S2E input grammar generation

Yu Xia

Australian National University

Supervisors: Adrian Herrera (DST), Michael Norrish (Data61)

Introduction

Currently, there are many different fuzzing types. According to [Godefroid et al., 2017], grammar-based fuzzing is one efficient method. However, the grammar of the program input is hard to gain. It is usually generated manually, which is challenging, time-consuming and error-prone. This research is using the symbolic execution to automatically analyze the parser and generate the valid input grammar accordingly.

Existing Work

Now, there are some input grammar learning tools, including:

- Tupni [Cui et al., 2008]
- GLADE [Bastani et al., 2017]
- Learn & Fuzz [Godefroid et al., 2017]

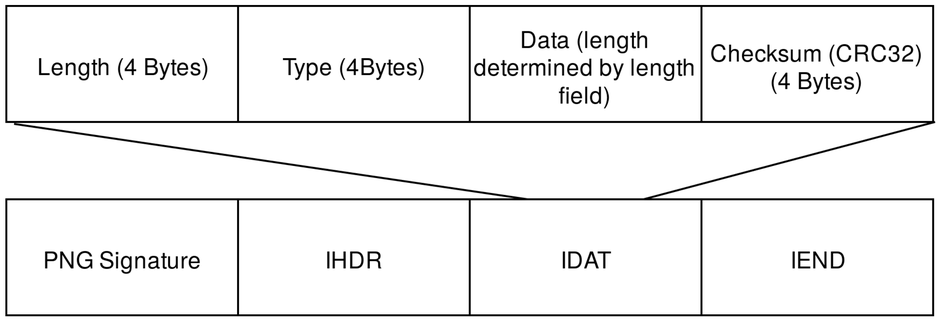
Compared to them, this research 3 contributions:

- Implementation based input grammar generation. The grammar is more specific to implementation
- No initial file preparation.
- New checksum handling method.

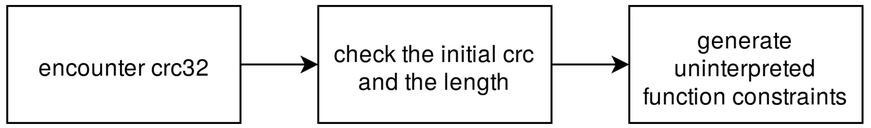
Grammar

The grammar here is a group of description which can identify valid program input.

- Different file will have different grammar to satisfy
- The actual grammar may not satisfy the file specification
- The grammar may not be published due to security reasons



The image shows the requirement in PNG specification

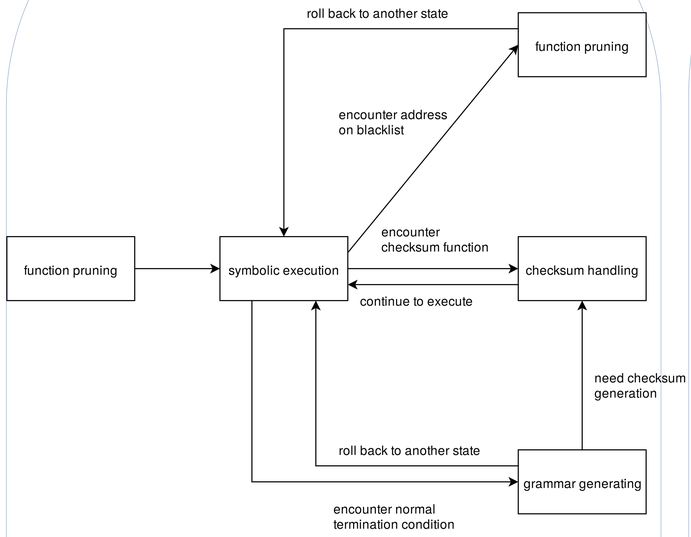


The image shows how we generate grammar of checksum

```
(UF w32 (UF w32 0x0
(Read w8 0xc v0___symfile____tmp_input___0_1_symfile___0) 0x4)
(Read w8 0x10 v0___symfile____tmp_input___0_1_symfile___0)
                0xd)))
```

The image shows the actual grammar of checksum

Working process design



This image shows the working process.

- **Function pruning**: Finding out the function which is irrelevant to the grammar, and preventing analyzing them during symbolic execution.
- **Symbolic execution**: Analyzing the parser, generating the path constraints.
- **Checksum handling**: Using uninterpreted function format to present checksum and avoiding analyzing.
- **Grammar generating**: When the program enter the valid termination condition, generating the grammar for input.

Sample result

Here is an example of PNG grammar (not complete):

```
(Eq 0x474e5089
     (ReadLSB w32 0x0 v0___symfile____tmp_input___0_1_symfile___0))
(Eq 0xa1a0a0d
     (ReadLSB w32 0x4 v0___symfile____tmp_input___0_1_symfile___0))
(Eq 0x49484452
     (Or w32 (Or w32 (Or w32
     (Shl w32 (ZExt w32
     (Read w8 0xc v0___symfile____tmp_input___0_1_symfile___0))
                                        0x18)
     (Shl w32 (ZExt w32
     (Read w8 0xd v0___symfile____tmp_input___0_1_symfile___0))
                                        0x10))
     (Shl w32 (ZExt w32
     (Read w8 0xe v0___symfile____tmp_input___0_1_symfile___0))
                                   0x8))
     (ZExt w32
     (Read w8 0xf v0___symfile____tmp_input___0_1_symfile___0))))
```

Here is an example of generated PNG details:

```
Hex view:
    8950 4e47 0d0a 1a0a 0000 000d 4948 4452
Ascii view:
     . P  N G  . .  . .  . .  . .  I H  D R
Hex view:
    0000 0020 0000 0001 0804 0000 0015 8364
Ascii view:
     . . .    . .  . .  . .  . .  . .  . d
Hex view:
    ec00 0000 1249 4441 5400 0000 0000 0000
Ascii view:
     . . . .  . I  D A  T .  . .  . .  . .
Hex view:
    0000
Ascii view:
     . .
```

Actually, we can find that the actual implementation doesn't match the file specification. **No IEND chunk and no checksum field in IDAT chunk.**

Limitation

- The path explosion problem exists. The complicated binary will prevent further grammar generation
- The problem exist in symbolic execution still get unsolved. For example, the unlimit chunk size will cause problem when generating multiple different PNG chunks.

Future Work

- Supporting more different checksum handling
- Giving solution to unlimited chuck size
- Testing more file parsers
- Implementing new grammar-based fuzzer to use the grammar we generated

References

- Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 95–110. ACM, 2017.
- Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In Proceedings of the 15th ACM conference on Computer and communications security, pages 391–402. ACM, 2008.
- Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pages 50–59. IEEE Press, 2017.