Testing Programs Expecting Highly Constraint Inputs

Valentin Huber
valentin.huber@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, 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 receiving 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 on a `clang`, indicating that approaches designed to provide structurally valid inputs are unable to test program logic behind the input validation steps. I finally propose three approaches to closing this gap by producing inputs that are both *correct* and *interesting*.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; *Dynamic analysis*; • **Theory of computation** → *Grammars and context-free languages*.

Keywords

Automated Testing, Grammars, Test Generator Evaluation

ACM Reference Format:
Valentin Huber. 2018. Testing Programs Expecting Highly Constraint 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, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

Doctoral Symposium Information. I am in year 1 out of expected 4 in my PhD, supervised by Prof. Dr. Andreas Zeller.

1 Introduction

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

(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**: If the input is syntactically valid, it can be parsed into structures. While `(P2)` consists of exclusively valid tokens, they do not appear in the correct structure.

```
(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).

(3) **Semantic Checks**: The parsed structures are then subject to tests on their semantics. `(P3)` for example can be parsed, but is invalid because 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 [8, 9] has explored this by building purpose-built fuzzers that ensure (partial) correctness of inputs.

1.1 Categorization of Existing Approaches

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

Purely *random input generation* [10] 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 [14] and thus reaches further up the tree (II). These approaches were extended

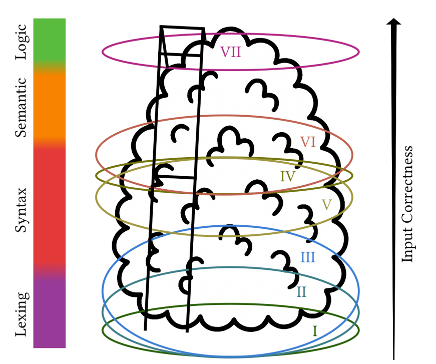

Figure 2: Bush representing the input space of a program, with increasing height representing increasing input correctness, assessed by the input processing steps.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

with *additional instrumentation and logic* that allow them to produce increasingly correct inputs [3, 6], 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 test generator 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 *violate* their inner model [1, 4] to reach further down (V), or attempt to *pass additional semantic checks* [5] (VI).

Recent advancements provide a generic model for input testers to receive a *complete model* of the PUT input structure [15], which allows testing the business logic (VII).

Test generators based on *symbolic executions* systematically attempt to cover all branches in the program and bush, but has significant limitations, as discussed in some of my prior work [7].

To the best of 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 measurements based 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, suggesting that existing input generators are limited in testing all program logic (Section 3).
(3) I propose three approaches to creating input generators that address these limitations (Section 4).

2 Evaluating Test Input Quality

Test generators can be evaluated along multiple, orthogonal axes. To evaluate their ability to reach all program parts, and focus on particularly interesting inputs, I propose two measurements, along with implementation approaches.

2.1 Input Correctness

To test a test generator's ability to reach all program parts, I propose evaluating them for *input correctness*. This can be defined and measured in two ways:

(1) First, one can test inputs **against an implementation** of a PUT, measuring the ratio of accepted to rejected inputs at each stage of the input processing pipeline. This can be achieved with one of the following:
   (a) Manual annotations of steps, similar to [2, 11].
   (b) Based on a diverse set of valid seed inputs, count edges of generated inputs that are executed by *all*, or *any* valid input.
(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 specifications in a structured way, as a combination of a context-free grammar and additional constraints over nodes in this grammar [15]. Evaluating the correctness against specification may be done in different steps:
   (a) For inputs that cannot be parsed, existing literature provides algorithms to measure the closeness of an input to a context-free grammar [12].
   (b) For grammar-valid inputs, the ratio of fulfilled to violated constraints is used.

Table 1: Ratio of inputs rejected by program steps and total coverage across approach categories 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

2.2 Input Interestingness

Not all inputs are equally interesting, even if they are all fully correct or are all processed by the exact same instructions. Assume a constraint checking whether a person is over 18. 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, in this case 18 or 19, is a more promising strategy to discover edge case bugs, such as off-by-one errors. Based on this insight, I define *Input interestingness* as the distance of an input to equivalence class boundaries.

Previous work proposed mutating input parts to contain generally interesting values, such as 0, -1, or 2^{16} [14]. However, the lack of approaches to specify input correctness fully meant that evaluating distance of an input to equivalence class boundaries was limited by an analyst's ability to manually annotate select boundaries [2]. With the advent of abstract full input specifications [15], one can now test this automatically:

(1) **Semantic Interestingness**: Inputs are more interesting if they are closer to boundaries of constraints, as described in the example above.
(2) **Syntactic Boundaries**: If an input part can be repeated an arbitrary number of times, repeating it 0, 1, or 999999 times is more interesting.

3 Initial Experiments and Results

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 test generator. For (V), outputs from the grammar test generator are used, both unchanged and binary mutated. (VI) is represented by a coverage-guided grammar test generator. All representatives were run twice, once with only self-created seeds, and once with an additional set of 10 diverse, manually written, high-quality seeds.

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 [13]. I am only testing against one implementation for one target, which requires highly structured inputs. I am evaluating only on *input correctness*, in coarse-grained steps, and not against a specification. Finally, 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 Bush

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 extend the parts of a PUT that are effectively tested by test generators:

(1) **Reach down from the top**: Similarly to out-of-grammar test generation, I want to explore out-of-language generation, 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 test generator 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 requires considerable effort. However, extending an existing grammar with a few simple, yet incomplete, constraints may be enough to allow a test generator 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 test generators 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 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 analyst-given lexical, syntactic, and semantic constraints. *Input interestingness* refers to the distance between an input and equivalence 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 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.

Acknowledgments

This work is funded by the European Union (ERC S3, 101093186). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] 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

[2] 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

[3] 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.

[4] 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

[5] Aarnav Bos. 2025. Autarkie. https://github.com/R9295/autarkie/

[6] 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

[7] Valentin Huber. 2023. Challenges and Mitigation Strategies in Symbolic Execution Based Fuzzing Through the Lens of Survey Papers. (2023).

[8] Valentin Huber. 2024. Differential Fuzzing on coreutils Using LibAFL. (2024).

[9] Valentin Huber. 2025. *FTZ: A State-Inferring Fuzzer for the TCP/IP Stack of Zephyr*. Master's thesis. Zürich University of Applied Sciences ZHAW.

[10] 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

[11] 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

[12] 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.

[13] 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

[14] Michał Zalewski. [n. d.]. *American Fuzzy Lop - Whitepaper.* https://lcamtuf.coredump.cx/afl/technical_details.txt

[15] 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