# Mining Constraints for Grammar Fuzzing

Michaël Mera
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
michael.mera@cispa.saarland

## ABSTRACT

Grammar-based fuzzing has been shown to significantly improve bug detection in programs with highly structured inputs. However, since grammars are largely handwritten, it is rarely used as a standalone technique in large-spectrum fuzzers as it requires human expertise. To fill this gap, promising techniques begin to emerge to automate the extraction of context-free grammars directly from the program under test. Unfortunately, the resulting grammars are usually not expressive enough and generate too many wrong inputs to provide results capable of competing with other fuzzing techniques. In this paper we propose a technique to automate the creation of attribute grammars from context-free grammars, thus significantly lowering the barrier of entry for efficient and effective large-scale grammar-based fuzzing.

## CCS CONCEPTS

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

## KEYWORDS

Fuzzing, Dynamic Tainting, Attribute Grammars, Context-free Grammars, Input formats

**ACM Reference Format:**
Michaël Mera. 2019. Mining Constraints for Grammar Fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3338983

## 1 INTRODUCTION

Grammar-based fuzzing uses formal grammars to describe input formats so that generated inputs are not likely to be rejected early by the program under test and thus reach more interesting regions of the code [7, 16]. Most of the available grammars describing input formats are, however, context-free grammars, which capture the structure of the input but ignore the relationships between the individual elements of this structure. These grammars are not sufficient to express common constraints found in programs such as checksums, field lengths, define-before-use, etc. Figure 1 illustrates the problem: the example inputs are generated by the grammar and are therefore correctly structured; however the grammar does not attach any particular meaning to the field `maxsum`. If this value refers to the upper bound on the sum of the integer values found in the `list` tag, then the input example on the left in Figure 1 will be rejected as invalid by any program that verifies this condition, preventing testing of any code after this check. We see that this grammar does not (and cannot) enforce this semantic.

```
   list   ::=  "<list maxsum=" integer ">" int * "</list>"
    int   ::=  "<int value=" integer "/>"
integer   ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

two examples of generated inputs:

```
<list maxsum=5>                     <list maxsum=42>
    <int value=93563 />                 <int value=15 />
</list>                                 <int value=19 />
                                    </list>
```

**Figure 1: A context-free grammar for an XML-based input format: it defines the syntax but cannot enforce complex semantic.**

To circumvent this problem, we present in this paper a solution to automatically augment context-free grammars into more expressive attribute grammars: given a program and a context-free grammar describing its input format, we extract constraints found at runtime and generalize them into attributes for the grammar. Attributes are similar to symbolic constraints but express the semantic relationships between the grammar elements rather than at the input level (see Figure 2). The resulting grammar can then be used to fuzz the original program by generating inputs that follow *by construction* the extracted semantic. This is drastically different from other fuzzing techniques where the semantic needs to be corrected *a posteriori* for each and every input [3, 12, 13, 15].

$$\sum_i \texttt{to\_int(list.int[i].integer)} \leq \texttt{to\_int(list.integer)}$$

**Figure 2: A semantic attribute expressing the missing semantic of the context-free grammar in Figure 1**

The proposed approach to mine grammar attributes can be described as a three steps process:

(1) **Trace:** Given a valid input, an execution trace of the program under test is generated.
(2) **Filter:** Inside this trace, comparisons that are used to check semantic properties of the input are identified using taint analysis.
(3) **Construct:** Starting from these comparisons, the structure of the check performed is reconstructed. Using the provided

context-free grammar it can be generalized into an attribute that applies to all valid inputs.

The assumption of this technique is that semantic properties of the input are verified explicitly by the parsing program and can thus be retrieved by detecting the relevant comparisons in the code. Therefore, our first research questions are designed to test the validity of this assumption as well as the effectiveness and efficiency of our approach:

**RQ1** To what extent can our approach extract grammar attributes?
**RQ2** How does our approach scale when applied on real-world programs?

Since our goal is to use the extracted grammar for fuzzing, the following additional questions are aimed at evaluating the results of our approach in this context:

**RQ3** How does fuzzing with the extracted attribute grammars compare to using the initial context-free grammars?
**RQ4** How does fuzzing with the extracted attribute grammars compare to other fuzzing methods?

## 2 PROPOSED APPROACH

The main idea of the proposed technique is that a semantic constraint (and thus an attribute) is reflected at some point in the program by a comparison that checks if the constraint holds. To retrieve these comparisons, an execution trace of the program under test is generated using a valid input (Section 2.1). This trace is then analyzed to keep only relevant comparisons (Section 2.2) which can finally be used to construct grammar attributes (Section 2.3).

In this section, the semantically valid example in Figure 1 (right) is used as an input for the parsing code in Figure 3, to illustrate the entire extraction process.

```
1   [...]
2   struct LIST* list = parse_list();
3
4   int sum = 0;
5   for (int i = 0; i < list->nbchildren; ++i) {
6     sum += list->child[i].value;
7   }
8
9   if (sum > list->maxsum) {
10     [... reject invalid input ...]
11  }
12  [...]
13
14  struct LIST* parse_list()
15  {
16    struct LIST* list = malloc(...);
17    list->maxsum = atoi(list_fetch_maxsum());
18
19    int j = 0;
20    while (integer = list_fetch_integer()) {
21      list->child[j].value = atoi(integer.value);
22    }
23
24    return list;
25  }
```

**Figure 3: A simplistic program parsing the format in Figure 1**

### 2.1 Execution Trace

The program under test is first instrumented to create a trace at runtime. This trace includes each instruction executed as well as the corresponding runtime values as shown in Figure 4. The instrumented program is then run with a valid input. This provides the guarantee that the execution trace follows a valid path in the program. Consequently, we know that the result of any comparison along this path applies to valid inputs and that there is no need to analyze the control flow graph of the program to remove comparisons applying to error paths and invalid inputs.

```
            executed instructions:                     runtime values:
list = call parse_list
        list = call malloc(...)                 ...
        list.child[j].v = atoi(...)             mem[0x...] ← to_int("15")
        list.child[j].v = atoi(...)             mem[0x...] ← to_int("19")
        list.maxsum = atoi(...)                 mem[0x...] ← to_int("42")
        return list                             ...
sum = 0                                         mem[0x...] ← 0
i < list->nbchildren                            0 < 2 (true)
sum += list.child[i].v                          mem[0x...] += 15
i < list->nbchildren                            1 < 2 (true)
sum += list.child[i].v                          mem[0x...] += 19
i < list->nbchildren                            2 < 2 (false)
sum > list.maxsum                               34 > 42 (false)
...                                             ...
```

**Figure 4: A (simplified) execution trace obtained by running the program in Figure 3 with a valid input**

In our prototype implementation, we focus on C programs and instrument them at the LLVM bitcode level [11]. The main advantage is the use of a language that is easy to instrument and reason about. Additionally, the large number of programming languages that can be used on top of LLVM opens perspectives for future extensions of this work.

### 2.2 Filtering Comparisons

While it is a reasonable assumption that semantic constraints are mostly expressed in a program by comparisons, the reverse is clearly false: not all the comparisons in a program (or even in a given execution trace) are relevant for the purpose of attribute mining. For example, many comparisons are used to guide the parsing process at the lexical or syntactical level, or to verify configuration. For example, in the trace in Figure 4, only the final comparison expresses a semantic check (corresponding to Figure 3 line 9), whereas the previous three comparisons are part of the loop that computes the sum of the integers (Figure 3 line 5). Hence the need for filtering the trace obtained in Section 2.1 to only keep the comparisons that can be later transformed into attributes.

Attributes are used to check a relationship between syntactical parts of the input. This provides a good criterion to filter out irrelevant comparisons: we consider only comparisons involving at least two distinct non-terminals of the context-free grammar. To that end, dynamic taint tracking is used to link the operands of a given comparison to corresponding input fragments. These input fragments are themselves linked to non-terminals of the grammar using the derivation tree for the considered input. For example in Figure 4 the operands of the last comparison are tainted with the **integer** non-terminals of Figure 1, making this comparison suitable for attribute extraction.

## 2.3 Attribute Construction

Once a relevant comparison is identified, a corresponding attribute is created from the execution trace. This is a three steps process:

(1) The structure of the comparison is reconstructed by iteratively replacing the operands until actual input fragments (or constants) are encountered. To this end, we use the executed instructions present in the trace. As an example, starting from the last comparison in Figure 4, we can reconstruct by going backward in the trace that the `sum` operand is the sum of three distinct elements, we can thus change the initial comparison `34 ≤ 42` into:

```
0 + 19 + 15 ≤ 42
```

Going back even further in the trace, it is possible to retrieve that the integers (except for the constant `0`) are created by transforming a string into integers using a call to `atoi`. This gives the following transformation:

```
0 + to_int("19") + to_int("15") ≤ to_int("42")
```

Since the strings in this last expression match the input fragments that we are looking for, we now stop this first reconstruction step.

(2) The input fragments in the obtained expression are replaced by grammar elements using the derivation tree of the input. Following our previous example, we replace the strings in the last expression by the corresponding non-terminals:

```
0 + to_int(list.int[1].integer) + to_int(list.int[0].integer)
                     ≤ to_int(list.integer)
```

(3) The expression is generalized into an attribute by detecting repeating patterns and matching them to repeating elements of the context-free grammar. In the example, all the `list.int` are found in the sum on the left of the previous expression, we can thus generalize it into the following attribute:

$$\sum_i \texttt{to\_int(list.int[i].integer)} \leq \texttt{to\_int(list.integer)}$$

## 3 EVALUATION

The first part of our evaluation will focus on demonstrating the effectiveness of our method (**RQ1**). To answer this question, we will first assemble a corpus of programs demonstrating common checks of semantic properties found in various input formats. This includes for example magic numbers, common checksums, define-before-use, data length field, data type field, etc. Additionally, for each of these programs a context-free grammar will be provided. Running our prototype on this corpus will provide insight into our ability to extract grammar attributes when similar verification patterns occur in real-world programs.

Once the effectiveness of our method will have been demonstrated in this simplified setting, it is necessary to evaluate its applicability to more complex programs (**RQ2**). However, the question of selecting appropriate test subjects does not have a straightforward answer. While there exist some benchmarks for fuzzing, such as LAVA [6], these programs do not use highly structured inputs. They are therefore not interesting targets for our particular goal. Classical target programs with structured inputs which are currently considered are programming language interpreters and compilers, network packet parsers and archive format parsers. Once the test subjects have been selected, for each of them we plan on computing a ground truth by manually listing all semantic relationships on top of a context-free grammar. The evaluation can then consist of measuring the percentage of these relationships that are correctly extracted by our prototype.

For the last part of our evaluation (**RQ3** and **RQ4**), we plan to use the attribute grammars extracted in the previous experiment for fuzzing. This is not a straightforward process because few grammar-based fuzzing tools currently use attribute grammars and the best generation heuristics have not been studied in depth. We plan on reusing NAUTILUS [2] which is a grammar-based fuzzer capable of accepting arbitrarily complex constraints on top of a context-free grammar. Defects will be introduced manually inside the programs selected in the previous step. They serve as a ground truth for our evaluation. AFL [17] will be used as a baseline for this evaluation. Additionally we will compare our results with both Angora [5] and REDQUEEN [3] which have demonstrated state-of-the-art performances on classical benchmarks.

Note that to completely answer **RQ4**, not only the number of bugs found by each fuzzer has to be compared, but also the location of these bugs. Since we use a grammar-based approach, most of the generated inputs will be syntactically and semantically valid. Therefore, we expect our fuzzer to find dramatically fewer bugs than its competitors in the early stage of the program. However for the same reason, our fuzzer should immediately reach deeper inside the program under test and reveal bugs that are difficult or even impossible to find with the other methods. If that is confirmed, our approach would then be positioned not as a direct alternative to the more classical fuzzers, but rather as a complementary method.

## 4 RELATED WORK

Closely related to this research are the various grammar-based fuzzers like LangFuzz [9], Grammarinator [8], Skyfire [14] or BlendFuzz [16], which have demonstrated the use and efficiency of grammars either as standalone fuzzing methods or in conjunction with mutation fuzzing for seed generation. However, they do not provide any method to obtain the initial grammar, other than writing it by hand. To fill this gap, several methods have been proposed to automate the creation of grammars. GLADE [4] and AUTOGRAM [10] are two examples of such methods. In each case, however, the generated grammar is context-free. By mining attributes to augment context-free grammars, the present work can thus be considered a natural extension of these methods.

Also, while not directly applicable to grammar fuzzing, some methods have been proposed to reach deeper inside the program under test when using other fuzzing techniques. By extracting additional knowledge from the program and by detecting structural or semantic constraints on the input, they are similar in intent to this research. The ones most relevant to this research follow:

- VUzzer [13] leverages taint analysis to link comparisons in the code to particular parts of the input and help the fuzzer pass specific 'magic bytes' checks. Unlike our approach however, more complex conditions are not handled which makes our technique more general.

- Similarly, TaintScope [15] detects comparisons that correspond to checksum verifications. Using partial symbolic execution, the checksums are then repaired in every invalid inputs. Since TaintScope does not leverage structural knowledge of the input, unlike our use of a context-free grammar, it cannot generalize the resolution of the constraints to preemptively fix the inputs.
- REDQUEEN [3] uses a method similar to TaintScope but replaces symbolic execution by a set of predefined patterns and checksums that are used to detect the exact checksums involved and correct the input accordingly. This approach is however limited by the collection of recognized patterns and (like TaintScope) must fix the constraints a posteriori for every input.
- Finally, T-Fuzz [12] uses a different but complementary approach to fuzzing: rather than extracting knowledge from the program to create more valid inputs, it purposely remove the semantic checks in the code, so that the process can continue further. This approach is particularly effective for checksums because the removal of the verification will probably not impact much subsequent processing. For checks verifying more fundamental properties of the input, this can however lead to a large number of false positives.

## 5 CONCLUSION

In this paper we presented a novel technique to address the shortcomings of context-free grammars in the context of fuzzing: starting from an execution trace we showed how a specific type of comparisons can be identified and used to construct grammar attributes.

Additionally, we envision the following contributions:

- A complete benchmark suite of semantic checks commonly used in real-world programs (see Section 3). Designed to answer **RQ1**, it can also be used outside of this study to assert the ability of fuzzers to overcome these "road blocks" in a more controlled environment than the currently available alternatives.
- Answering **RQ3** and **RQ4** will provide a better understanding of the strengths and weaknesses of grammar-based fuzzing when compared to other approaches. To the best of our knowledge, this has not been extensively studied, in particular with respect to the location of bugs.
- Finally, a prototype of our attribute extraction technique will be implemented to work on C programs and will be publicly available for future comparisons.

While still being work in progress, our expectation is that a technique to automatically augment the expressiveness of grammars instead of manually encoding additional semantics will significantly lower the barrier of entry for efficient and effective grammar-based fuzzing.

## ACKNOWLEDGMENTS

This work is supervised by Prof. Dr. Andreas Zeller.

## REFERENCES

[1] 2018. *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8418581

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. https://www.syssec.ruhr-uni-bochum.de/media/emma/veroeffentlichungen/2018/12/17/NDSS19-Nautilus.pdf

[3] Cornelius Aschermann, Sergej Schumil, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. https://www.syssec.ruhr-uni-bochum.de/media/emma/veroeffentlichungen/2018/12/17/NDSS19-Redqueen.pdf

[4] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 95–110. https://doi.org/10.1145/3062341.3062349

[5] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search, See [1], 711–725. https://doi.org/10.1109/SP.2018.00046

[6] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 110–121. https://doi.org/10.1109/SP.2016.15

[7] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 206–215. https://doi.org/10.1145/1375581.1375607

[8] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir (Eds.). ACM, 45–48. https://doi.org/10.1145/3278186.3278193

[9] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Tadayoshi Kohno (Ed.). USENIX Association, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[10] Matthias Höschele and Andreas Zeller. 2017. Mining input grammars with AUTOGRAM. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 31–34. https://doi.org/10.1109/ICSE-C.2017.14

[11] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[12] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation, See [1], 697–710. https://doi.org/10.1109/SP.2018.00056

[13] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[14] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 579–594. https://doi.org/10.1109/SP.2017.23

[15] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 497–512. https://doi.org/10.1109/SP.2010.37

[16] Jingbo Yan, Yuqing Zhang, and Dingning Yang. 2013. Structurized grammar-based fuzz testing for programs with highly structured inputs. *Security and Communication Networks* 6, 11 (2013), 1319–1330. https://doi.org/10.1002/sec.714

[17] Michal Zalewski. [n. d.]. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/