Reverse Engineering Input Syntactic Structure From Program Execution

Salehen Shovon 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 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. We already have

2 This is Another Section

Some embedded literal typset code might look like the following :

```
int wrap_fact(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]) {
    int result;
    int arg0;
    if (argc != 2) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    arg0 = atoi(argv[1]);
    result = fact(arg0);
    sprintf(interp->result,"%d",result);
    return TCL_OK;
}
```

Now we're going to cite somebody. Watch for the cite tag. Here it comes [**?**, **?**]. The tilde character (˜) in the source means a non-breaking space. This way, your reference will always be attached to the word that preceded it, instead of going to the next line.

3 This Section has SubSections

3.1 First SubSection

Here's a typical figure reference. The figure is centered at the top of the column. It's scaled. It's explicitly placed. You'll have to tweak the numbers to get what you want.

This text came after the figure, so we'll casually refer to Figure 1 as we go on our merry way.

Figure 1: Wonderful Flowchart

3.2 New Subsection

It can get tricky typesetting Tcl and C code in LaTeX because they share a lot of mystical feelings about certain magic characters. You will have to do a lot of escaping to typeset curly braces and percent signs, for example, like this: "The `%module` directive sets the name of the initialization function. This is optional, but is recommended if building a Tcl 7.5 module. Everything inside the `%{, %}` block is copied directly into the output. allowing the inclusion of header files and additional C code."

Sometimes you want to really call attention to a piece of text. You can center it in the column like this:

`_1008e614_Vector_p`

and people will really notice it.

The noindent at the start of this paragraph makes it clear that it's a continuation of the preceding text, not a new para in its own right.

Now this is an ingenious way to get a forced space. `Real *` and `double *` are equivalent.

Now here is another way to call attention to a line of code, but instead of centering it, we noindent and bold it.

`size_t :  fread ptr size nobj stream`

And here we have made an indented para like a definition tag (dt) in HTML. You don't need a surrounding list macro pair.

> `fread` reads from `stream` into the array `ptr` at most `nobj` objects of size `size`. `fread` returns the number of objects read.

This concludes the definitions tag.

3.3 How to Build Your Paper

You have to run `latex` once to prepare your references for munging. Then run `bibtex` to build your bibliography metadata. Then run `latex` twice to ensure all references have been resolved. If your source file is called `usenixTemplate.tex` and your `bibtex` file is called `usenixTemplate.bib`, here's what you do:

```
latex usenixTemplate
bibtex usenixTemplate
latex usenixTemplate
latex usenixTemplate
```

3.4 Last SubSection

Well, it's getting boring isn't it. This is the last subsection before we wrap it up.

4 Acknowledgments

A polite author always includes acknowledgments. Thank everyone, especially those who funded the work.

5 Availability

It's great when this section says that MyWonderfulApp is free software, available via anonymous FTP from

`ftp.site.dom/pub/myname/Wonderful`

Also, it's even greater when you can write that information is also available on the Wonderful homepage at

`http://www.site.dom/~myname/SWIG`

Now we get serious and fill in those references. Remember you will have to run latex twice on the document in order to resolve those cite tags you met earlier. This is where they get resolved. We've preserved some real ones in addition to the template-speak. After the bibliography you are DONE.

Notes

[1]Remember to use endnotes, not footnotes!