

# Shaping Test Inputs in Grammar-Based Fuzzing

José Antonio Zamudio Amaya
jose.zamudio@cispa.de
CISPA Helmholtz Center for Information Security
Germany

## Abstract

Fuzzing is an essential method for finding vulnerabilities. Conventional fuzzing looks across a wide input space, but it cannot handle systems that need intricate and specialized input patterns. Grammar-based fuzzing uses formal grammars to shape the inputs the fuzzer generates. This method is crucial for directing fuzzers to generate complicated inputs that adhere to syntactical requirements. However, existing approaches are biased towards certain input features, leading to significant portions of the solution space being under-explored or ignored. In this paper, we review the state-of-the-art methods, emphasizing the limitations of grammar-based fuzzing, and we provide a first approach for incorporating distribution sampling into fuzzing, accompanied by encouraging first findings. This work can represent a significant step towards achieving comprehensive input space exploration in grammar-based fuzzing, with implications for enhancing the robustness and reliability of the fuzzing targets.

## CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; Software reliability; Software safety; • **Theory of computation** → **Grammars and context-free languages**; • **Security and privacy** → *Systems security*.

## Keywords

Fuzzing, Grammar-based fuzzing, Sampling, Search-based testing

**ACM Reference Format:**

José Antonio Zamudio Amaya. 2024. Shaping Test Inputs in Grammar-Based Fuzzing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650212.3685553

## 1 Introduction

Grammars serve as essential tools in computer science, particularly in areas such as natural language processing and programming language theory [7]. A grammar, formally defined as a set of production rules describing the syntax of a language, facilitates the generation and analysis of valid strings within that language. In the context of software testing and security, we use grammars to define the structure of valid inputs for a system. This technique, known as grammar-based fuzzing, emerges as a powerful method for generating diverse and syntactically correct inputs to test the robustness of systems [9].

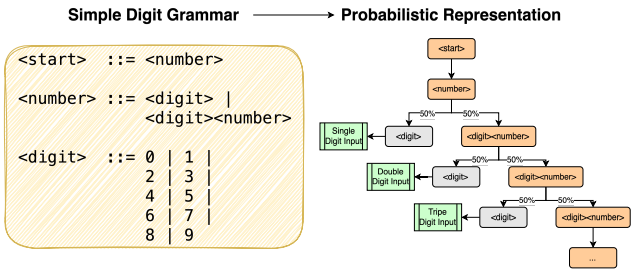



**Figure 1: Grammars represented as probabilistic trees**

Consider the grammar represented in Figure 1 as a probabilistic tree, where at each step, there is a 50-50 decision whether to stop or continue expanding a non-terminal symbol. This process creates unbalance at the generation step, notably, a single digit will have a 50% chance to be produced, a double digit will have a 25% chance, and every time we increase the length of the input, we reduce the probability by half. For instance, our fuzzer will have a 0.0004% chance to produce an input composed by 20 or more digits.

The decision-making process involved in string generation introduces biases toward certain regions, thereby affecting the effectiveness of fuzzing. This inherent bias makes it challenging, or even infeasible, to produce inputs from outside these biased regions. Consequently, this limitation may result in the omission of fault-inducing inputs. Such inputs, which could never be produced by a random fuzzer due to their structural complexity, will potentially never be explored with a grammar-based fuzzer, leaving critical vulnerabilities undiscovered.

Nevertheless, in literature we can find techniques for forcing the fuzzer to produce longer strings. The most common approaches are using a probabilistic fuzzer, where we can modify the grammar in order to guide the fuzzer deeper in the tree, or forcing a minimum number of expansions, usually to a high number, in order to expand more non-terminals and produce longer strings [11]. Figure 2 illustrates the distribution of input lengths generated by three fuzzers for 10,000 inputs, showing that every fuzzer denotes bias, being unable to produce specially long inputs.

Since both extended approaches work over controlling the number of expansions taken, we could potentially only control the length of an input, but it would be impossible to control any other input feature, thus the problem extends beyond input length, as similar biases can affect other input features such as value ranges, constraints, or specific character sequences.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ISSTA '24, September 16–20, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3685553

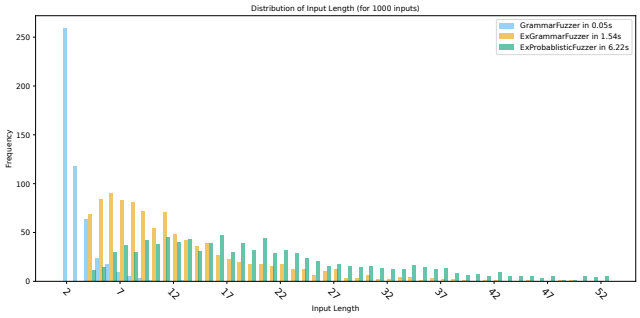



**Figure 2: Input length distribution comparison**

Applying distribution shifting to these properties allows for the generation of tailored inputs that can systematically explore and stress-test different aspects of a system. With this idea in mind, there is a need for an efficient and effective fuzzer that is able to shape the distribution of inputs based on input features. A fuzzer with said capabilities could find security issues in uncovered areas, allocating better resources and finding these vulnerabilities faster than the current approaches.

Previous work has explored techniques for generating longer strings. Despite these efforts, fuzzers still show bias. This bias also affects other input features, potentially missing critical behaviors tied to value ranges, constraints, or specific sequences.

To address these limitations, we have designed an approach that integrates grammar-based fuzzing with search-based optimization techniques to control the distribution of input features. It begins by extracting the grammar and constraints from the subject under test (SUT). Then, a scaffolding is produced, ensuring syntactical validity. The optimizer module then adds values that satisfy the given logical and distribution constraints, achieving the desired distribution for specified input features. Figure 3 shows an example of the intended workflow, with sample artifacts that get converted into a tailored test suite.

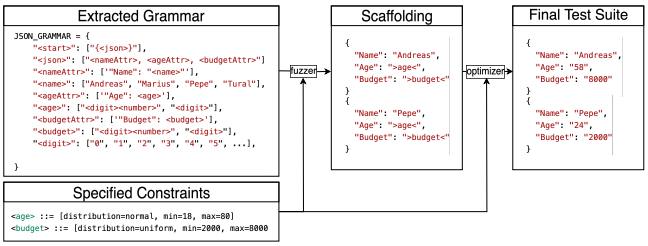



**Figure 3: Workflow example with artifacts**

Preliminary results demonstrate that we can efficiently produce diverse and valid inputs that uncover a broader area compared to state-of-the-art methods. Future work is discussed, introducing the possibility to compile logic, semantic and syntactic constraints in a single fuzzer, potentially outperforming the state-of-the-art approaches.

Our framework integrates grammar-based fuzzing with search-based optimization techniques to produce test suites tailored to logic, distributional and syntactic constraints over input features, improving the diversity of generated inputs, potentially finding more bugs than state-of-the-art methods.

## 2 Related Work

Nevertheless, this work is not the first that addresses and tries to mitigate biases in grammars in the literature. Bar-Hillel et al. [1] laid foundational groundwork by discussing the formal properties of simple phrase structure grammars. Bertoni et al. [2] analyzed the complexity of computing the number of strings of a given length in context-free languages. Hickey & Cohen [6] investigated uniform random generation of strings in context-free languages, exploring techniques to achieve uniformity in the sampling process, and Mairson [8] presented methods for generating words uniformly at random from context-free languages. His work addressed the need for unbiased sampling techniques in grammar-based fuzzing, to this day, the challenge still remains open. Gore et al. [4] proposed a quasi-polynomial-time algorithm for sampling words from context-free languages, aiming to improve the efficiency of sampling procedures. This algorithm is limited in both space and time complexity, and cannot deal with complex grammars. Steinhöfel & Zeller [10] proposed the first grammar-based fuzzer that can handle logical constraints over single elements. Their solution involves using ILP solvers that cannot scale on large input sets and cannot deal with complete set properties. Despite past efforts, there is currently no widely accepted general approach for sampling input features towards specific distributions within the grammar-based fuzzing framework. This gap in methodology motivates the need for further research to develop techniques that effectively address biases in grammar-based fuzzing and enable controlled exploration of the input space.

## 3 Goal Statement

The goal of this thesis is to develop a tool that automatically generates better-distributed tailored test suites by leveraging grammars and constraints extracted from programs. This tool will allow users to specify desired distributions for various input features, and then produce a specified number of valid inputs that not only meet the program's constraints and structure but also effectively stress the system based on the chosen input feature distributions, unifying logic, semantic and syntactic constraints in a single fuzzer. By systematically exploring and shifting the input feature distributions, this tool aims to find bugs and vulnerabilities, enhancing the robustness and security of software systems. This tool allows for the generation of diverse and targeted test cases that explore various regions of the input space, including those that are typically underrepresented in state-of-the-art fuzzing techniques. To achieve these goals, we aim to address the following research questions:

- **RQ1**: How can we effectively utilize grammars and constraints to generate valid and diverse test inputs?
- **RQ2**: How can we integrate distribution goals into the input generation process to uncover edge cases and rare conditions?

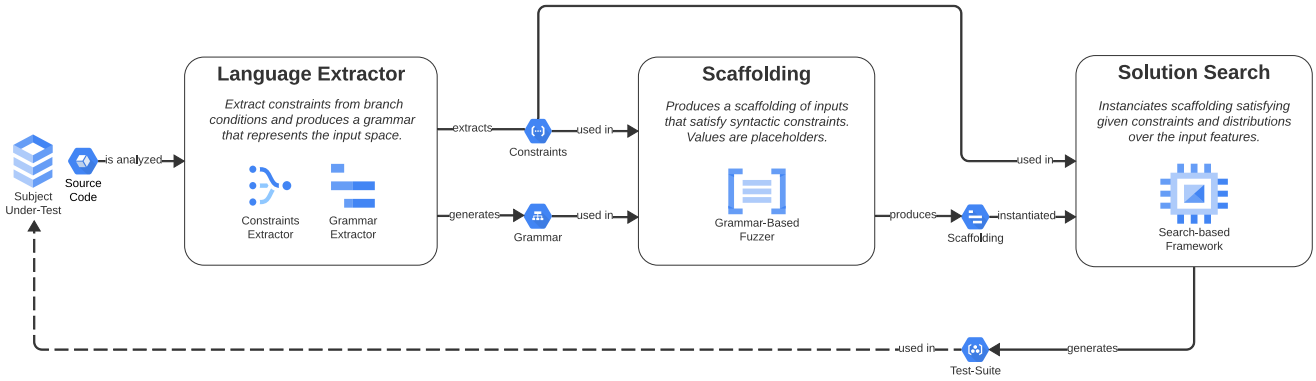



**Figure 4: Proposed framework, consisting on three modules**

- **RQ3**: How can we compile grammars, constraints and distributions into an optimized executable for efficient test input generation?
- **RQ4**: How can the integration of coverage-based testing improve with better-distributed inputs?
- **RQ5**: How can programs benefit from better-distributed tailored test sets generated to stress specific input features?

## 4 Abstract Framework

We have designed a framework that combines grammar-based fuzzing and search based testing to produce a tailored test suite that takes into account logic and distribution constraints. The process can be made as a standalone operation without any necessary call to the SUT during the generation of the test suite, potentially outperforming state-of-the-art coverage guided fuzzing methods. As shown in Figure 4, the process consists of three key steps:

**Language Extractor** Extract constraints and the grammar from the Subject Under Test (SUT) source code.
- In: SUT source code.
- Out: Grammar and input constraints.

**Scaffolding** Generate a scaffolding for the set of inputs, ensuring syntactical constraints are satisfied while using placeholders for values.
- In: Grammar of the SUT.
- Out: Scaffolding of inputs with placeholder values.

**Solution Search** It replaces placeholders with real values that satisfy logic and semantic constraints, producing the desired distribution for the specified input features.
- In: Scaffolding of inputs with placeholder values and input constraints.
- Out: Tailored test suite satisfying given constraints.

In the proposed framework, constraints can be both logical and distributional. Logical constraints specify conditions that individual values must satisfy, and distribution constraints specify the shape of generated inputs.

The transformation of these constraints into the fitness function is a crucial step in the optimization process. Logical constraints are incorporated into the fitness function by penalizing any generated input that violates these conditions, thereby ensuring that the generated inputs satisfy the required logical properties. Distribution constraints are handled by evaluating how closely the generated inputs match the desired distribution pattern. This approach ensures that the generated inputs not only meet logical requirements but also adhere to the specified distribution patterns.

To address the limitations discussed previously, we have developed a framework that integrates grammar-based fuzzing with search-based testing, taking into account both logical and distribution constraints. The framework involves extracting constraints and grammar from the SUT, generating input scaffolding with syntactical placeholders, and using an optimization module to replace placeholders with real values that meet specified logic and distribution criteria.

## 5 Implementation

As a proof-of-concepts, we have implemented both the scaffolding and the optimizer module. We assume the extractor module is implemented, and we have extracted our grammar and the constraints from the system, since there are ongoing research efforts aimed at automatically extracting near-perfect grammars from programs [3] and deriving constraints from branch conditions [5].

The Scaffolding Module is built using a state-of-the-art grammar-based fuzzer, which takes the grammar and produces an initial scaffolding of the total input set, with placeholder values. Then, the Optimizer Module, built using a genetic algorithm, takes the constraints of each value into account and iterates over an initial seed over a given number of generations to fit over the constraints, in order to generate the desired values. At the moment, we have implemented logical constraints and three distribution constraints, that guide the penalization system.

Once the optimal set of values is computed according to constrains and distributions, the scaffolding is instantiated by the genetic framework, producing a complete test suite of inputs tailored to the user specification, which will be later used to test the SUT. With this mechanism that shapes distributions over input features, we expect to find hidden bugs that fuzzers will not be able to detect in complex systems.

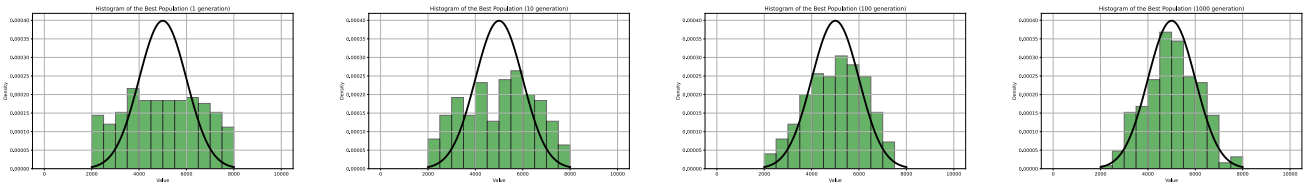



**Figure 5: Transformation of the *Budget* attribute over 1,000 generations**

## 6 Preliminary Results

We have implemented a web app that accepts JSON objects with 3 attributes, *Name*, *Age* and *Budget*. The accepted range for *Age* is between 18 and 80. The accepted range for *Budget* is between 0 and 10,000. To test this SUT, we will generate 1,000 JSON inputs with a uniform distribution of the attribute *Age* where $18 < Age < 80$, and a normal distribution of the attribute *Budget*, where $2000 < Budget < 8000$. This proof-of-concept, and the obtained results are available in the official GitHub repository[1].

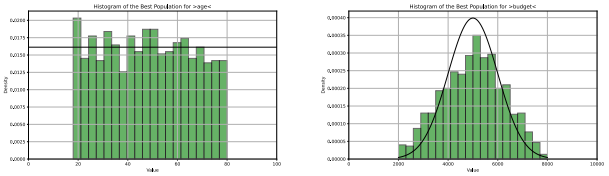



**Figure 6: Final sampling of *Age* and *Budget***

In Figure 5 we can see how the attribute *Budget* has been shaped over 1,000 generations producing a normal distribution, ranging from 2,000 up to 8,000, showcasing the genetic algorithm. In Figure 6 we can see the final shape of the attributes *Age* and *Budget*, satisfying specified logical and distributional constraints.

> Our framework demonstrates the ability to shape the distribution of values satisfying constraints efficiently. This approach could help uncover more bugs and vulnerabilities by systematically exploring underrepresented regions.

## 7 Evaluation

To validate the effectiveness of our tool and address the proposed research questions, we will conduct an evaluation encompassing usability, speed, bug-finding capabilities, and overall performance. We expect to validate the following aspects of the tool:

- Generating valid and diverse test inputs, being the first fuzzer to consider logic, semantic, and syntactic constraints.
- Finding at least the same percentage of known bugs and potentially uncovers previously unknown bugs.
- Providing a user-friendly interface and workflow that is easy for testers and developers to use effectively.

The road-map for the evaluating each research question should be as follows:

[1] https://github.com/joszamama/unifuzz

[**RQ1**]: Implement and test the tool on various software systems using grammars and constraints extracted from the programs. Compare the generated inputs with those produced by ISLa [10].

[**RQ2**]: Configure the tool to generate inputs with various distributions and test it on benchmark programs with known vulnerabilities. Compare the bug-finding effectiveness of different distribution configurations.

[**RQ3**]: Develop a prototype that compiles the grammar and constraints into a standalone executable. Measure the performance in terms of input generation speed and resource utilization. Conduct stress tests with large input sizes and complex constraints.

[**RQ4**]: Integrate coverage measurement into the tool and generate inputs with different distributions. Test these inputs on various software systems and measure code coverage. Compare the code coverage achieved by our tool with other coverage-guided fuzzers over several iterations of feedback.

[**RQ5**]: Generate test sets with specified distributions focusing on different input features and apply them to a variety of programs. Evaluate the stress levels on specific input features and their impact on the program's behavior.

## 8 Discussion

This research thesis serves as a baseline for combining grammars, constraints, and distributions into a single tool capable of generating tailored input sets designed to stress specific input features. Our framework is the first to propose logic, distributional, and syntactic constraint-solving in fuzzing, which can be used for uncovering new areas, performing automatic boundary testing, targeted fuzzing, and stressing specific input features. We also expect this approach to be particularly useful in the testing of AI systems, as it can produce training sets with given feature distributions. By unifying these aspects, our tool will provide a comprehensive solution for finding bugs by concentrating on specific areas of interest within the input space. Through the discussed experiments, we aim to validate the tool's efficiency, effectiveness, and usability, potentially demonstrating improved speed, better bug-finding capabilities, and practical usability in real-world software testing scenarios. We envision creating a fast, efficient, and general high-level intelligent fuzzer that respects logical, syntactical, and semantic constraints.

## Acknowledgements

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] Yehoshua Bar-Hillel, Micha Perles, and Eli Shamir. 1961. On formai properties oî simple phreise structure grammars. *STUF-Language Typology and Universals* 14, 1-4 (1961), 143–172. https://doi.org/10.1524/stuf.1961.14.14.143

[2] Alberto Bertoni, Massimiliano Goldwurm, and Nicoletta Sabadini. 1991. The complexity of computing the number of strings of given length in context-free languages. *Theoretical Computer Science* 86, 2 (1991), 325–342. https://doi.org/10.1016/0304-3975(91)90023-U

[3] Leon Bettscheider and Andreas Zeller. 2024. Look Ma, No Input Samples! Mining Input Grammars from Code with Symbolic Parsing. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 522–526. https://doi.org/10.1145/3663529.3663790

[4] Vivek Gore, Mark Jerrum, Sampath Kannan, Z Sweedyk, and Steve Mahaney. 1997. A quasi-polynomial-time algorithm for sampling words from a context-free language. *Information and Computation* 134, 1 (1997), 59–74. https://doi.org/10.1006/inco.1997.2621

[5] Christopher A. Healy and David B. Whalley. 2002. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering* 28, 8 (2002), 763–781. https://doi.org/10.1109/TSE.2002.1027799

[6] Timothy Hickey and Jacques Cohen. 1983. Uniform random generation of strings in a context-free language. *SIAM J. Comput.* 12, 4 (1983), 645–655. https://doi.org/10.1137/0212044

[7] Tao Jiang, Ming Li, Bala Ravikumar, and Kenneth W Regan. 2009. Formal grammars and languages. In *Algorithms and Theory of Computation Handbook, Volume 1*. Chapman and Hall/CRC, 549–574. https://doi.org/10.1201/9781584888239

[8] Harry G. Mairson. 1994. Generating words in a context-free language uniformly at random. *Inform. Process. Lett.* 49, 2 (1994), 95–99. https://doi.org/10.1016/0020-0190(94)90033-7

[9] Sevak Sargsyan, Shamil Kurmangaleev, Matevos Mehrabyan, Maksim Mishechkin, Tsolak Ghukasyan, and Sergey Asryan. 2018. Grammar-based fuzzing. In *2018 Ivannikov Memorial Workshop (IVMEM)*. IEEE, 32–35. https://doi.org/10.1109/IVMEM.2018.00013

[10] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. 583–594. https://doi.org/10.1145/3540250.3549139

[11] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2024. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/


Received 2024-07-07; accepted 2024-07-22