Barely Broken — Fuzzing with Almost Valid Inputs

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

Abstract

1.write me

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 — Fuzzing with Almost Valid 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, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

2.change title

1 Introduction

3.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.
(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. 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.

```
(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)`.

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, 10], 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

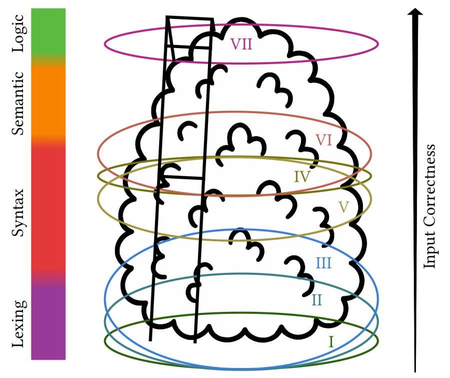

Figure 2: Bush representing the input space of a program, where increasing height represents increasing correctness of a certain input

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).

distribution of bugs like the distribution of berries in a bush, with berries higher up in the bush representing more correct inputs, as shown in Figure 2. Historically, a set of fundamentally different approaches was proposed. These produce inputs that have varying distributions of input correctness and interestingness and therefore test different parts of the program and check for different kinds of software errors. The following section provides a categorization of approaches along with the program parts they test, as indicated in Figure 2. The exact extent of the figurative reach of each fuzzer remains unknown; this figure is not to scale, including the amount of overlap.

Different programs have vastly different requirements for their inputs: zip is designed to process any data during compression, while compilers need to be very strict about the input they allow. Generally, one can distinguish between three steps in input validation, correlating to the examples given in Figure 1. They are increasingly less likely to be fulfilled by random data:

(1) **Lexing**: Random bytes are unlikely to produce exclusively valid input tokens, such as valid keywords, operators, or variable names. The rules for this steps can be given as a list of valid input tokens.
(2) **Syntax**: Even if one would provide a random list of exclusively valid tokens, they likely do not appear in the right structures. The rules for this step are typically given as a context-free grammar.
(3) **Semantics**: Programming languages require many additional checks, such as that each variable is defined before it is used. Recent works propose giving such additional constraints as code evaluated on elements of the grammar [19].

Various approaches have been suggested to produce inputs that adhere to some or all of these constraints [4, 6–10, 14, 19].

In this work, I present the following contributions:

(1) I propose two alternative ways of *evaluating test generators* by examining the inputs they produce for *correctness* and *interestingness*, along with proposals for various approaches to measuring them.
(2) I argue why I suspect that significant portions of the input space of PUTs requiring highly structured inputs remain untested, along with results from initial experiments confirming my suspicion.
(3) Finally, I provide three suggested approaches to closing this gap.

I explore how to evaluate suspected limitations in these approaches for testing PUTs expecting highly structured inputs, along with potential solutions.

4.Talk about prior research somewhere, even if unpublished?

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. But while code coverage is comparatively easy to measure due to the availability of instrumentation passes in popular compilers [1, 2]**5.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.

While code coverage is often evaluated dynamically during a fuzzing campaign to aid the fuzzer in progressing through a program, this has to be distinguished from using code coverage as a technique to evaluate fuzzer performance.

2.1 Input Correctness

Limitation (1) 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. 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, but 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. According to the testing principle of equivalence classes, incrementing a byte is unlikely to trigger a bug. This is of course unless this new byte is at the edge of legality for the PUT, for example by changing this byte so that it is no longer in a valid range. This is known as boundary testing: If a requirement allows values between 5 and 100, the likelihood of discovering a problem is higher with values 4, 5, 6, 99, 100, and 101 compared to values between and outside. Special values such as 0, -1, or the maximum possible number may further be interesting.

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 interesting. This can be reinterpreted as syntactic boundary testing.

3 Evaluating Existing Test Generators

I do not know of any work systematically evaluating correctness and interestingness of the inputs produced by different test generators. Considering that previous work found significant problems with the evaluation of fuzzers [17], evaluating different approaches along the axes outlined in Section 2 may lead to greater insight into what approaches are actually evaluating interesting parts of the target; or more importantly, which approaches produce individual in parts of the input space no other test generator reaches.

For an initial experiment, I ran the grammar fuzzer Nautilus[1] [4] against the JavaScript runtime QuickJS and the C compiler clang with respective grammars. According to the first correctness metric outlined above, I recorded the correctness of inputs produced by the fuzzer.

Running against QuickJS, Nautilus produced approximately 45% inputs that could not be parsed, 5% inputs that resulted in an exception, and 50% inputs that were evaluated without an error. For clang, the results show considerable differences: Only about 50% of inputs made it past the lexing step, and not a single one made it past the parser. One can therefore see how the compiler, assembler, and linker step of clang are never tested with this fuzzing setup. This is likely because clang uses custom lexing and parsing logic that integrates semantic checks. So while the inputs produced by Nautilus are discarded by clang, they by construction can be parsed by the abstract grammar.

The differences in complexity of constraints between JavaScript and C do not explain the difference in result. It seems much more likely that these differences are because of the permissiveness and dynamic typing of JavaScript, while many deeper parts of the interpreter are not actually getting tested. Here, evaluating test correctness against a specification may be more appropriate.

For clang, Nautilus does not seem to be able to generate test inputs that fully compile and thus test all program parts; other approaches should be used, at least in addition to Nautilus.

6.Add plot of correctness levels against time

4 The Ladder to Success: Categorizing Approaches

One can categorize most fuzzers along the correctness of the inputs they provide. Imagine the distribution of bugs like the distribution of berries in a bush, with berries higher up in the bush representing more correct inputs, as shown in Figure 2. Historically, a set of fundamentally different approaches was proposed. These produce inputs that have varying distributions of input correctness and interestingness and therefore test different parts of the program and check for different kinds of software errors. The following section provides a categorization of approaches along with the program parts they test, as indicated in Figure 2. The exact extent of the figurative reach of each fuzzer remains unknown; this figure is not to scale, including the amount of overlap.

The initial approach was using fully random data [13]. As discussed above, this will only test the outermost input validation, equivalent to the very low-hanging fruit (as represented by the ellipsis labeled I).

The next major step was the introduction of naïve mutational coverage-guided fuzzing (CGF) in American Fuzzy Lop (AFL) [18]. By reaching more and more coverage, these fuzzers "learn" the structure the inputs they produce require to reach more coverage. They are however severely limited by the kind of structure they can produce: Even multi-byte comparison operations requires them to guess the value. These fuzzers reached more of the low-hanging fruit in the tree of bugs with no help in the form of an input specification from the analyst (II).

Some of these limitations were mitigated to a point by extending CGF with additional instrumentation. By logging the values input parts are compared to, the fuzzer can mutate the input again to fix these parts and therefore pass the checks [6]. This further allows CGF to pass simple checksums. More recent advancements allow CGF to automatically fix length fields based on sudden coverage drop-offs [10]. These improvements allow the fuzzer to produce more correct inputs and reach further up the tree of bugs (III).

However, certain structure sill cannot be learned by fuzzers from feedback, which is why**7.cite** grammar fuzzers were introduced [4]. These start from a different place, producing inputs that pass at least the lexing and parsing steps of a PUT[2]. By providing the grammar, the analyst "helps" the fuzzer onto the first step of a ladder on the bush where it reaches left and right (IV).

From this next step on the ladder, fuzzers can reach further down by producing inputs that do not (fully) adhere to the grammar, by mutating the grammar or the trees after production (V) [4, 7]. There have been initial steps towards extending grammar-based fuzzers to reach further up, for example by using the same instrumentation on value comparisons as described above (VI) [8].

Language-based fuzzing extends context-free grammars with additional constraints that ensure the inputs produced are not only syntactically but also semantically correct. This correlates to an analyst providing all information necessary for the fuzzer to produce fully correct input (assuming a complete and correct specification), which is equivalent to the topmost step of the ladder (VII).

Fuzzers based on symbolic execution can be thought of as systematically walking each branch of the bush. In some of my previous work, I have discussed the limitations of fuzzers relying on symbolic execution [11].

There further exists a list of fuzzers purpose-built for a certain target specification. They contain information about the expected input format, encoded in their logic. And while they may be highly optimized, they are fundamentally equivalent to a generic fuzzer given the same information in the form of grammar or language specifications and should be categorized accordingly.

5 Reaching the Rest of the Tree

The combined input space coverage as seen in Figure 2 seems to suggest that existing approaches may not reach all program logic;

[1]Nautilus version 2.0 through libafl_nautilus; which notably no longer provides out-of-grammar fuzzing

[2]At least in principle; unless constraint validation is intermixed with parsing

although this will have to be validated with a survey based on the measurements as described in Section 2. The following approaches may do so:

(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.
(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 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).

6 Conclusion

In this work, I outline how suspected limitations of existing test generators; namely their inability to test parts of target software that requires highly structured inputs to be reached. Initial results on a JavaScript interpreter and a C compiler suggest significant parts of such targets may not be tested at all by current test generators.

I further propose multiple measurements to evaluate the inputs of existing and future test generators with regards to their correctness against an implementation and an abstract specification, along with their interestingness as measured by distance to boundaries of equivalence classes.

Finally, I propose multiple approaches of how to design future test generators to fill gaps in existing work, agnostic to specification and implementation.

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] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.
[10] 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
[11] Valentin Huber. 2023. Challenges and Mitigation Strategies in Symbolic Execution Based Fuzzing Through the Lens of Survey Papers. (15 12 2023). https://github.com/riesentoaster/review-symbolic-execution-in-fuzzing/releases/download/v1.0/Huber-Valentin-Challenges-and-Mitigation-Strategies-in-Symbolic-Execution-Based-Fuzzing-Through-the-Lens-of-Survey-Papers.pdf
[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