. Bug Selection and Insertion

Magma contains 118 bugs, spanning 11 CWEs (summarized in Figure 2). Compared to existing benchmarks, Magma has both the second-largest variety of bugs (by CWE) and the second-largest "bug density" (the ratio of the number of bugs to the number of targets) after the CGC and LAVA-M, respectively. While the CGC has a wider variety of bugs, its targets are not indicative of real-world software (in terms of both size and complexity). Similarly, while LAVA-M's bug density (566.25 bugs per target) is an order-of-magnitude larger than Magma's (16.86 bugs per target), LAVA-M is restricted to a single, synthetic bug type.

Importantly, Magma contains *real* bugs sourced from bug reports and *forward-ported* to the most recent version of the target codebase. This is in contrast to existing fuzzing benchmarks (e.g., BugBench, Google FTS) that rely on old, unpatched versions of the target codebase. Unfortunately, using older codebases limits the number of bugs available in each target. In comparison, forward-porting—which is synonymous to back-porting fixes from newer codebases to older, buggy releases—does not suffer from this issue, making Magma's targets easily extensible.

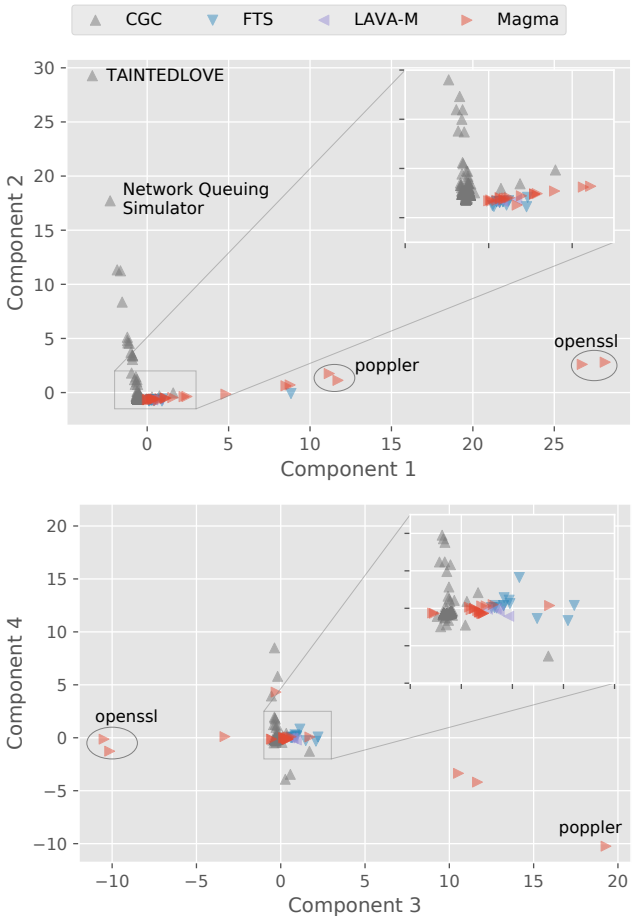

Figure 1: Scatter plots of benchmark scores over the first four principal components (60% variance in the benchmark targets). Each point represents a benchmark subject and closer points indicate lower target diversity.

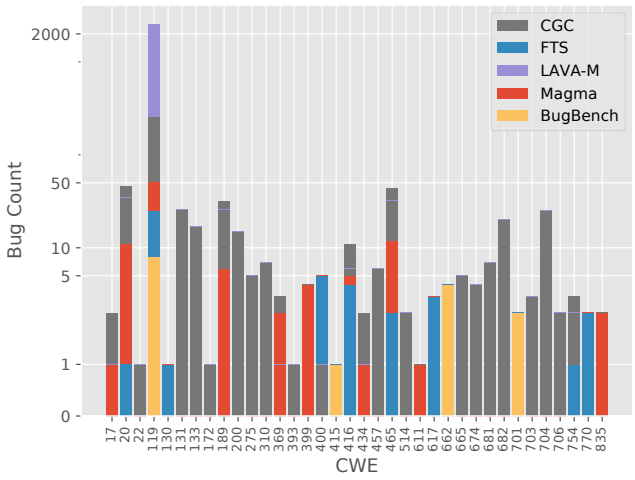

Figure 2: Comparison of benchmark bug classes.

Forward-porting begins with the identification—from the reported bug fix—of the code changes that must be reverted to reintroduce the bug. Bug-fix commits can contain multiple fixes to one or more bugs, so disambiguation is necessary to prevent the introduction of unintended bugs. Alternatively, bug fixes may be spread over multiple commits (e.g., if the original fix did not cover all edge cases). Following the identification of code changes, we identify what program state is involved in evaluating the trigger condition. If necessary, we introduce additional program variables to access that state. From this state, we determine a boolean expression that serves as a light-weight oracle for identifying a triggered bug. Finally, we identify a point in the program where we inject a canary before the bug can manifest faulty behavior. This canary helps measure our fuzzer performance metrics, which are discussed in the following section.

2.3. Performance Metrics

Magma uses three bug-centric performance metrics, i.e., *reaching*, *triggering*, and *detecting*, to evaluate fuzzers.

A *reached* bug refers to a bug whose oracle was called, implying that the executed path reaches the context of the bug, without necessarily triggering the fault. This is where coverage profiles fall short: simply covering the faulty code does not mean that the program is in the correct state to trigger the bug. Hence, a *triggered* bug refers to a bug that was reached, and *whose triggering condition was satisfied*, indicating that a fault occurred. Source-code instrumentation (i.e., the canary) provides ground-truth knowledge and runtime feedback of reached and triggered bugs. Each bug is approximated by (a) the lines of code patched in response to a bug report, and (b) a boolean expression representing the bug's trigger condition. The canary reports: (i) when the line of code is reached; and (ii) when the input satisfies the conditions for faulty behavior (i.e., triggers the bug).

When a bug is triggered, the oracle only indicates that the conditions for a fault have been satisfied, but this does not imply that the fault was encountered or detected by the fuzzer. Therefore, we also draw a distinction between *triggering* and *detecting* a bug. Whereas most security-critical bugs manifest as a low-level security policy violation for which state-of-the-art sanitizers are well-suited (e.g., memory corruption, data races, invalid arithmetic), other bug classes are not as easily observed. For example, resource exhaustion bugs are often detected long after the fault has manifested, either through a timeout or an out-of-memory error. Even more obscure are semantic bugs, whose malfunctions cannot be observed without a specification or reference. Consequently, various fuzzing techniques have been developed to target these bug classes (e.g., SlowFuzz [50] and NEZHA [49]). Such advancements in fuzzer techniques may benefit from an evaluation that includes the bug *detection* rate as another dimension for comparison.

2.4. Runtime Monitoring and Post-Processing

Magma provides a runtime monitor that collects real-time statistics from the instrumented target. This provides a mechanism for visualizing the fuzzer's progress and its evolution over time, without complicating the instrumentation.

The runtime monitor collects data about reached and triggered bugs. Because this data primarily relates to the fuzzer's program exploration capabilities, we post-process the monitor's output to study the fuzzer's fault detection capabilities. This is achieved by replaying the crashing inputs (produced by the fuzzer) against the benchmark canaries to determine which bugs were reached and/or triggered. Furthermore, crashing inputs were validated by replaying them through the ASAN-instrumented targets, enabling us to assess the fuzzer's bug detection capability.

3. Impact

Magma is actively used and maintained by the research community; until Sept 2025, it has been cited 286 times, forked 108 times, starred 319 times, and received 78 pull requests. Specifically, Magma has been actually used in peer-reviewed papers [20, 46, 39, 5, 59, 24, 61, 9, 29, 78, 32, 21, 28, 68, 35, 73, 11, 40, 15, 74, 47, 77, 19, 58, 42, 60, 65, 67, 63, 22, 34, 54, 23, 31, 37, 14, 57, 71, 12, 6, 66, 79, 72, 53, 64, 4, 80, 17, 70, 75, 10, 45, 76, 13, 69, 30, 26, 43, 36, 38, 33, 62, 56]. Next, we will explain how Magma serves as a fuzzing benchmark year by year, facilitating and continuously pushing the relevant research moving forward.

As shown in Table 2, since Magma's release in 2021, at least 45 high-quality papers have used Magma in their own research. In 2024 and 2025, the number of papers remained stable at 14. Notably, these papers were published in well-known security and software engineering conferences, and even system and programming language conferences, demonstrating Magma's broad impact.

Magma has been widely adopted to advance various aspects of fuzzing, including input generation, seed selection and mutation, mutation scheduling, coverage feedback, and bug sanitization. It has also been used in directed fuzzing

TABLE 2

Year	Venue	Paper	Task	Category
2021	ISSTA	[20]	Seed Selection	Fuzzing
2022	CCS	[59]	Directed Fuzzing	Fuzzing
2022	NDSS	[24]	Mutation Scheduling	Fuzzing
2022	ACSAC	[61]	Directed Fuzzing	Fuzzing
2022	ACSAC	[29]	Mutation Scheduling	Fuzzing
2022	ASIACCS	[9]	Directed Fuzzing	Fuzzing
2022	ISSTA	[46]	Coverage Feedback	Fuzzing
2023	ICSE	[32]	Mutation Scheduling	Fuzzing
2023	TOSEM	[21]	Coverage Feedback	Fuzzing
2023	SBFT	[28]	Fuzzing Deployment	Deployment
2023	ISSTA	[68]	Directed Fuzzing	Fuzzing
2023	OOPSLA	[35]	Coverage Feedback	Fuzzing
2023	NDSS	[73]	Byte Selection and Mutation	Fuzzing
2023	Security	[11]	Parallel Fuzzing	Acceleration
2023	CCS	[40]	Input Generation	Fuzzing
2023	CCS	[74]	Fuzzing Acceleration	Acceleration
2023	ASE	[47]	Fuzzing for Program Analysis	Others
2024	ASIACCS	[42]	Seed Selection	Fuzzing
2024	CCS	[60]	Seed Selection and Mutation	Fuzzing
2024	TOSEM	[65]	Seed Selection	Fuzzing
2024	TOSEM	[63]	Coverage Feedback	Fuzzing
2024	TOSEM	[54]	Seed Selection	Fuzzing
2024	S&P	[22]	Directed Fuzzing	Fuzzing
2024	S&P	[23]	Directed Fuzzing	Fuzzing
2024	Security	[34]	Directed Fuzzing	Fuzzing
2024	Security	[57]	Directed Fuzzing	Fuzzing
2024	ISSTA	[31]	Mutation Scheduling	Fuzzing
2024	ISSTA	[14]	Directed Fuzzing	Fuzzing
2024	ASPLOS	[37]	Bug Sanitization	Fuzzing
2024	ASPLOS	[71]	Program Transformation	Others
2024	TSE	[79]	Seed Selection and Mutation	Fuzzing
2025	TOSEM	[53]	Coverage Feedback	Fuzzing
2025	TOSEM	[36]	Coverage Feedback	Fuzzing
2025	Security	[72]	Input Generation	Fuzzing
2025	Security	[64]	Coverage Feedback	Fuzzing
2025	Security	[4]	Directed Fuzzing	Fuzzing
2025	EuroS&P	[17]	Directed Fuzzing	Fuzzing
2025	EuroS&P	[69]	Coverage Feedback	Fuzing
2025	ISSTA	[70]	Modular-Based Fuzzing	Implementation
2025	ISSTA	[76]	Parallel Fuzzing	Acceleration
2025	FSE	[75]	Mutation Scheduling	Fuzzing
2025	FSE	[26]	Crash Deduplication	Post-Fuzzing
2025	ICSE	[10]	Directed Fuzzing	Fuzzing
2025	ICSE	[30]	Fuzzing for Backdoor Detection	Others
2025	ICSE	[62]	Bug Sanitization	Fuzzing

and post-fuzzing analyses, such as crash deduplication. Moreover, researchers have leveraged Magma to evaluate the acceleration and practical deployment of fuzzing. Beyond fuzzing, Magma has been applied to broader tasks such as program analysis, program transformation, and backdoor detection. Collectively, these applications demonstrate Magma's impact across the relevant research community.

4. Magma v1.3

Magma v1.0 (2020) was initially designed with seven targets and 118 ground-truth bugs. It was soon updated to v1.2 (2021), which remained stable with nine targets and a total of 138 ground-truth bugs. However, keeping Magma up to date is essential to staying relevant. We found that 57 out of 138 bugs in the old version of Magma could no longer be applied due to code changes. We also searched public CVE databases and found 246 new CVEs affecting Magma's targets since Magma v1.2. Without such update, Magma would no longer work with the latest fuzzers or reflect the vulnerabilities that are being discovered recently.

In 2025, we update Magma from version v1.2 to v1.3. Our updates include: bringing targets and fuzzers to recent versions, fixing broken bug patches so they work with the new versions (11 bug patches going to the graveyard), and improving the infrastructure to make it easier to debug and validate fuzzing results. In addition to this, we introduce tools for automated versioning and patching. We also add a Proof of Concept (PoC) mode that can be used to build a centralized collection of bug-triggering inputs and crash logs, giving users stronger evidence that a fuzzer worked correctly. This update to v1.3 shows that with structured updates and thoughtful tooling, Magma can continue to serve as a practical and realistic benchmark for evaluating fuzzers on real bugs in modern software.

To test our updates, we ran 24-hour fuzzing campaigns using AFL++ [16], Honggfuzz [18], and libFuzzer [41] on nine targets. AFL++ triggered the most bugs (40 total), followed by Honggfuzz (28 total) and libFuzzer (10 total). Of the 127 bugs in the benchmark, 77 were reached during fuzzing and 43 were triggered. Among the 57 bugs that were ported from older versions, 91% were reached and 34% were successfully triggered. These results show that the updated bugs are still meaningful and reachable, and that the benchmark provides useful data on fuzzer performance.

This update turns Magma into a sustainable benchmark that can evolve with the fuzzing ecosystem. We introduce automated tools for versioning and patching, update the infrastructure to support modern environments, and add new features for debugging and bug validation. Together, these changes make Magma v1.3 a reliable and future-proof benchmark for fuzzing research.

5. Conclusion and Outlook

Magma enables accurate and consistent fuzzer evaluation and performance comparison with an open ground-truth fuzzing benchmark, where we forward-ported 118 bugs across seven diverse targets (Magma v1.0). This was followed by Magma v1.2, which includes nine targets and 138 ground-truth bugs. Magma has gained community recognition, and we have continued to improve Magma, releasing v1.3 in 2025,with nine updated targets and 127 ground-truth bugs. By updating the software, refining the infrastructure, and making bugs easier to test, Magma v1.3 advances the state of fuzzing benchmarks. It offers researchers an up-to-date, realistic, and repeatable way to test fuzzers in a ground-truth setting. More importantly, it provides a path forward for maintaining such a benchmark, ensuring that the comparisons it provides remain relevant in the future.

References

[1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. *Announcing OSS-Fuzz: Continuous fuzzing for open source software*. Accessed: 2019-09-09. 2016.

[2] Brad Arkin. *Adobe Reader and Acrobat Security Initiative*. Accessed: 2019-09-09. 2009.

[3] Abhishek Arya and Cris Neckar. *Fuzzing for security*. Accessed: 2019-09-09. 2012.

[4] Andrew Bao, Wenjia Zhao, Yanhao Wang, Yueqiang Cheng, Stephen McCamant, and Pen-Chung Yew. "From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification". In: *USENIX Security Symposium (USENIX Security)*. 2025, pp. 6977–6997.

[5] Craig Beaman, Michael Redbourne, J Darren Mummery, and Saqib Hakak. "Fuzzing vulnerability discovery techniques: Survey, challenges and future directions". In: *Computers & Security* (2022), p. 102813.

[6] Sofiane Benahmed, Abdullah Qasem, Anis Lounis, and Mourad Debbabi. "Modularizing Directed Greybox Fuzzing for Binaries over Multiple CPU Architectures". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2024, pp. 84–103.

[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. 2006, pp. 169–190.

[8] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. "AURORA: Statistical Crash Analysis for Automated Root Cause Explanation". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 235–252.

[9] Sadullah Canakci, Nikolay Matyunin, Kalman Graffi, Ajay Joshi, and Manuel Egele. "Targetfuzz: Using darts to guide directed greybox fuzzers". In: *Proceedings of the 2022 ACM on Asia conference on computer and communications security*. 2022, pp. 561–573.

[10] Xu Chen, Ningning Cui, Zhe Pan, Liwei Chen, Gang Shi, and Dan Meng. "Critical Variable State-Aware Directed Greybox Fuzzing". In: *IEEE/ACM International Conference on Software Engineering (ICSE)*. 2025, pp. 755–755.

[11] Yongheng Chen, Rui Zhong, Yupeng Yang, Hong Hu, Dinghao Wu, and Wenke Lee. "{μFUZZ}: Redesign of Parallel Fuzzing using Microservice Architecture". In: *USENIX Security Symposium (USENIX Security)*. 2023, pp. 1325–1342.

[12] Alexandru Dura and Christoph Reichenbach. "Clog: A Declarative Language for C Static Code Checkers". In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. 2024, pp. 186–197.

[13] Haroon Elahi and Guojun Wang. "Forward-porting and its limitations in fuzzer evaluation". In: *Information Sciences* (2024), p. 120142.

[14] Haoran Fang, Kaikai Zhang, Donghui Yu, and Yuanyuan Zhang. "DDGF: Dynamic Directed Greybox Fuzzing with Path Profiling". In: *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2024, pp. 832–843.

[15] Kai Feng, Marco M Cook, and Angelos K Marnerides. "Sizzler: Sequential fuzzing in ladder diagrams for vulnerability detection and discovery in programmable logic controllers". In: 2023, pp. 1660–1671.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++ : Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. Accessed: 2020-10-19. 2020.

[17] Elia Geretto, Andrea Jemmett, Cristiano Giuffrida, and Herbert Bos. "LibAFLGo: Evaluating and Advancing Directed Greybox Fuzzing". In: *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. 2025, pp. 355–373.

[18] Google. *honggfuzz*. Accessed: 2019-10-19.

[19] Yubo He and Yuefei Zhu. "RLTG: Multi-targets directed greybox fuzzing". In: *PLOS ONE* (2023), pp. 1–23.

[20] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. "Seed selection for successful fuzzing". In: *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2021, pp. 230–243.

[21] Adrian Herrera, Mathias Payer, and Antony L Hosking. "DatAFLow: Toward a data-flow-guided fuzzer". In: *ACM Transactions on Software Engineering and Methodology* (2023), pp. 1–31.

[22] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. "Titan: Efficient multi-target directed greybox fuzzing". In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024, pp. 1849–1864.

[23] Heqing Huang, Anshunkang Zhou, Mathias Payer, and Charles Zhang. "Everything is good for something: Counterexample-guided directed fuzzing via likely invariant inference". In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024, pp. 1956–1973.

[24] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. "DARWIN: Survival of the fittest fuzzing mutators". In: *NDSS*. 2022.

[25] Ajay Joshi, Aashish Phansalkar, L. Eeckhout, and L. K. John. "Measuring benchmark similarity using inherent program characteristics". In: *IEEE Transactions on Computers* (2006), pp. 769–782.

[26] Kieun Kim, Seongmin Lee, and Shin Hong. "Refining Fuzzed Crashing Inputs for Better Fault Diagnosis". In: *Proceedings of the 33rd ACM International Con-*

ference on the Foundations of Software Engineering. 2025, pp. 1248–1249.

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. "Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2123–2138.

[28] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. "Continuous fuzzing: a study of the effectiveness and scalability of fuzzing in CI/CD pipelines". In: *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. 2023, pp. 25–32.

[29] Yuki Koike, Hiroyuki Katsura, Hiromu Yakura, and Yuma Kurogome. "Slopt: Bandit optimization framework for mutation-based fuzzing". In: *Proceedings of the 38th Annual Computer Security Applications Conference*. 2022, pp. 519–533.

[30] Dimitri Kokkonis, Michaël Marcozzi, Emilien Decoux, and Stefano Zacchiroli. "ROSA: Finding Backdoors with Fuzzing". In: *IEEE/ACM International Conference on Software Engineering (ICSE)*. 2025, p. 720.

[31] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. "An Empirical Examination of Fuzzer Mutator Performance". In: *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2024, pp. 1631–1642.

[32] Myungho Lee, Sooyoung Cha, and Hakjoo Oh. "Learning seed-adaptive mutation strategies for greybox fuzzing". In: *IEEE/ACM International Conference on Software Engineering (ICSE)*. 2023, pp. 384–396.

[33] Junru Li, Zhongxu Yin, Guoxiao Zong, and Haiya Sang. "GDFuzz: An efficient directed fuzzing method based on XAI". In: *Journal of Systems and Software* (2025), p. 112568.

[34] Penghui Li, Wei Meng, and Chao Zhang. "{SDFuzz}: Target States Driven Directed Fuzzing". In: *USENIX Security Symposium (USENIX Security)*. 2024, pp. 2441–2457.

[35] Shaohua Li and Zhendong Su. "Accelerating fuzzing through prefix-guided execution". In: 2023, pp. 1–27.

[36] Hao Ling, Heqing Huang, Yuandao Cai, and Charles Zhang. "Efficient Fuzzing Infrastructure for Pointer-to-Object Association". In: *ACM Trans. Softw. Eng. Methodol.* (2025). Just Accepted.

[37] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. "GIANTSAN: Efficient Memory Sanitization with Segment Folding". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2024, pp. 433–449.

[38] Hao Ling, Heqing Huang, Chengpeng Wang, Yuandao Cai, and Charles Zhang. "GiantSan: Efficient Operation-Level Memory Sanitization with Segment Folding". In: *ACM Trans. Comput. Syst.* (2025).

[39] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. "An empirical study on the effectiveness of static C code analyzers for vulnerability detection". In: *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2022, pp. 544–555.

[40] Yinxi Liu and Wei Meng. "Dsfuzz: Detecting deep state bugs with dependent state exploration". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 1242–1256.

[41] LLVM Foundation. *libFuzzer*. Accessed: 2019-09-06.

[42] Simon Luo, Adrian Herrera, Paul Quirk, Michael Chase, Damith C Ranasinghe, and Salil S Kanhere. "Make out like a (multi-armed) bandit: Improving the odds of fuzzer seed scheduling with t-scheduler". In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 2024, pp. 1463–1479.

[43] Dekang Ma and Fuli Shi. "Research on coverage-guided fuzzing technique based on Magma dataset". In: *International Conference on Software Engineering and Computer Applications*. 2025, pp. 176–182.

[44] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *IEEE Transactions on Software Engineering* (2019).

[45] Hyeonmin Mo, Jongmun Yang, and Yunho Kim. "RCFuzzer: Recommendation-based Collaborative Fuzzer". In: *Journal of Systems and Software* (2025), p. 112564.

[46] Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, and Sébastien Bardin. "Fine-grained coverage-based fuzzing". In: *FUZZING 2022 - 1st International Fuzzing Workshop*. 2022.

[47] Nikhil Parasaram, Earl T. Barr, Sergey Mechtaev, and Marcel Böhme. "Precise Data-Driven Approximation for Program Analysis via Fuzzing". In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023, pp. 611–623.

[48] Karl Pearson. "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* (1901), pp. 559–572.

[49] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. "NEZHA: Efficient Domain-Independent Differential Testing". In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 615–632.

[50] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2155–2168.

[51] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. "Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites". In: *IEEE In-*

ternational Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005. 2005, pp. 10–20.

[52] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 31–47.

[53] Ruixiang Qian, Quanjun Zhang, Chunrong Fang, Lihua Guo, and Zhenyu Chen. "FunFuzz: Greybox Fuzzing with Function Significance". In: *ACM Transactions on Software Engineering and Methodology* (2025), pp. 1–34.

[54] Ruixiang Qian, Quanjun Zhang, Chunrong Fang, Ding Yang, Shun Li, Binyu Li, and Zhenyu Chen. "Dipri: Distance-based seed prioritization for greybox fuzzing". In: *ACM Transactions on Software Engineering and Methodology* (2024), pp. 1–39.

[55] Tim Rains. *Security Development Lifecycle: A Living Process*. Accessed: 2019-09-09. 2012.

[56] Timothée Riom, Sabine Houy, Bruno Kreyssig, and Alexandre Bartel. "Evaluating the maintainability of forward-porting vulnerabilities in fuzzer benchmarks". In: *International Conference on Software Maintenance and Evolution*. 2025.

[57] Huanyao Rong, Wei You, XiaoFeng Wang, and Tianhao Mao. "Toward Unbiased Multiple-Target Fuzzing with Path Diversity". In: *USENIX Security Symposium (USENIX Security)*. 2024, pp. 2475–2492.

[58] Yuyang Rong, Chibin Zhang, Jianzhong Liu, and Hao Chen. "Valkyrie: Improving fuzzing performance through deterministic techniques". In: *Journal of Systems and Software* (2024), p. 111886.

[59] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. "Mc2: Rigorous and efficient directed greybox fuzzing". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 2595–2609.

[60] Dongdong She, Adam Storek, Yuchong Xie, Seoyoung Kweon, Prashast Srivastava, and Suman Jana. "Fox: Coverage-guided fuzzing as online stochastic control". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2024, pp. 765–779.

[61] Prashast Srivastava, Stefan Nagy, Matthew Hicks, Antonio Bianchi, and Mathias Payer. "One fuzz doesn't fit all: Optimizing directed fuzzing via target-tailored program state restriction". In: *Proceedings of the 38th Annual Computer Security Applications Conference*. 2022, pp. 388–399.

[62] Xiaolei Wang, Ruilin Li, Bin Zhang, Chao Feng, and Chaojing Tang. "Practical Object-Level Sanitizer with Aggregated Memory Access and Custom Allocator". In: *IEEE/ACM International Conference on Software Engineering (ICSE)*. 2025, pp. 2854–2866.

[63] Wei-Cheng Wu, Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin, and Christophe Hauser. "Fine-grained coverage-based fuzzing". In: *ACM Transactions on Software Engineering and Methodology* (2024), pp. 1–41.

[64] Jifan Xiao, Peng Jiang, Zixi Zhao, Ruizhe Huang, Junlin Liu, and Ding Li. "Robust, Efficient, and Widely Available Greybox Fuzzing for {COTS} Binaries with System Call Pattern Feedback". In: *USENIX Security Symposium (USENIX Security)*. 2025, pp. 6239–6258.

[65] Hang Xu, Liheng Chen, Shuitao Gan, Chao Zhang, Zheming Li, Jiangan Ji, Baojian Chen, and Fan Hu. "Graphuzz: Data-driven Seed Scheduling for Coverage-guided Greybox Fuzzing". In: *ACM Transactions on Software Engineering and Methodology* (2024), pp. 1–36.

[66] Hang Xu and Jianshan Peng. "ViScheduler: Visualized Seed Scheduling Based on Runtime Features". In: *2024 10th International Conference on Computer and Communications (ICCC)*. 2024, pp. 815–820.

[67] Yijiang Xu, Hongrui Jia, Liguo Chen, Xin Wang, Zhengran Zeng, Yidong Wang, Qing Gao, Jindong Wang, Wei Ye, Shikun Zhang, et al. "ISC4DGF: Enhancing Directed Grey-Box Fuzzing with LLM-Driven Initial Seed Corpus Generation". In: *arXiv preprint arXiv:2409.14329* (2024).

[68] Songtao Yang, Yubo He, Kaixiang Chen, Zheyu Ma, Xiapu Luo, Yong Xie, Jianjun Chen, and Chao Zhang. "1dfuzz: Reproduce 1-day vulnerabilities with directed differential fuzzing". In: *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2023, pp. 867–879.

[69] Haruki Yoshida, Yuichi Sugiyama, and Ryota Shioya. "AceCov: Auxiliary Composite Edge Coverage for Fuzzing". In: *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. 2025, pp. 1021–1033.

[70] Dongsong Yu, Yiyi Wang, Chao Zhang, Yang Lan, Zhiyuan Jiang, Shuitao Gan, Zheyu Ma, and Wende Tan. "xFUZZ: A Flexible Framework for Fine-Grained, Runtime-Adaptive Fuzzing Strategy Composition". In: *Proceedings of the ACM on Software Engineering* (2025), pp. 69–91.

[71] Bowen Zhang, Wei Chen, Peisen Yao, Chengpeng Wang, Wensheng Tang, and Charles Zhang. "SIRO: Empowering Version Compatibility in Intermediate Representations via Program Synthesis". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2024, pp. 882–899.

[72] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. "Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators". In: *USENIX Security Symposium (USENIX Security)*. 2025.

[73] Kunpeng Zhang, Xiaogang Zhu, Xi Xiao, Minhui Xue, Chao Zhang, and Sheng Wen. “SHAPFUZZ: Efficient fuzzing via Shapley-guided byte selection”. In: *NDSS*. 2023.

[74] Yunhang Zhang, Chengbin Pang, Stefan Nagy, Xun Chen, and Jun Xu. “Profile-guided system optimizations for accelerated greybox fuzzing”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 1257–1271.

[75] Han Zheng, Flavio Toffalini, Marcel Böhme, and Mathias Payer. “MendelFuzz: The Return of the Deterministic Stage”. In: *Proceedings of the ACM on Software Engineering* (2025), pp. 44–64.

[76] Anshunkang Zhou, Heqing Huang, and Charles Zhang. “KRAKEN: Program-Adaptive Parallel Fuzzing”. In: *Proceedings of the ACM on Software Engineering* (2025), pp. 274–296.

[77] Zhengxiang Zhou and Cong Wang. “Practical anti-fuzzing techniques with performance optimization”. In: *IEEE Open Journal of the Computer Society* (2023), pp. 206–217.

[78] Zhengxiang Zhou, Cong Wang, and Qingchuan Zhao. “No-fuzz: Efficient anti-fuzzing techniques”. In: *International conference on security and privacy in communication systems*. 2022, pp. 731–751.

[79] Shunkai Zhu, Jingyi Wang, Jun Sun, Jie Yang, Xingwei Lin, Tianyi Wang, Liyi Zhang, and Peng Cheng. “Better Pay Attention Whilst Fuzzing”. In: *IEEE Transactions on Software Engineering* (2024), pp. 190–208.

[80] Hongsheng Zuo, Yong Fang, Peng Jia, Ximing Fan, Xi Peng, YiJia Xu, and Rui Pan. “Directed fuzzing based on path constraints and deviation path correction”. In: *Information and Software Technology* (2025), p. 107875.