Barely Broken — Testing Highly Structured Inputs

Valentin Huber
valentin.huber@cispa.de
CISPA Helmholtz Center for Information Security
Germany

Abstract

Current test generators are limited in the inputs they produce. This leads to an imbalance in the program parts tested, with significant program parts reaching insufficient attention, particularly with programs expecting inputs that are highly structured. In this work, I propose two measurements to evaluate different approaches and find which parts of programs are under-tested. *Input correctness* provides a level to which an input satisfies the expected structure, measured either against an implementation or specification. *Input interestingness* checks the distance to equivalence class boundaries.

I present the distribution of *input correctness* from an initial evaluation of different approaches on a C compiler, suggesting significant limitations of existing approaches. I further propose three approaches to closing the gap in testing all input parts.

CCS Concepts

• **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords

Fuzzing, Grammar Fuzzing, Language Fuzzing

ACM Reference Format:
Valentin Huber. 2018. Barely Broken — Testing Highly Structured Inputs. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1.change title

1 Introduction

2.number programs in fig

Programs processing input typically do so in several distinct steps, with the programs in Figure 1 passing increasingly more of these steps:

(1) **Lexing**: Unstructured input is split into an unstructured list of tokens. `(P1)` will be rejected by the lexer of a C compiler.

```
(P1): VBq-"7.6arK w;zu%6$K>%OTV"ryD*
```

```
(P2): && float += % " ( ; == [
```

```
(P3): int main() { return i; }
```

```
(P4): int main() { return 0; }
```

Figure 1: C programs that are lexically (P1), syntactically (P2) and semantically (P3) invalid and fully valid (P4).

(2) **Syntactic Parsing and Checks**: If the input is syntactically valid, it is parsed into structures. While `(P2)` consists of exclusively valid tokens, they do not appear in the correct structure to be parsed into an abstract syntax tree and are therefore rejected by the parser of a compiler.
(3) **Semantic Checks**: These parsed structures are tested on their semantic meaning. `(P3)` can be parsed, but is still not valid — the variable `i` is not defined.
(4) **Business Logic**: Finally, the input is processed by a program's inner logic, e.g. translated into an executable.

My observation is that test generators generally produce inputs that test only a subset of the steps above, depending on their internal model of the input structure. Some of my previous work [10, 11] has explored this by building purpose-built fuzzers that ensure (partial) correctness of inputs.

Imagine the input space of a program expecting highly structured input as a bush, with bugs represented by berries spread throughout, as shown in Figure 2.

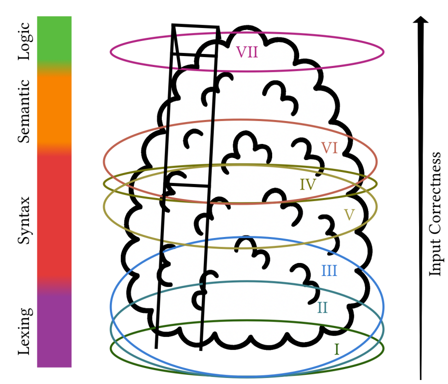

Figure 2: Bush representing the input space of a program, where increasing height represents increasing correctness of a certain input

Purely random input generation, like the first fuzzers [13] is unlikely to test anything but the lexing stage, it is therefore similar to reaching only the very bottom of the bush (I).

In a next step, coverage-guided fuzzing was proposed, which is able to incrementally find inputs that reach deeper into the program with random mutation [18], and thus reaches further up the tree (II). These approaches were extended with instrumentation and logic that allow them to produce increasingly correct inputs [6, 9], thus extending their reach once more (III).

By manually providing the test generator with a model of the target input structure, an analyst can help a fuzzer produce increasingly correct inputs. This is the equivalent of helping a test generator climb on the first step of a ladder.

One widely used approach is to provide a test generator with a context-free grammar, based on which it can produce inputs that will by construction not be rejected by the parser and thus reach further into the program (IV). From there, they can produce inputs that validate their inner model [4, 7] to reach further down (V), or attempt to pass additional semantic checks [8] (VI).

Recent advancements provide a generic model for input testers to receive a complete model of the PUT input structure [19], which allows testing the business logic (VII).

3.Talk about symbex and target-specific fuzzers?

To my knowledge, there is no systematic evaluation of representatives of these approaches — we do not know the extent of their reach. Following this, we do not know what parts of targets remain untested, even when multiple approaches are combined.

Based on this insight, I present the following contributions:

(1) I propose two axes on which to compare the distribution of inputs from different test generators: input *correctness*, and input *interestingness* (Section 2)
(2) I provide initial experiments and their results, which suggest that existing input generators are incapable of testing all program logic (Section 3)
(3) I propose three approaches to creating input generators that fill these holes (Section 4)

2 Evaluating Test Input Quality

Evaluating test generators can be done in multiple, orthogonal ways. The most common measurement is the code coverage reached when executing a certain input on an instrumented implementation of the PUT. While code coverage is comparatively easy to measure due to the availability of instrumentation passes in popular compilers [1, 2]**4.citation format of these** and binary-only fuzzing tools [12], it has two fundamental limitations: (1) It assigns equal weight to all branches and (2) assumes any value covering a new branch is equally interesting.

2.1 Input Correctness

Limitation (1) of using code coverage in evaluating a fuzzer's performance means that fuzzers that reach far into the PUT but only provide valid inputs may receive the same score as a fuzzer that tests all error paths in the initial input validation steps of a program. Alternatively, I propose evaluating fuzzers for *input correctness*. The correctness of an input with regards to a PUT can be evaluated in two ways:

(1) First, one can test inputs on an implementation of a PUT, measuring the ratio of accepted to rejected inputs at each stage of the input processing pipeline. This can further be refined by annotating or instrumenting the code to record the step at which a certain input is rejected. This can be achieved with one of the following:
   (a) Manual annotations of steps along the path through the program, similar to [5, 15]
   (b) Selective coverage instrumentation of entire program parts as a very coarse measurement [14]
   (c) I further propose the following heuristic: Based on a diverse set of (fuzzer-produced) inputs, *instrument the edges executed by all valid inputs*.
(2) Alternatively, inputs can be evaluated against an abstract specification of the language expected by a PUT. Such specifications are usually given in natural language. Recent works have explored how to express such specification in a structured way, as a combination of a context-free grammar and additional constraints over nodes in this grammar [19]. Evaluating the correctness against such a specification may be done in different steps:
   (a) For inputs that cannot be parsed with the provided grammar, existing literature provides algorithm to measure how close to valid the input is. [3, 16]
   (b) For inputs that are parsed, the ratio of fulfilled to violated constraints can be used to assess the correctness of an input.

2.2 Input Interestingness

Not all inputs are equally interesting, even if they are all fully correct, or are processed by the exact same instructions (as described in Limitation (2)). Assume a constraint that checks whether a person is an adult. According to the principle of equivalence classes, mutating their age to random values, say from 27 to 43, is unlikely to trigger a new bug. Changing it to values near the boundaries of equivalence classes like 18 or 19 is a more promising strategy to discover bugs in edge cases, such as off-by-one errors. Alternatively, one can mutate an input to a generally interesting value, like 0, -1, or 2^{16} [18]**5.AFL or AFL++?**.

Previously, it was non-trivial to evaluate how close an input part is to such a constraint boundary. With the availability of formal and machine-readable specifications, such as described above, this now becomes possible. An input (part) is more interesting the closer to the boundaries provided by the specification it is. This can be applied to semantic constraints, but also syntactic structure: If a node can be repeated an arbitrary number of times, testing 0 repetitions, 1 repetition and a large number of repetitions is more interesting. This can be reinterpreted as syntactic boundary testing.

An alternative approach to incentivize a fuzzer to produce more interesting inputs is to instrument comparison instructions on parts of the input in a way that reward the fuzzer for inputs that achieve smaller differences between the two values that are compared. This approach is limited by the ability of the instrumentation of the fuzzer to distinguish between comparisons that represent closeness to equivalence class boundaries, as opposed to comparisons that are independent of the input or do not represent boundaries. Absent an

Table 1: Ratio of inputs rejected by program steps and total coverage across approaches in clang. Approaches marked with * receive an initial corpus of valid C programs.

	Other	Lexing	Syntax	Semantic	Valid	Cov
*	0.000%	0.000%	0.000%	0.000%	1.000%	12772
(IV)	0.086%	0.181%	0.108%	0.623%	0.001%	13074
(IV)*	0.085%	0.177%	0.106%	0.627%	0.004%	13076
(V)	0.085%	0.176%	0.108%	0.629%	0.001%	13074
(V)*	0.086%	0.183%	0.110%	0.618%	0.004%	12921
(VI)	0.000%	0.000%	0.503%	0.497%	0.000%	12787
(VI)*	0.000%	0.002%	0.498%	0.500%	0.000%	12787

automatic heuristic, one could use manual annotation for selected comparisons [5].

3 Evaluating Existing Test Generators

Section 1 introduces potential limitations of existing input generators. To evaluate this claim, I test representatives from a subset of the proposed categories against the C compiler `clang`. I evaluate the different approaches according to *input correctness* as measured through manual annotation of additional instrumentation in the target.

Table 1 presents the ratios of inputs generated by the evaluated approaches that reach a certain step during program execution. Approach (IV) is represented by a pure grammar-based fuzzer. For (V), outputs from the grammar fuzzer are used both unchanged and binary mutated. (VI) is represented by a coverage-guided grammar fuzzer. All representatives were run twice, once with only self-created seeds, and once with an additional set of 10 high-quality manually written seeds using different aspects of the target language.

The results confirm the suspected inability of the presented approaches to test program parts past the semantic checks. They do however have significant limitations and only partially adhere to general evaluation best practices [17]: I am only testing against one implementation, for one target requiring highly structured inputs. I am evaluating only on *input correctness*, in coarse-grained steps, and not against a specification. Additionally, the examples were only run once for 12 hours each.

In future work, I want to expand on these experiments by extending the number and kind of targets, measurements, and evaluation robustness. I further want to test additional approaches and multiple implementations in each; along with test generators purpose-built for a specific target.

4 Reaching the Rest of the Tree

The results in Section 3 suggest that current approaches for test generators are limited — large parts of clang remain untested. Section 1 provides a conceptual explanation for these limitations. Building on top of these, I present three approaches that may extend the parts of a PUT that are effectively tested by automated test generators:

(1) **Reach down from the top**: Similarly to out-of-grammar fuzzing, I want to explore out-of-language fuzzing, where constraints of a full specification are iteratively and deliberately violated, while the remaining constraints are still fulfilled. This approach could further be improved by out-of-grammar construction of the inputs that are evaluated for constraints, giving the fuzzer theoretical reach from the top of the ladder all the way to the ground.

(2) **Incremental steps towards the top**: Writing fully correct specifications for tools like Fandango is considerable work due to a current lack of automated specification generation. However, providing a few simple constraints to an existing grammar may be enough to allow a fuzzer to reach code it was previously unable to test. This by design will additionally produce inputs that do not match the full specification, thus leading to a similar effect as approach (1).

(3) **Targeted testing**: Due to the availability of tools that able to use full input specifications, I would like to explore fuzzers producing inputs that reach higher interestingness in their test cases by steering input generation towards boundaries of both the input semantics (i.e. the constraints) and the syntax (the structure).

5 Conclusion

In this work I discuss the limitations of existing test generations to test programs that require highly structured inputs. Specifically, I present two measurements to evaluate different test generation approaches. *Input correctness* measures either how far an input gets along the input processing pipeline of a target program, or to what extent it fulfills analyist-given lexical, syntactic and semantic constraints. *Input interestingness* measures how close an input is to equivalenc class boundaries.

I then present results of an initial experiment showing that current generic test generators fail at testing significant parts of a C compiler, as measured by *input correctness*. Finally, I propose three approaches to creating test generators that are able to to test previously untestable program parts. To achieve improved *input correctness*, systematically violate semantic and syntactic rules of an input specification or extend current structure-aware test generators based on context-free grammars with additional, incomplete semantic constraints. To produce inputs with increased *input interestingness*, prioritize inputs closer to equivalence class boundaries.

References

[1] [n. d.]. *gcov—a Test Coverage Program.* https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[2] [n. d.]. *SanitizerCoverage.* https://clang.llvm.org/docs/SanitizerCoverage.html
[3] Alfred V. Aho and Thomas G. Peterson. 1972. A Minimum Distance Error-Correcting Parser for Context-Free Languages. *SIAM J. Comput.* 1, 4 (1972), 305–312. arXiv:https://doi.org/10.1137/0201022 doi:10.1137/0201022
[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars, In NDSS. *NDSS Symposium* 19, 337. doi:10.14722/ndss.2019.23412
[5] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring Deep State Spaces via Fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1597–1612. doi:10.1109/SP40000.2020.00117
[6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*, Vol. 19. 1–15.
[7] Bachir Bendrissou, Cristian Cadar, and Alastair F. Donaldson. 2025. Grammar Mutation for Testing Input Parsers. *ACM Trans. Softw. Eng. Methodol.* 34, 4, Article 116 (April 2025), 21 pages. doi:10.1145/3708517

[8] Aarnav Bos. 2025. Autarkie. https://github.com/R9295/autarkie/
[9] Harrison Green, Claire Le Goues, and Fraser Brown. 2025. FrameShift: Learning to Resize Fuzzer Inputs Without Breaking Them. arXiv:2507.05421 [cs.CR] https://arxiv.org/abs/2507.05421
[10] Valentin Huber. 2024. Differential Fuzzing on coreutils Using LibAFL. (25 06 2024).
[11] Valentin Huber. 2025. *FTZ: A State-Inferring Fuzzer for the TCP/IP Stack of Zephyr.* Master's thesis. Zürich University of Applied Sciences ZHAW.
[12] Romain Malmain, Andrea Fioraldi, and Aurélien Francillon. 2024. LibAFL QEMU: A Library for Fuzzing-oriented Emulation. In *BAR 2024, Workshop on Binary Analysis Research, colocated with NDSS 2024*. San Diego (CA), United States. https://hal.science/hal-04500872
[13] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (12 1990), 32–44. doi:10.1145/96267.96279
[14] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. doi:10.1145/3293882.3330576
[15] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (Oct. 2019), 29 pages. doi:10.1145/3360600
[16] Sanguthevar Rajasekaran and Marius Nicolae. 2016. An Error Correcting Parser for Context Free Grammars that Takes Less Than Cubic Time. In *Language and Automata Theory and Applications*, Adrian-Horia Dediu, Jan Janoušek, Carlos Martín-Vide, and Bianca Truthe (Eds.). Springer International Publishing, Cham, 533–546.
[17] 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 *2024 IEEE Symposium on Security and Privacy (SP)*. 1974–1993. doi:10.1109/SP54263.2024.00137
[18] Michał Zalewski. [n. d.]. *American Fuzzy Lop - Whitepaper.* https://lcamtuf.coredump.cx/afl/technical_details.txt
[19] José Antonio Zamudio Amaya, Marius Smytzek, and Andreas Zeller. 2025. FANDANGO: Evolving Language-Based Testing. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA040 (June 2025), 23 pages. doi:10.1145/3728915

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009