FUZZ3: Entropy as a Third Oracle

Karine Even-Mendoza[1][0000-0002-3099-1189], Janine Obiri[2][0009-0002-3356-3689], Aidan Dakhama[3][0009-0002-7318-7964], Phil McMinn[4][0000-0001-9137-7433], and W.B. Langdon[2][0000-0002-6388-4160]

[1] King's College London, UK. karine.even_mendoza@kcl.ac.uk
[2] University College London, UK. {janine.obiri.25, w.langdon}@ucl.ac.uk
[3] University of Edinburgh, UK. adakhama@ed.ac.uk
[4] University of Sheffield, UK. p.mcminn@sheffield.ac.uk

Abstract. Fuzz testing finds security issues and improves robustness, however it has only two implicit test oracles: timeout and crash. Information theory gives a third: entropy, which is generic, low cost and widely applicable. FUZZ3 is programming language agnostic, treating software as a black box, searching its input space based on entropy distributions. Applied to 6 open source software, it found 4 bugs, 2 of which have already been fixed since we reported them to the original developers.

Keywords: SBSE · Entropy · Fuzz Testing

1 Introduction: Entropy as a Third Fuzz Testing Oracle

Whereas software testing usually refers to the process of running software with given inputs and checking that it produces the expected output (known as the test's oracle), fuzz testing gains its power by automatic bulk testing and requires huge volumes of test cases which it creates itself. Unfortunately, typically, there is no corresponding automated way of generating the expected output corresponding to each new input [10]. (Although automated test oracles are possible in special cases, e.g. in quantum computing [9].) Instead, fuzzing relies on two implicit test oracles, that should typically hold: 1) programs should not crash (e.g. on divide by zero errors) and 2) they should stop within a reasonable time. Thus, fuzzing automatically detects and records crashes and exceptions, while hangs are controlled using a user-defined timeout. Although there are only two implicit oracles, fuzz testing has proved very successful in practice, particularly for security and robustness [1,4]. We suggest a third, entropy, based on information theory; which is similarly universal. We show that entropy can be incorporated into the fuzz test process and can be useful as an indicator of software problems. This should not be confused with Marcel Bohme's Entropic fuzzer [3], which uses ideas from Biological statistical sampling theory to choose how much effort to spend on testing known paths within the program's source code, rather than FUZZ3's use of "hyper-testing" to quantify information flows that may indicate information leaks, non-determinism, and other software misbehaviour.

$Entropy = -\sum_i p_i \log_2(p_i)$ can be thought of as how much information a probability distribution (p_i) contains. In a wide class of deterministic programs, information (entropy) enters via its inputs, is processed, and the program outputs an answer and stops. In the course of computation, information can be lost but not gained, so the entropy of the distribution the outputs cannot be bigger than that of the inputs. That is, $Entropy_{in} \geq Entropy_{program} \geq Entropy_{out}$. If any of the inequalities are violated, this indicates the program is non-deterministic or is obtaining undeclared information (e.g. via an external API or a network), which may indicate a security problem. Measuring entropy can highlight these issues and helps guides fuzz testing.

Contributions. We investigate entropy as a feedback mechanism and third oracle for fuzz testing, asking:

RQ1: Can entropy be used to identify odd or invalid program behaviours beyond crashing and hanging (i.e., the oracles used by "traditional" fuzzers)?

RQ2: What bugs or program behaviours are exposed by an entropy oracle?

We formalise entropy as a fuzz testing oracle (RQ1). We propose a new fuzzing strategy that integrates it alongside traditional oracles, and we design and implement Fuzz3. Fuzz3 is a modular blackbox language independent fuzzer that supports entropy as a first-class oracle while remaining compatible with existing oracles such as crash and hang oracles (RQ1 & RQ2). We evaluate Fuzz3 on 6 C, C++ or Python open-source projects.

Availability. See ref. [6] for Fuzz3, scripts, input corpus, logs and bug reports.

2 Motivating Example: Noisy Triangle Program

The well-known classic Triangle Program classifies three side lengths into 6 types (outputs 0 to 5) [5][12][13, appendix A.8]. In a noise-free environment, the theoretical maximum output entropy is $\log_2(6)$.

To demonstrate Fuzz3, we injected artificial timing noise ($\sigma = 1.8 \times 10^{-6}$) into output. Traditional crash or hang oracles completely miss this subtle non-determinism. We initialised Fuzz3 with all 216 (6^3) possible valid seeds and applied four simple custom mutators (increment, decrement, force isosceles, force equilateral) over 1000 iterations.

We devised four mutations for the test cases. The first two choose uniformly at random one of the three lengths `add_one` increases it by +1, `sub_one` reduces it (-1). There are no negative numbers in the initial seeds; `sub_one` generates some. There are few isosceles and equilateral triangles, so the mutator `isosceles` selects two different sides at random and makes them equal to $\lceil \frac{total}{3} \rceil$, where *total* is the original sum of the three lengths. Similarly `equilateral` sets all 3 to $\lceil \frac{total}{3} \rceil$.

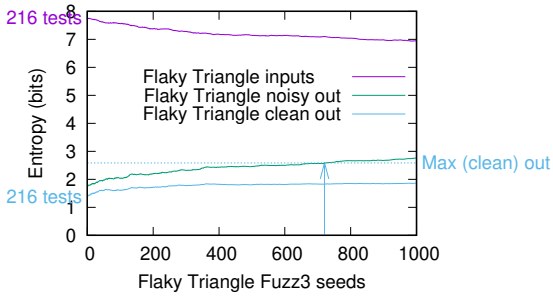

Fig. 1. Entropy of flaky program during Fuzz3 testing. Output entropy *exceeds* limit. Input entropy shouldn't *fall*.

We ran Fuzz3 on our 3.40GHz X86 Linux (Rocky 9.6) desktop for 1000 mutations. Figure 1 reports the inputs and the outputs' entropies. The lower curves show the entropy of the distribution of outputs produced by Fuzz3 mutating the program's inputs 1000 times. As expected, no crashes or timeouts occurred. However, Figure 1 demonstrates how the entropy oracle flags two distinct anomalies. First, the output entropy (green line) exceeds the theoretical maximum of $\log_2(6)$ (indicated by the arrow), successfully detecting the injected noise. Second, the input entropy (purple line) falls. In a healthy fuzzing campaign, input entropy should rise as the test suite diversifies. Input entropy's fall indicates a flaw in the mutators: Instead of exploring new paths, they generate many redundant edge cases. The entropy oracle therefore not only detects subtle bugs, but can also provide a diagnostic signal for fuzzer health.

3 Fuzzing with Fuzz3

Fuzz3, our new blackbox fuzzer, applies entropy-guided feedback with mutation-based fuzzing as a lightweight heuristic to guide exploration, identify interesting seeds and direct corpus evolution over time. We investigate the effectiveness of entropy as a third oracle, alongside crash and hang oracles, and study (§4) its behaviour, its impact on crash and hang detection, and whether measuring input–output diversity helps uncover logical faults without degrading traditional bug-finding performance

Fuzzing Search Strategy & Oracle. Fuzz3 requires sufficient observability over sources of variability; otherwise, entropy cannot meaningfully guide exploration. Hence, Fuzz3 aborts if the input corpus satisfies $\text{Entropy}_{in} \geq \text{Entropy}_{out} + \epsilon$ (default $\epsilon = 0.05$); it may already indicate a potential bug. During fuzzing, a seed is selected iteratively from a queue using a *weighted prioritisation-based entropy scheme*: seeds that previously induced a larger input-output entropy discrepancy are assigned higher priority, and their direct predecessors are upweighted, as they have the potential to lead to similar results. Crashing or hanging seeds are excluded and stored for later analysis. *Oracle:* Seeds are marked *interesting* if they induce notable entropy changes (e.g., output entropy equals zero, or input entropy suddenly fluctuates). Seeds that consistently trigger such changes but do not eventually become part of a population that stabilises entropy are retained for further analysis. *Sliding Window Mechanism:* To ensure stability in the entropy signal, we apply a *sliding window* over recent execution outcomes (default 1024 seeds). This allows entropy to reflect local trends in program behaviour, while also normalising comparisons over time, preventing entropy inflation caused simply by an increasing number of observed seeds.

Fuzz3 Structure. To better understand its effect across different SUTs (Software Under Test), Fuzz3 is modular, implemented in Python and draws on ideas from the LibAFL methodology [8], thus its architecture separates executors, mutators, observers, oracles, and optional generators (all selected via command-line arguments). The framework supports Bash-based targets, Python wrappers (including Pytest), and direct execution of compiled binaries via a generic executor in Unix. It takes an initial seed corpus and repeatedly selects seeds from a queue

Table 1. Fuzz3 supported mutators

Mutator	Description T=Text N=Numeric C=Code	T	N	C
bit_flip	Flips a random bit in the input	✓	✓	✓
delete_line	Removes a random line	✓		✓
delete_char	Removes a random character	✓	✓	✓
duplicate_line	Duplicates a random line	✓		✓
duplicate_char	Duplicates a random character	✓	✓	✓
crazy_indentation	Random whites pace and punctuation edits	✓		✓
insert_block_comment	Inserts synthetic C-style comment			✓
none	Returns input unchanged (baseline)	✓	✓	✓
add_one	Increments a random numeric token		✓	
add_one_mixed_tokens	Increments a numeric substring in mixed input		✓	
sub_one	Decrements a random numeric token		✓	
equilateral	Forces three equal numeric values		✓	
isosceles	Forces two equal numeric values		✓	
cmutation_assignment	Mutates assignment expressions (AST-based [7])			✓
cmutation_conditional	Mutates conditional expressions (AST-based [7])			✓
cmutation_duplicate_stmt	Duplicates statements (AST-based [7])			✓
cmutation_jump	Mutates control flow jumps (AST-based [7])			✓
cmutation_unary	Mutates unary operators (AST-based [7])			✓

using our fuzzing search scheme. Each selected seed is executed by an *executor*, which wraps the SUT and runs it with input arguments (taken from the seed). Fuzz3 then applies user-selected *mutators*, *observers*, and *oracles* during fuzzing; e.g., numerical inputs may use ± 1 mutations, whereas code inputs may use line duplication or line deletion. Table 1 lists the mutators Fuzz3 supports.

We implement entropy-aware observers to record streams of input seeds and their corresponding execution outputs. The sliding-window observer has a fixed-size temporal FIFO window. The observers' data are processed by an *oracle* component. We consider two classes of oracles: (i) traditional crash and hang detection, and (ii) an entropy-based oracle that calculates the entropy of both the *distribution* of inputs and the *distribution* of outputs in the sliding window. The information from both oracles is fed back into the entropy-based seed prioritisation mechanism.

4 Evaluation

We evaluate our research questions (RQs) from §1 using the following setup.

Experimental Setup. We evaluated Fuzz3's effectiveness in fuzzing with SUTs, written in different programming languages: ***Target #1*** noisy triangle (see §2). ***Targets #2–#3*** LLVM/Clang & clang-format: fuzzed for compiler and formatting bugs (LLVM 23.0.0git, commit `64e75b1`, Ubuntu 20). ***Target #4*** Pytest + Cirq: fuzzed Cirq 1.6.1 test suite via pytest 9.0.2 (with empty mutator list) to detect flakiness; run on Ubuntu 20, Python 3.11.11 (CPU-only). ***Target #5*** httpcore: fuzzed for GET request bugs on Windows with Python 3.14. ***Target #6*** GCC: fuzzed GCC 13.4.0 (Ubuntu 22), similar to Target #2.

Results RQ1: Entropy can be a useful Implicit Oracle when Fuzzing. As mentioned in §3, entropy has been successfully used to guide the search. We analysed the input-output entropy over 50,000 iterations. All SUTs had higher input entropy than output entropy. GCC and Clang exhibited smaller input–output entropy gaps and more stable output entropy, while clang-format and httpcore were noisier, with larger input-output entropy differences and greater output entropy fluctuations during fuzzing. (All graphs are available at Fuzz3 Git.) Further, it highlighted a discrepancy (Bug #3, below), which, on investigation, resulted in the discovery and reporting of a bug in a non-trivial program.

Results RQ2: Bug Finding. We fuzzed several SUTs (§4) and checked them all for crashes/hangs. We used pytest for Cirq and differential testing for Clang. We found no compiler bugs in Clang-23 and GCC-13. However, during short campaigns on three targets, Fuzz3 exposed a crash, a hang, a test suite health issue and an invalid output. We discuss the bugs in detail below.

Bug #1: *A crash in clang-format due to NUL character in comment:* We found a bug in clang-format. With `-dry-run`, it crashes on an input file containing a `NUL` byte (i.e. `^@`) inside a block comment (see [6], example1-bug.zip):

```
#if defined(AAA) && (BBB + 3) > 9 /*^@comment */ && !defined(CCC)
```

We reported the bug in `LLVM-23`. We traced this bug back to LLVM-10. The LLVM developers *acknowledged* within an hour and *fixed* it on the same day [6].

Bug #2: *clang-format hangs due to* `\x1B` *character in a comment:* `clang-format test.c` (see [6], example2-bug.zip) hangs due to the non-printable, ASCII "escape" character in a comment (hexa of `\x1B` represented by `^[`), which began an incomplete terminal control sequence. We chose not to report it to the LLVM developers, as it is an OS-level issue; redirecting output to a file resolves it.

Bug #3: *Failures in CI pipeline of Cirq with pytest due to missing dependencies:* The initial output entropy (2.585) was higher than expected: with no mutators and log order removed, variation should come only from runtime (seconds) and likely be smaller. Upon investigation, missing LaTeX libraries raised errors, e.g.:

```
cirq-core/cirq/contrib/quantikz/circuit_to_latex_render_test.py:41:
    AssertionError
!!! pdflatex failed on run 1 (exit code 1) !!!
! LaTeX Error: File 'standalone.cls' not found.
...
E           pylatex.errors.CompilerError: No LaTex compiler was found
```

caused two `cirq.contrib` tests to fail rather than skipping these if missing external dependencies. Also, error messages appeared in inconsistent order (see [6]). We reported this to Cirq as a CI/health issue, promptly fixed by the developers.

Bug #4: *Bad URL validation & incorrect response codes in httpcore:* httpcore incorrectly accepted malformed URLs during URL validation. This example leads to a 301 status code instead of a 404 or an exception, and the redirect provided is an invalid URL. Also, for some erroneous inputs returning a 301 code, the redirect URL was syntactically valid but pointed to an incorrect domain. We reported this bug to the developers.

5 Related Work

As mentioned on the first page, the closest work to ours is that of Böhme et al. [3] and the Entropic fuzzer. Although called Entropic, it uses statistical sampling theory due to Good and Turing to improve fuzzing *efficiency*, not as an additional oracle to find bugs. Clark et al. [11] have also applied the concepts of information theory and entropy to testing, but with the idea of explaining why testing works and when it doesn't. For example, when software errors do not propagate to outputs [2]. This is designed to help developers write better test suites, rather than find bugs when fuzzing, as with our work.

6 Conclusions and Future Work

Fuzz testing is the state of the art in automatically ensuring software quality [1] and yet relies on only two universal implicit test oracles and typically requires access to source code. We provide Fuzz3, a brand new blackbox fuzzer for Unix and Microsoft Windows. It does not need source code and for the first time uses universal information theory to implement an entropy implicit test oracle, as well as the traditional two. Fuzz3 uses hyper testing to quantify information flows that may indicate information leaks, non-determinism and other software misbehaviour. The entropy implicit test oracle is inexpensive and we hope will be seized upon by existing fuzzers. Fuzz3 does not incur the overhead of instrumenting source code, which is essential to whitebox fuzzers and often limits them to a single programming language.

This is an early study and yet Fuzz3 has already been used to search for bugs in 6 real-world non-trivial open source programs written in multiple languages in distant lands. 4 bugs have been found, some of which were acknowledged and fixed within a day by open-source programmers (independent of ourselves).

We built our own fuzzer, nevertheless, as part of future work, we plan to merge the third oracle into LibAFL [8] and assess it with the coverage oracle against state of the art fuzzers such as AFL and Entropic.

We would like to thank our anonymous reviewers.

References

1. Aizatsky, M., et al.: Announcing OSS-Fuzz (2016), https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html, Accessed 1 Apr
2. Androutsopoulos, K., et al.: An analysis of the relationship between conditional entropy and failed error propagation in software testing. In: ICSE 2014
3. Bohme, M., et al.: Boosting fuzzer efficiency: An information theoretic perspective. Commun. ACM **66**(11), 89–97 (2023). https://doi.org/10.1145/3611019
4. Daniele, C., et al.: LibAFLstar: Fast and state-aware protocol fuzzing. In: ESORICS. pp. 105–123 (2025). https://doi.org/10.1007/978-3-032-07894-0_6
5. DeMillo, R.A., et al.: Hints on test data selection: Help for the practical programmer. IEEE Computer (1978). https://doi.org/doi:10.1109/C-M.1978.218136
6. Even-Mendoza, K., Obiri, J., Dakhama, A., McMinn, P., Langdon, W.B.: Artifact of Fuzz3: Entropy as a third oracle (2026). https://doi.org/10.5281/zenodo.19494172
7. Even-Mendoza, K., et al.: GrayC: Greybox fuzzing of compilers and analysers for C. In: ISSTA 2023. pp. 1219–1231. https://doi.org/10.1145/3597926.3598130
8. Fioraldi, A., et al.: LibAFL: A framework to build modular and reusable fuzzers. In: SIGSAC CCS 2022. pp. 1051–1065. https://doi.org/10.1145/3548606.3560602
9. Langdon, W.B.: Implicit test oracles for quantum computing. arXiv 2409.14076
10. Langdon, W.B., Yoo, S., Harman, M.: Inferring automatic test oracles. In: SBST. pp. 5–6 (2017). https://doi.org/10.1109/SBST.2017.1
11. Patel, K., et al.: An information theoretic notion of software testability. Inf. Soft. Tech. **143**. https://doi.org/https://doi.org/10.1016/j.infsof.2021.106759
12. Ramamoorthy, C.V., et al.: On the automated generation of program test data. TSE **2**(4), 293–300 (1976). https://doi.org/doi:10.1109/TSE.1976.233835
13. Watson, A.H.: Structured Testing: Analysis and Extensions. PhD, Princeton (1996)