New threat grammars

Gueguen Geoffroy, Filiol Eric

Mai 2010

ESIEA Laval $(C+V)^o$,
Université de Rennes 1

Outline

- Introduction
- Grammars
  - General introduction to grammars
  - What are they used for ?
  - W-grammars
- Metamorphism with W-grammars
  - Word generation
  - Integration in libthor
- K-ary codes
  - What are K-ary codes ?
  - Representation of K-ary codes by a W-grammar
- Conclusion

Introduction

- A lot of research has been done on the grammar field.
- Powerful tools to describe things.
- In particular, they are used to describe programming languages.

- Already used to produce polymorphic code.
- Almost not used for metamorphic code.
  - When this is the case, they are not complex enough, and words produced can be parsed too efficiently.

- We will present a type of grammar which can be written with reasonable facility, and can be very powerful (can generate Type 0 languages).

Grammars

Grammars

General introduction to grammars

- What is a grammar ?
  - Σ : an alphabet.
  - N ∉ Σ : a finite set of non-terminal symbols.
  - T (=Σ) : a finite set of terminals symbols, with $N \cap T = \emptyset$
  - S ∊ N : the start symbol.
  - $R \subseteq (T \cup N)^* \times (T \cup N)^*$: a finite set of rewriting rules over Σ, defining how non-terminal and terminal symbols can be combined to form the language.

  - A grammar G is the 4-uple (N,T,S,R) and the language described by G is $L(G) = \{x \in \Sigma^* | S \Rightarrow^* x\}$.

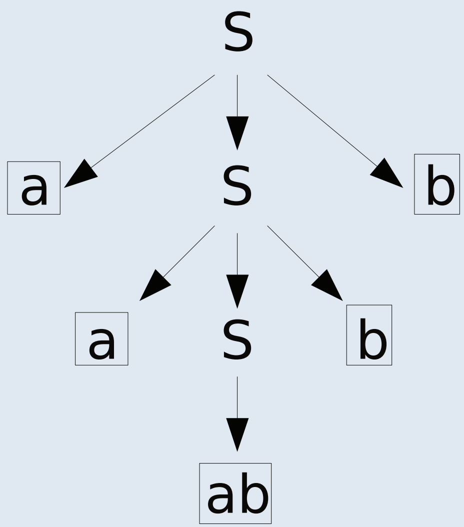
Grammars

General introduction to grammars

- A basic example

  - G = ({S},{a,b},{S},{S : aSb, S : ab})
  - G defines the language $\mathcal{L}(G)$ = $\{a^n b^n \mid n > 0\}$

Parsing tree of aaabbb :



Grammars

General introduction to grammars

- Chomsky made a well-known classification [1].
  - Type 0 $\supset$ Type 1 $\supset$ Type 2 $\supset$ Type 3.
  - Type 0 are the most general grammars, type 3 the more restricted one.
  - Parsing (word recognition) is rather easy for Type 2 & 3, whereas it is PSPACE-complete (PSPACE $\supseteq$ NP) for Type 1 and undecidable in general for Type 0 grammars.

Grammars

What are they used for ?

Grammars

What are they used for ?

- Used for describing/mutate malwares
  - Current work : Polymorphism, Metamorphism
    - Filiol [Filiol07], Zbitskiy [Zbitskiy09], Almeida Lopes [Butkowski09], ..
  - Polymorphism (Zbitskiy) : polymorphic generator based on a formal grammar.
    - Example for mov R1, len
    - X : mov R1, len | push len ⊕ pop R1
      | sub R1, R1 ⊕ add R1, len.
    - Possible to detect (word problem)
    - Language is finite, so only a finite number of words can be generated.

Grammars

What are they used for ?

- Metamorphism :
    - Filiol's definition :

      Let G1 = (N,T,S,R) and G2 = (N',T',S',R') be 2 grammar with T' a set of formal grammars, S' the starting grammar G1 and P' a set of rewriting rules wrt (N'∪T').*
      A metamorphic code is thus described by G2 and all of its mutated forms are words of L(L(G2)).

POC_PBMOT implements this principle. Moreover, it contains a undecidable rewriting system such that the word problem is undecidable in general. (It is undecidable wether 2 words are equivalent up to the rewriting rules)

Grammars

What are they used for ?

- Metamorphism :
    - Almeida Lopes :

      Use of attribute grammar to 'translate' an instruction in equivalent instruction(s) after its parsing was done.

        - Example :
            - Parsing of

              pushl $0x0c popl %edx

              gives non terminal put_v_in_r($0x0c, %edx)
            - Translation rule :

              put_v_in_r(v1,r1) : pushl_v(v1) popl_r(r1)

              | movl_v_r(v1,r1) ;

Grammars

Van Wijngaarden

Grammars

W-grammars

- W-grammar ?
  - Basically, a W-grammar consists of two finite sets of rules :
    - Metaproduction rules (metaproductions)
    - Hyper-rules
  - From these sets of rules, a third (possibly infinite) set of production rules is derived.
  - If the metaproductions describe an infinite language, productions rules will be infinite.

Grammars

W-grammars

- Before we go further, some terminology :
  - We define a « protonotion » as a possibly empty sequence of small syntactic marks (e.g. **int** and **bool**).
  - A « metanotion » is a non-empty sequence of large syntactic marks that is defined in the metaproductions (e.g **LETTERS**).
  - A « hypernotion » is a possibly empty sequence of metanotions and/or protonotions (e.g **int LETTERS**).
  - A « consistent substitution » is the substitution of all the same metanotion throughout a single rule.

Grammars

W-grammars

- Formally, we can define a W-grammar as a 7-tuple : $(M, V, N, T, R_M, R_V, S)$ with :
  - M : a finite set of metanotions
  - V : a finite set of metaterminals $M \cap V = \emptyset$
  - N : a finite set of hypernotions, subset of $(M \cup V)^+$
  - T : a finite set of terminals
  - R_M : a finite set of metarules
  - R_V : a finite set of hyperrules
  - $S \in N$: the start symbol

Grammars

W-grammars

- A little example to make is easier to understand :
  - Language $a^n b^n c^n$ cannot be described by a CFG. The W-grammar for that language is :

$$N \rightarrow i \mid i\,N$$
$$A \rightarrow a \mid b \mid c$$
$$\langle S \rangle \Rightarrow \langle aN \rangle \langle bN \rangle \langle cN \rangle$$
$$\langle Ai \rangle \Rightarrow A$$
$$\langle AiN \rangle \Rightarrow A \langle AN \rangle$$

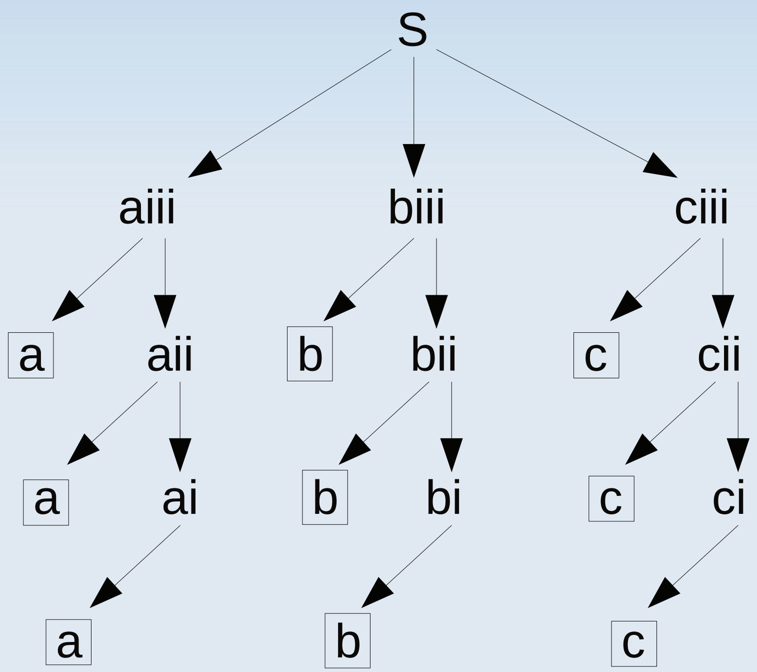
  - For this grammar we have :

$$M = \{N , A\}$$
$$V = \{a , b , c , i\}$$
$$N = \{aN , bN , cN , Ai , AiN , AN , A\}$$
$$T = \{a , b , c\}$$
$$R_M = N_{rule} , A_{rule}$$
$$R_V = \langle S \rangle_{rule} , \langle Ai \rangle_{rule} , \langle AiN \rangle_{rule}$$

Grammars

W-grammars

- A derivation tree for aaabbbccc is :



$$N \rightarrow i \mid i\,N$$
$$A \rightarrow a \mid b \mid c$$
$$\langle S \rangle \Rightarrow \langle aN \rangle \langle bN \rangle \langle cN \rangle$$
$$\langle Ai \rangle \Rightarrow A$$
$$\langle AiN \rangle \Rightarrow A \langle AN \rangle$$

Metamorphism with W-grammar

Word generation

Metamorphism with W-grammar

Word generation

- W-grammar can be used to rewrite instructions into semantically equivalent instructions thanks to consistent substitution.
- In 1984 Dick Grune made a program which produces all sentences from a W-grammar [Grune84].
- When a grammar has a lot of metanotions, generation takes too much time to generate even the first word.
- So the program has been modified in order to produce one random word in the language.

Metamorphism with W-grammar

Word generation

- Simple example of generation :
  - Input instruction : *mov eax, 5* called by vw_start(''mov eax 5'');

```
~/vanWijngaardenGenerator/proj$ ./iawacs
res =    add  esp, -4
         mov dword [esp], 5
         sub dword eax, eax
         add dword eax, [esp]
         add  esp, 4

~/vanWijngaardenGenerator/proj$ ./iawacs
res =    sub  eax, eax
         sub  eax, -5

~/vanWijngaardenGenerator/proj$ ./iawacs
res =    push  5
         pop  eax

~/vanWijngaardenGenerator/proj$ ./iawacs
res =    add  esp, -4
         mov dword [esp], ecx
         sub dword [esp], ecx
         pop  eax
         lea  eax, [eax+5]
~/vanWijngaardenGenerator/proj$
```

Metamorphism with W-grammar

Integration in libthor

Metamorphism with W-grammar

Integration in libthor

- Started to implement it in a libthor module :
  - The grammar is used to generate words.
  - It has no starting symbol : its start is decided by the word given to it.
  - Consistent substitution enables us to « save » some context to keep the semantic of an instruction.

  - The grammar is still relatively simple but can do :
    - Instruction substitution
    - Junk code insertion
    - Basic transformation of control flow
  - Of course, these things are not mutually exclusive.

Metamorphism with W-grammar

Integration in libthor

- An example taken from a libthor execution :
  - A shellcode is read and translated in intel instructions by libthor (it's a multiplication by 2) :

"\x55\x89\xe5\x83\xec\x10\xc7\x45\xfc\x00\x00\x00\x00\xeb\x08\x83\x45\xfc\x02\x83\x6d\x08\x01\x83\x7d\x08\x00\x7f\xf2\x8b\x45\xfc\xc9\xc3"

The instructions are « given » to the grammar which produce a new shellcode from it :

"\x55\x87\x2c\x24\x83\xec\x04\xc7\x04\x24\x00\x00\x00\x00\x5d\x03\x2c\x24\x87\x2c\x24\x31\xed\x03\x2c\x24\x54\x5d\x83\xc4\xfc\x89\x3c\x24\x31\xff\x8d\xbc\x24\x72\xde\xff\xff\x8b\x3c\x24\x83\xec\xfc\x83\xc4\xf0\x83\xc4\xfc\x89\x3c\x24\x87\x3c\x24\x6a\x00\xff\x34\x24\x5f\x83\xc4\x04\x03\x3c\x24\x87\x3c\x24\x31\xff\x03\x3c\x24\x87\x3c\x24\xbf\x00\x00\x00\x00\x03\x3c\x24\x87\x3c\x24\xbf\x00\x00\x00\x00\x03\x3c\x24\xbf\x00\x00\x00\x00\x81\xc7\x96\x08\x00\x00\x29\xff\x03\x3c\x24\x83\xc4\x04\xc7\x45\xfc\x00\x00\x00\x00\xeb\x08\x83\x6d\xfc\xfe\x83\x45\x08\xff\x83\x7d\x08\x00\x7f\xf2\x83\xec\x04\x89\x1c\x24\xbb\x00\x00\x00\x00\x8d\x1c\x24\x5b\x29\xc0\x03\x45\xfc\xc9\xc3"

When the shellcode is executed, we obtain the right result :

```
int main(int argc, char *argv[])
{
int ret = test_exec(shellcode, 3);
printf("RES = %d\n", ret);
return 0;
}
```

```
~/libthor/metamorphism$ ./test
RES = 6
~/libthor/metamorphism$
```

K-ary codes

What are they ?

K-ary codes

What are K-ary codes ?

- What is a K-ary code ?
  - Main idea [Filiol07]: A k-ary virus is a set of k files (some of which may not be executable) whose union consitutes a virus.
  - These codes have been categorized in 2 classes, each of them having 3 subclasses :
    - Class 1 : sequential execution
      - Class A : Every part contains a reference to the others.
      - Class B : No part is refering to another one.
      - Class C : Dependency between code is partial and directed only.
    - Class 2 : parallel execution
      - Same subclasses

K-ary codes

Representation by a W-grammar

K-ary codes

Representation by a W-grammar

- Definition from a « grammar point of view » :
  - Let $x_{1,} x_2$ be 2 files and v a virus described by a grammar G_v , we define a relation $\mathcal{R}_v$:

$$x_i \mathcal{R}_v x_j \Leftrightarrow \{x_i \oplus x_j\} \in \mathcal{L}(G_v)$$

- Such virus can be described by a W-grammar :
  - A W-grammar is capable of handling the semantics of a language/program.
  - Each part of the virus may be described by a grammar. If we put them together in a rule, the consistent substitution allow us to keep a track of some informations between each parts.

K-ary codes

Representation by a W-grammar

- A dummy example (Class 1 – B):
  - We want a virus to delete files named 'example' :
    - ALPHA :: a; b; c; .... ; z.
    - LETTERS :: ALPHA; ALPHA LETTER.
    - TEMP :: LETTERS ~.
    - FILE :: example.
    - S : Program which rename FILE into TEMP, Program which place TEMP file in trash, Program which empty trash.
    - Program which rename FILE into TEMP : grammar1
    - Program which place TEMP file in trash : grammar2
    - Program which empty trash : grammar3

Conclusion

- Grammar are powerful tools to manipulate languages and so programs.
- W-grammars, by the use of two CFG, allow us to describe quite easily type 0 languages.
- The word decision problem for this type of languages is known to be undecidable.
- Thanks to its integration into it, libthor provides us a working framework to test code metamorphism.

- This is work in progress.. a lot more can be done and will, eventually.
- Any questions ?

References

Chomsky, N. (1956) : Three models for the description of languages, *IRE Transactions on Information Theory,* 2, 113-124.

Filiol, E. (2007) : Metamorphism, formal grammars and undecidable code mutation, *Proceedings of World Academy of Science, Engineering and Technology (PWASET),* Vol.20

Filiol, E. (2007) : Formalisation and implementation aspects of K-ary (malicious) codes, *Journal in Computer Virology,* Vol. 3, No. 2, p.75-86, June

Zbitskiy P.V. (2009) : Code mutation techniques by means of formal grammars and automatons, *Journal in Computer Virology,* Vol.5, No.3, p.199-207, August

Almeida Lopes, A : The development of an offensive code framework, http://bukowski-framework.blogspot.com

Grune D. (1984), How to Produce All Sentences From a Two-level Grammar, *Information Processing Letters,* 19, p 181-185.