Grampus - A (Crappy) Grammar Fuzzer

0xca7

December 15, 2021

Abstract

This is the documentation for *Grampus*, the very simple grammar fuzzer I coded for fun. The name Grampus is derived from the mythological figure known as *Krampus* [1] in Austria and Germany. Krampus is the evil counterpart to St. Nikolaus and is usually depicted as a furry (fuzzy), scary looking demon. Because I wrote a grammar fuzzer, I found it appropriate to swap the K for a G resulting in *Grampus*.

1 Introduction

I wrote Grampus for multiple reasons. The first is, I wanted to code something in Rust to get a better understanding of the language. Second, I wanted a fuzzer that is capable of generating valid inputs automatically instead of having to throw together a corpus manually. The third reason is that wanted to learn more about fuzzing. The fourth and final reason is the fuzzer *Nautilus* of Aschermann et al. [2] which is the direct inspiration for Grampus. That fuzzer is just damn cool and I wanted to build something (remotely) like it. For more about information about fuzzing check out *The Fuzzing Book*, it is a very good resource [3] (also covers grammar fuzzers) and also the survey paper by Manès et al. [4].

To make this clear: In contrast to Nautilus, Grampus is blind, that means there is no coverage feedback. In addition, it is much, much, much less sophisticated . . . what did you expect? Look at the title . . .

2 Architecture

Grampus has a pretty easy to understand architecture as you would expect. The architecture is depicted in fig. 1, the numbering in the graphic is used in the following paragraph.

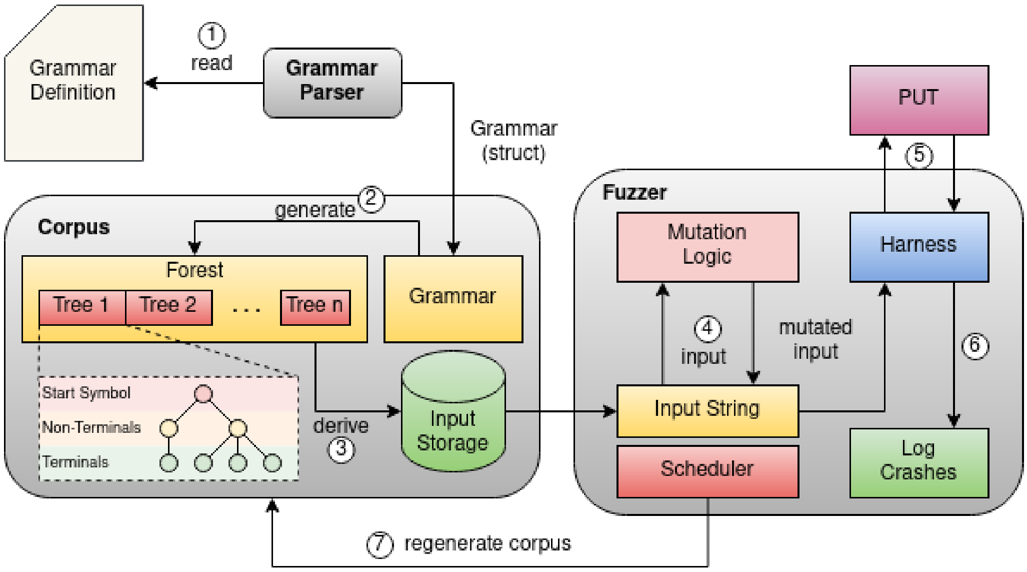

Figure 1: Architecture of Grampus

The grammar definition resides in a file on disk. In a first step, the *Grammar Parser* reads this file ①. The result of this operation is an internal representation of the grammar, which is used to randomly generate multiple syntax trees (forest) ②. These syntax trees represent valid sentences produced by the grammar read from the file. The root node of each tree is the start symbol of the grammar and a leaf node is always a terminal symbol. If a node has children, it is a non-terminal symbol. These trees are then used to derive inputs for fuzzing ③. For each fuzz case, the *Fuzzer* takes a random input from the input storage and applies a mutation to it via the *Mutation Logic* ④. The mutated input is used by the harness, which passes it to the Program Under Test (PUT) and reads the result ⑤. In the case the result is a crash, the input is logged to a file ⑥. After the Scheduler determines it is time to generate new inputs (see Section 3), it triggers a re-generation of inputs in the Corpus entity ⑦.

3 Mutations

Currently, there are only basic mutations available, these are applied to an input string s, with length s_l. The i-th position in an input string is denoted as s_i, $i \in [0; s_l]$. Mutations are applied in cycles, for instance the *Deterministic* Cycle (abbreviated to *Det*) applies only specific mutations, while *Havoc* (abbreviated to *Hav*) applies all available mutations. Which cycle the fuzzer is in, is determined by the Scheduler. Once all cycles are finished, new inputs are generated from the grammar. Currently implemented mutations are:

Mutation	Cycle	Function	Description
Bitflip	Det, Hav	$flip(s, pos, x)$	flip a random bit x in a random character pos
XOR	Det, Hav	$x \oplus s_i$	xor a random byte x with a randomly chosen character s_i
Arithmetic	Det, Hav	$x \circ s_i$	add a random byte x to a randomly chosen character s_i
Remove	Hav	$Rem(s, pos)$	remove a character at a random position pos
Insert	Hav	$Ins(s, pos, x)$	insert a random byte x at a random position pos

At most mut_{max} mutations are performed on one input string s, with the number of mutations chosen at random in the interval $[1; mut_{max}]$ in each fuzzing iteration. The result of input mutation is denoted as s_{mut}.

For future implementations, more sophisticated mutations are planned. These are summarized below.

- Combination - if possible, take two syntax trees and combine them to a new tree. For example, if a non-terminal *NT* is present in both syntax trees, replace the subtree with the root *NT* in tree 1 with the subtree with root *NT* of tree 2. This is what Nautilus does to get *deeper* into a program.
- Bitflip L/S as implemented in AFL.

4 Testing Grampus

I ran Grampus against the *kson* component of *klib*. This module contains a JSON parser, thus, Grampus uses the JSON grammar defined in *json.txt* in the directory *grammars*. For testing, I wrote the program *example_target.c* in **fuzz_target** directory. This program reads a JSON file and parses it using *kson*. Here, Grampus was able to detect multiple crashes, for instance the one below:

```
# JSON from corpus: [9.51508], after mutation: [:9.5}508]
# Output of example_target with input [:9.5}508]
CHAR:
[:9.5}508]
HEX:
5b 3a 39 2e 35 7d 35 30 38 5d
[0xca7] kson_parse_core returned an error
double free or corruption (out)
Aborted (core dumped)
```

References

[1] Wikipedia, "Krampus — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Krampus&oldid=1058950390, 2021. [Online; accessed 06-December-2021].

[2] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[3] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021. Retrieved 2021-10-26 15:30:20+02:00.

[4] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, pp. 2312–2331, 2021.