Grail: Introduction to Grammar Learning

Shane Magrath[1]

[1]CCMR
CEWD
Defence Science and Technology Group

October 2016

Outline

Background

- Common reverse engineering problems
  - Specify a **Network Protocol** used by a communicating service
  - Specify an input **File Format** for a target application
- File formats and network protocols are examples of formal *grammars*
- For example: what is the file format that "Atril Document Viewer" v1.14.2 expects its file inputs to comply with?
  - Note: the specific target application defines the expected grammar **not** external standards etc...
- Why do I want the grammar?
  - Because fuzzing is much more effective and efficient when the input structure, type information and semantics are known

Problem

Basic Problem

- Can you automatically learn the grammar a parser is expecting by observing the parser process inputs?
  - **Note**: You are not likely to have the source code, debug symbols or other supporting information. Just the target binary.

Collecting Data

- Parser operation is observable through dynamic program analysis techniques
  - Intel PIN is a binary instrumentation tool that can produce an **assembly instruction trace**
  - Another approach is **API Call tracing**
    - A higher level of abstraction
    - Nice if you can get access to the API...
- A large corpus of sample parser inputs can be collected
  - maximise code coverage of the parser tracing

Grammar Learning

Refined Problem

Can you automatically learn the grammar given the *multiple traces* collected?

Prior Art

- Microsoft Research: "Tupni: Automatic Reverse Engineering of Input Formats", 2008 ACM CCS
  - This research is considered best-practice/state of the art
  - **Essential reading**
  - **Benchmark** - can we do duplicate the results? Can we do better?

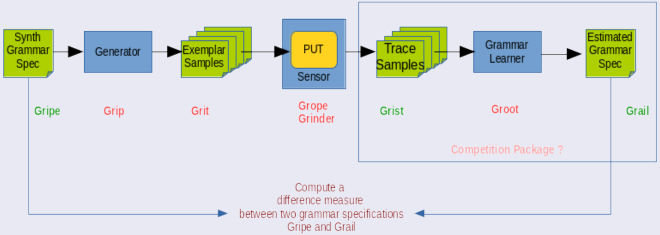
TUPNI Critiques

- **Critiques**:
- Formalise the parser tracing format into using a *data description language* (JSON based).
  - separate the grammar learning process from the data collection process
  - Both are difficult problems of very different natures
- Design a series of benchmark parsers and sample inputs of varying complexity
  - Series of "Challenge Problems" (also known as Unit Tests :-)
  - Allows explicit control of what grammar features you are testing the GL on
  - Transparent scoring of a GL's capabilities
  - Competition! Grand Challenge! Man Versus Data!

Grail: The Grammar Gravy Train :-)

- Architecture breaks up the system design to avoid TUPNI's apparent monolithic approach.
- Focus is on making the relationships between the system components "observable" and subject to "standardised format"
  - Divide and conquer
  - Separation of concerns

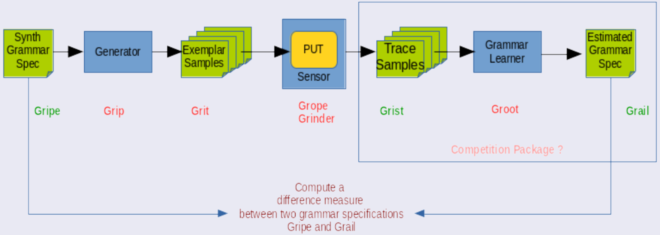
Grail Flowchain



Grail

- **Gripe** and **Grit:**
  - "**Gripe**": synthetic test grammars
  - "**Grit**" files are samples of input for each test grammar
    - both "good" and "bad" samples
- At the moment, Gripe is just an idea.
  - We currently manually create grammars, parsers and Grit files...

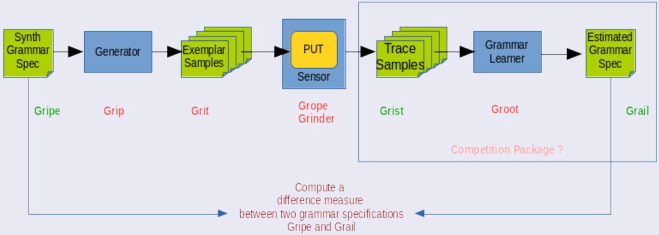
Grail Flowchain



Grail

- **Grope** includes
  - "Parser Under Test" (PUT)
  - "test runner" - specifies what to test, where the Grit files for each test is located, where to store test results,...
- **Grinder** is the means by which we instrument the PUT in Grope to produce trace data.
  - Grinder provides parser API Call tracing
- **Grist** are the traces of the PUT
  - one for each Grit sample
  - a data description language for describing the trace data
  - JSON format

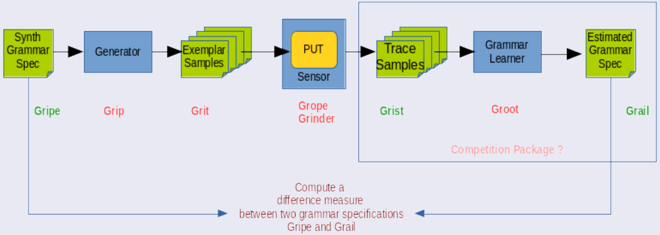
Grail Flowchain



Grail

- **Groot** is the "Grammar Learning" component
  - takes one or more Grist files as input
  - produces one Grail file
- **Grail** is the estimated grammar from the GL component
  - ideally, should be "identical" to the original Gripe specification
  - need a way of computing the actual difference between the two grammars

Grail Flowchain



Grail

- All code is in Python to make the research problem accessible to a wide range of people
  - Avoids dealing with binary instrumentation, dynamic tracing, PIN and assemble instructions ...
  - Future work to create a Grinder implementation for arbitrary binaries that produces trace files in Grist format

What is a “Grammar”

```
Wikipedia:
“a grammar is a set of production rules for strings in a formal language.
The rules describe how to form strings from the language's alphabet
that are valid according to the language's syntax.
A grammar does not describe the meaning of the strings
or what can be done with them in whatever context
-only their form.”
```

- **Grammars** and **Types** are *specifications* which **Parsers** can implement to process *strings* in a *formal* language

Grail Grammar

- Based on **EBNF** (Extended Backus-Naur Form)
- Extended to be able to specify ***Constraints***
  - constraints mean that the grammars don't need to be "*context-free*"
- Only very simple **types** required

Definition

Types

- `UINT8, INT8`
- `UINT16, INT16`
- `UINT32, INT32`
- `UINT64, INT64`
- `Boolean`
- `Floating Point (64 bit, 80 bit)`
- `String (null-term, fixed len)`
- `BLOB (N bytes of binary data)`

Definition

Grammar Operators

1. `Production Rules`
2. `Concatenation`
3. `Repetition`
4. `Constant Constraints`
5. `Alternation`
6. `Containment / Composition`
7. `Functional Relationship Constraints`

Grail Grammar

Here is a simple but important example for an almost useful file format:

Example

```
File := Magic Record+ CRC32
Magic := UINT16 //value == 0x23 0x32
Record := Type Length Value
Type := UINT8
Length := UINT16
Value := BLOB //sizeof(Value) == Length.value
CRC32 := UINT32 //value == crc32(File.value)
```

- Notice the operators in use: *concatenation*, *repetition*, *encapsulation*, *constant and functional constraints*,...
- Also only simple ***primitive types*** used

Grope Parsers

- Test Parsers are implemented in Python PyParsing
  - Grope and Grinder are implemented as Python *Decorators*
- Example:

```
@registry.set(test_files="grit")
@registry.set(enabled=runme)
@registry.set(test_dir=root_dir + "/test1")
@registry.set(result_dir=root_dir + "/test1/results")
@registry.set(name="test_1")
def test_1_factory():
  TermA = gt.trace(Word(nums, exact=1).setName("TermA"))
  TermB = gt.trace(Word(nums, exact=1).setName("TermB"))
  TermC = gt.trace(Word(alphas, exact=1).setName("TermC"))
  TermD = gt.trace(Word(alphas, exact=1).setName("TermD"))
  R1 = gt.trace(TermA + TermB + TermC + TermD + StringEnd())
  R1.setName("R1")
  return R1
```

Tracing: Grist Format

- Grist is simply a JSON encoding of the parser tracing data
- Schema is very simple
    - But I believe sufficient to describe the same data ontology as TUPNI requires
        - That is, an *assembly instruction trace* should be reducible to an *API Call trace* with some good thinking

Grist Example: Partial Trace


```
{"timestamp": 1474250616.730216, "data": "52AW\n",
"trace": [
{"addr": 140344043851088,
     "rtype":"try",
     "seqno": 0,
     "rule": "R1",
    ‘‘line": 1,"col": 1},
{"addr": 140344043850768,
     "rtype": "success",
     "seqno": 2,
     "rule": "TermA",
     "tokens": "['5']",
     "length": 1,
     "line": 1,"col": 1},... ],}
```

Grist Example: Partial Trace

```
{"timestamp": 1474250616.730216, <-- Date Time
"data": "52AW\n", <-- Input string being Parsed
"trace": [ <-- An array/list of tracing operations
{"addr": 140344043851088, <-- Code address in Memory

     "rtype":"try", <-- Trace Operation: Start of Parsing
     "seqno": 0, <-- Index in the trace array/list
     "rule": "R1", <-- Not likely to get function names/rules in practice
     ‘‘line": 1,"col": 1}, <-- Start of Parsing Input

{"addr": 140344043850768, <-- Code address in Memory

     "rtype": "success", <-- Trace Operation: Parsing Success
     "seqno": 2, <-- Trace sequence number
     "rule": "TermA", <-- Function Name/ Parsing Rule Matched
     "tokens": "['5']", <-- Terminal Accepted
     "length": 1, <-- Length of parsing input matched
     "line": 1,"col": 1}, <-- Start of Parsing Input
     ... ],}
```

What Now?

- Grail Project is on our GitLab server
- Half a dozen test parsers available
  - complete with sample Grit inputs and Grist trace data
- More Work Required
  - Groot implementation(s)
    - the fun bit!
  - Metric/Loss Function: We need a difference measure for two grammars
  - Tool: Assembly instruction tracing that produces data in Grist format