Reverse Engineering Input Syntactic Structure From Program Execution

Salehen S. Rahman
shovonr

Grant Wallis
gbwallis

Dawson Perron
dperron

Abstract

There may come a time when we may need to understand the syntactic structure of inputs. For instance, we may want to generate test cases. Other times the input structure is not published. Or perhaps, the structure is formally published, but it may be erroneous. Here, we present a technique to reverse engineer input syntactic structure from program execution. We are basing our implementation from the works of Lin and Zhang on *Deriving Input Syntactic Structure From Execution* [1].

1 Introduction

Most applications accept structured input with a defined grammar, be it formally or informally defined, where formal being documented, and informal being implemented. Even to this day, with the definition of various formal formats, such as XML, YAML, and JSON, new input structures continually emerge. For instance, recently, Dropbox implemented lepton, an alternative to the JFIF container format for compressing JPEG-encoded data. Not only are we seeing the emergences of new *defined* formats, most malware are also introducing their own, largely for the sake of secrecy.

In the case of emerging formats such as JFIF, reverse engineering input syntactic structure may reveal to us the structure as laid out in a parse tree. If the structure differs from what has been defined in the grammar, then we have a discrepancy.

As for malware, we may be able to implement a parser for the communication protocol that it is using for communicating with a remote end-point.

Other use case is test data generation. Fuzz testing allows us to generate random input, which may lead to interesting paths that may or may not result in a bug. However, random fuzz testing is intended to uncover faults, where as it fails to test for conformance to a standard. However, as was mentioned earlier, if we do not have a standard, then we are unable to generate test cases for a standard, for test cases targeting future iterations.

With the aforementioned use case in mind, we have implemented a technique to reverse engineer the input syntactic structure, inferred from program execution.

2 Approach

Our approach consisted of three parts. The first was creating a trace which contained input for our algorithm. Afterwards we constructed a parse tree from the input. Later we turned the parse tree into a grammar which we defined.

3 Getting Input for the Algorithm

The first thing we needed to do was take the program, and turn it into an acceptable input. The algorithm takes instructions from the program in order to create a tree. For each instruction, we need to know its immediate post dominator. We also need to identify if the instruction is a method call, predicate, getchar or ungetchar, or other. Lastly, we need to know if a predicate or method call uses input from the file directly or indirectly. To store all this information, we used a struct for each instruction containing the needed info. Later we do a dynamic analysis in order to match input from the file to each instruction. Lastly we print a trace which contains correct input for our analysis later.

3.1 Instruction classifying

For instructions we did two things. We mapped each instruction to a unique integer id in order to be sent to the trace later. After that we had to classify the type. To classify an instruction, we labeled it as either

GetChar, UngetChar, “Method Call”, “Predicate”, or “other”. To identify GetChar, we checked for call instructions. If the instruction it called to was fgetc, then it was GetChar. We did the same for ungetchar by searching for ungetc. For all other calls, we labeled it as “Method Call”. To identify predicates, we looked for cmp instructions. For all others, we labeled it as “other”. Once we knew the types, we looked for immediate post dominators.

3.2 Immediate Post Dominators

To find immediate post dominators, we had to handle two cases. The first was finding the immediate post dominator of method calls. For this we took the next instruction inside the basic block as this signifies that the call has ended. For predicates we used a function from LLVM that allowed us to see if a basic block dominates another. To apply this to an instruction, we took the basic block the instruction belonged to. Normally a basic block dominates itself which doesn’t work in our case because we need to know when the predicate’s control over the flow is over. To get around this, we made sure the two basic blocks were not the same. Once we had our function, we ran it over every instruction in the program until it comes to the first instruction that post dominates it. We use our id method from earlier and save the id of the immediate post dominator in the struct.

3.3 Tainted Instructions

An instruction is considered tainted if input from the file affects the instruction in anyway. For any tainted instruction we also want to know which instruction it gets the input from. For this we used a ”find” method that returns the GetChar where the input came from. Later we replace the GetChar dynamically by calling fgetc on the input file whenever we encounter GetChar instruction. For later instructions, use the character from the attached instruction. Unfortunately this approach didn’t work for loops so we had it look back in the for the last GetChar used. Once this was done we were ready to build the parse tree.

3.4 Building the Parse tree

To build the parse tree, we fed our instructions with their features into the algorithm. The algorithm uses a stack to track the location of the instructions inside the program. If an instruction is on the stack, it means the flow of the program in dependendt on the instruction. These are popped when we give an instruction that is an immediate post dominator. An instruction is pushed onto the stack if it is a method call or predicate because these change the flow of the program. Whenever a method call or predicate is found with tainted input, we make a node in our tree labeled with that input. In bottom-up order, we make a node for each instruction

4 Evaluation and Observations

We evaluated the technique with a parser for a sample language that has the following grammar:

```
S -> ’H’ { Characters } ’/H’
Body -> ’B’ { Tags } ’/T’
Tags -> ’T’ { Tags | Characters } ’/T’
Characters -> ? a to z ?
```

With the above grammar, the following are some example valid input strings:

Haa/HBTbb/TTccc/T/B

Haa/HBTbb/TTccc/TThey/T/BB

The sample language is implemented in C, and is compiled using Clang 3.9.1 into LLVM IR.

We pass the IR into our instrumentation tooling. In the instrumentation phase, we uniquely identify each LLVM instruction, from which we also derive the immediate post-dominators of each instructions. After acquiring the ID, and post-dominators, at each instruction, we have the instrumentated application output to a file the instruction ID, the instruction type, the immediate post dominator (also an instruction ID), and the character that it is ”tainted. The output acts as our “trace”.

We then execute the instrumented program with a sample input, from which a trace is generated. From the trace, we then run the algorithm described in section 3.4, to generate a CSV-formatted adjacency list. Per the nature of section 3.4, the adjacency list describes a tree data structure (figure 3).

From the tree data structure we then derive the inferred grammar of our input, which is based entirely on the execution of the sample program.

Below is an example grammar derived from the application execution.

```
10 -> 110 + 115
110 -> 140
140 -> H + 143 + 153
143 -> 216
216 -> a + 216
216 -> a + 216
216 -> /
153 -> H
115 -> 178
178 -> B + 182 + 189
182 -> 273
```

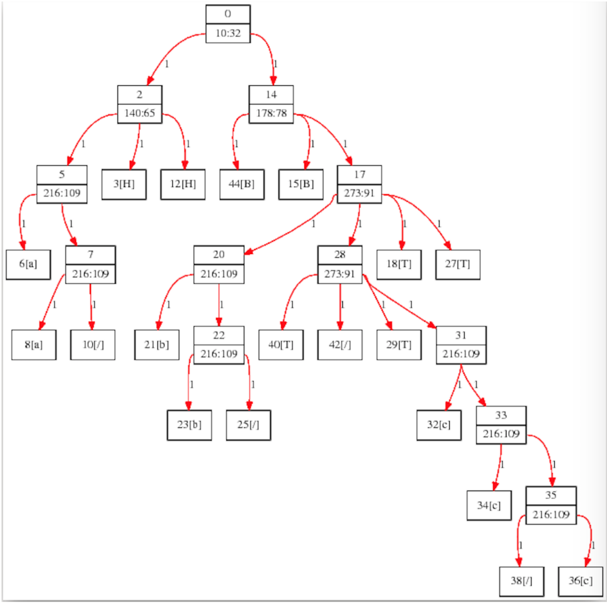

Figure 1: Example tree

```
273 -> T + 280 + 314 + 273
280 -> 216
216 -> b + 216
216 -> b + 216
216 -> /
314 -> T
273 -> T + 280 + 314 + 273
280 -> 216
216 -> c + 216
216 -> c + 216
216 -> c + 216
216 -> /
314 -> T
273 -> /
189 -> B
```

Although the format above is unconventional, if we should easily be able to convert the above grammar into EBNF form.

5 Related Work

Our effort is based on the works by Lin and Zhang on *Deriving Input Syntactic Structure From Execution*. Our contribution, however, is to derive a grammar from the resulting trace.

A close embodiment of our work is one by Hschele and Zeller on *Mining Input Grammars from Dynamic Taints* [2]. The similarity from their work and ours

A distant embodiment of this work is *Grammar Based Whitebox Fuzzing* by Godefroid et. al. [3]. Although, with their work, the grammar needs to be known ahead-of-time, but the similarity between their work and ours is that we have the opportunity to derive test data given a grammar.

6 Future Work

As it stands, our application can only handle character input from a file. This limits the inference of input grammars of applications that read from files sequentially, character-by-character only. Our next step would be to instrument applications that read character by character from TCP sockets, and then more sophisticated parsers, such as `std::getline`.

Another limitation is that this application is only able to infer the grammars that use top-down parsers. Parsers that buffer characters into memory will also hinder.

Of course, the grammar that we do generate from the application is context-free in nature. Although it is unclear how we may be able to derive context-sensitive grammar, but if we do, it will act as a valuable tool for test data generation, and possibly other applications.

7 Conclusion

Having an understanding of the input syntactic structure of an application is very important for reasons including but not limited to test data generation, verification of standards implementation, malware input parsing, etc. In this paper we introduced an implementation of a sample parser application. The implementation generates a parse tree, as well as a resulting grammar. From the grammar, we should be able to pass in to a random input generator for test data generation.

References

[1] L. Zhang, X. Z. Deriving input syntactic structure from execution, 2008.

[2] M. Hschele, A. Z. Mining input grammars from dynamic taints, 2016.

[3] P. Godefroid, A. Kiezun, M. Y. L. Grammar based whitebox fuzzing, 2008.