Tutorials - Polymorphism and Grammars

ivanlef0u.fr/repo/madchat/vxdevl/vdat/tupolgra.htm

Polymorphism and Grammars

By Qozah
qozah@hax0r.com.ar

Objectives in this article

This is a technical article about polymorphism and grammar/automaton theories, which is intended to give a new point of view about this virus technique, and to show you some things on polymorphism theory. Before you start this, keep in mind it's not a "begginer article". If you never did a polymorphic engine or you don't know what the hell it is, you may come here later.

Maybe some of the article's points look obscure for you, but I'm putting all my effort to avoid that from happening.

INDEX:

Defining some stuff

You've got to have clear in mind some concepts:

- Alphabet: Non-empty finite set of graphic symbols.
- Word: Word from an alphabet is an ordered and finite sequence of alphabet elements.
- Universal language: Set containing all the possible words that can be made from alphabet elements.
- Language: Subset of the universal language.

Formal grammars

A grammar is an algorithm that describes us all the elements in a language. That is, a generative description of a lenguage, which describes it's elements and generates them, giving us the set of rules that will help us distinguish if a word belongs to a language or not.

It has for different elements, this way: G = (Et,En,S,P).

```
Et: Set of ending symbols ( main alphabet )
En: Set of not ending symbols ( auxiliar alphabet )
S: Initial symbol / axiom
P: Set of productions. Starting from it, you can write all the
   language words.
```

Let's explain this with an example. Imagine you have a little alphabet made of {0,1} elements. Let's imagine the language you want to make is made from the words that have at least one zero. We can represent that this way:

```
(0+1)* 0 (0+1)*
```

In the beggining there's a (0+1)*. 0+1 means an OR, that is, you can select 0 or 1 as you wish. Then, the '*' after the parenthesis means that you can repeat that as long as you want. For example, that single stuff could generate something like "1001101101011011011...". Then, we have a 0. This means we MUST put a zero. And then again we can generate all the elements we want, be it 0 or 1. This assures that, even if the (0+1)* is empty in the two cases, there is at least one zero. An example on this would be '11101'. That is a word that belongs to the language because it has at least one zero. Of course you can imagine the "(0+1)" in the beggining generated three '1's, then there was the zero, and the last element made a "1" for us.

Now we are going to play with another formalization that will be more useful for our objectives, which is called "productions", and which is what the "P" element in the grammar repressents.

A production is always this way: " x ::= y ", where we have terminal and non-terminal symbols (go back to the grammar definition). The first symbol in our production will have to be the "S" we said also in the grammar.

Let's go to an example with the same language (at least one zero) I gave before. The ending symbols will be {0,1} (as I said, they are the main alphabet symbols), and the non-ending ones will be called always A,B,C...

```
S ::= 1S/0S/A
A ::= 0B
B ::= 1B/0B/Lambda
```

This new notation is the notation production. The initial symbol (S) produces either "1S","0S" or "A". 1 and 0 are terminal symbols, while S makes it act recursively on itself. "A" symbol makes it go to the "A" production. Finally, Lambda means "generate nothing", and is really an alphabet symbol that means "no symbol".

To make you understand, I'll generate the word 011010 with this production set. First of all I define the grammar, then I make the productions to get this word. G = {(0,1),(A,B),S,P}

```
What I'm doing          Word I've got
S -> 0S                     0S
S -> 1S                     01S
S -> 1S                     011S
S -> A                      011A
A -> 0B                     0110B        Here we generate the zero needed
B -> 1B                     01101B
B -> 0B                     011010B
B -> Lambda                 011010
```

I hope you understood with this. Only thing I've got to say is that you can make productions of these kinds in example, with lower-case letters as terminal symbols and upper-case ones as non-terminal ones

```
A ::= aB
A ::= Ba            You can extend by left or right
Ab ::= a            This sets a condition for it, as it should be Ab
A ::= cabB/cBa/Lambda
[etc]
```

What about polymorphism ?

I was going to give some more theory, but it's enough by now to make a direct relationship among polymorphism and grammars.

The alphabet we're going to talk in this section of the article is the x86 PC instructions set. It has a very large number of elements, but a finite one. Also it's true that that instructions have lots of internal rules, addressing, etc etc that makes "subinstructions" inside instructions. I mean that it's not the same a mov [eax],esi than a mov [eax+125],ebx, but to simplify all this, we'll treat them as equivalent instructions.

So, we have an alphabet composed on this large number of instructions. Where you make a polymorphic language, you make a garbage generator which doesn't generate ALL the instructions in the PC, but a set of them.

Now imagine your garbage generator can make four instructions, 'a', 'b', 'c' and 'd'. Then, you also need to generate two instructions to make the real virus decryptor, which will be called 'x' and 'z'. A normal polymorphic engine works this way:

```
G = {(a,b,c,d,x,z),(A,B),S,P}

  {  S ::= aS/bS/cS/dS/xA
P {  A ::= aA/bA/cA/dA/zB
  {  B ::= aB/bB/cB/dB/Lambda
```

It's easy to understand this production. The garbage generator makes an undefinite number of garbage instructions (a,b,c,d). Then, it generates "xA". So, then it will generate another undefinite number of garbage instructions till it generates "zB", then in B it generates some more garbage and finally goes to "Lambda", that is, finishing.

So, a poly engine is really a grammar that generates instructions according to the production it has, limited to it's alphabet. Depending on how complex the alphabet is, you could generate a wider number of words.

In this case, the language the polymorphic engine generates is formed by the words that are:

```
    (a+b+c+d)* x (a+b+c+d)* z (a+b+c+d)*
```

This is garbage + decryptor instruction + garbage + decrinstr + garbage. All the words that can be generated by this algorithm form the language of the polymorphic engine; for example, "abcddxd" isn't in this language as that word is not include in the polymorphic generator language.

Bad stuff: Automatons

If we stare at what I said, every polymorphic engine can be detected, and I'll show by now why. I'm anyway forgetting that you can make the code look as real code etc, but taking a decryptor this way, it can always be detected. For that, I'm introducing you our new friend, that is the automaton.

An automaton can be defined just as a bunch of states or situations with links among them which depend on conditions. Nothing better as an example:

```
                a
         .---.---->.---.
    --->|A/1|     |B/0|
         '---'<----'---'
                b
```

We start in state 'A' as the initial arrow tells us. When we are in state A, our automaton gives us a "1" as result. If out automaton receives the alphabet sign called "a", we go to the state "B", which generates the "0" result and so on.

Now that you've got the main idea on how an automaton works, I'm going into the kind of them we're going to use when talking about polymorphism detection.

This automaton kind is called Finite Determinist Automaton (FDA). The results it generates can only be 0 or 1, which is indicated not in the state but with a double square around it (though it's circles and not squares in the normal representation). The states that have the double square are called "terminal states", and the single square ones are "non-terminal states".

An automaton can be made so it can detect our polymorphic engine, as the automaton's work is recognizing which words belong to a determinated language. For example this automaton detects when does it detect the first language we used, the "there is at least one zero" language.

```
          .---.  0   .===.
   ------>| A |----->| B |
        .>'+--'    .>"==="
        |  |       |   |
        '--'       '---'
         (1)       (0,1)
```

The automaton starts on the "A" state, and can generate "1" symbols going to itself again. As soon as it generates a "0" symbol, it goes to the "B" state, which is a terminal state. As I said above, as A is non/terminal it will always return "0", and B will always return "1" as it is terminal.

There's another more accurate way to explain this. The automaton is trying to recognize a word that is in the language that tells us it's words are "all of them which have a zero". So, in the state "A", the automaton will ignore all ones and give us an answer that is "0", which means that it hasn't still recognize that it's a word. As soon as a 0 is sent to the automaton, it will change to "B" which is a terminal state, because the word we sent it DOES have a 0. So, the automaton will return us a "1", meaning that it recognized the word. Of course, if we send more 1 or 0 signals to the automaton, it will remain in the "B" state, because it did recognize the word.

Now this is an automaton that will detect our polymorphic engine. You should remember it was made this way:

```
     {  S ::= aS/bS/cS/dS/xA
   P {  A ::= aA/bA/cA/dA/zB
     {  B ::= aB/bB/cB/dB/Lambda
```

An automaton detecting this language would be as easy as this:

```
      .---.  x    .---.  z   .===.---.(a,b,c,d)
   --->| S |----->| A |----->| B |<--'
     .>'+--'    .>'+--'      "==="
     |  |       |  |
     '--'       '--'
  (a,b,c,d)   (a,b,c,d)
```

You can easily figure what this automaton is doing. Ignoring any (a,b,c,d) instructions, it will go to state "A" when it finds the "x" instruction, then to "B" when it finds a "z". When it reaches the "B" state, the word is recognized into the language, the polymorphic engine language, returning a "1": in other words, the virus is detected.

Sadly, this is demostrating us that roughly every polymorphic engine can be detected by just using simple authomatons; in Appendix B I give an algorithm to accomplish this. You can make an automaton for each language you form with regular grammars, that is, something that recognizes that language (every decryptor generation). So, roughly every poly engine can be detected if we are inside this rules.

No fuckin' chance

PC 80x86 instructions are a really big alphabet, having in mind that each "alphabet symbol" inside it as I defined it earlier has other symbols, which are the different implementations on the same instruction, depending on which registers or inmediate data they use. Anyway, it's a language, and every language made out from this alphabet can be detected by an automaton.

Seems we're condemned to death, and that there is no fuckin chance on making undetectable any polymorphic engine. That's true: even if we change all the garbage instructions a lot, even if we can use all the x86 range in there (which is not possible) we'll generate the polymorphic decryption instructions within a limited number of signs that belong to the 80x86 instruction set library.

That is our main problem. The automaton knows the alphabet you use for making garbage and recognizes it, doing so with the decrypting alphabet which is so smaller. Even if you could make the alphabet of the garbage generator, you won't solute the main problem, that is recognizing the decryptor instructions.

Maybe this last sentence sounds strange to you. "Well, really if it's all the alphabet or nearly all the alphabet, it could also generate decrypting instructions alike, so it won't detect my virus.". That's sadly false. There are another kind of automatons called Non-Determinist Finite Automatons (NDFAs), which just means that you can explore many ways on them at the time, and that giving it an entry doesn't keep you in a determinate state; anyway they can be easily translated by an algorithm to DFAs.

If you had (a,b,c,d,x,z) as garbage instructions in the test generator I talked about above, the state S would go to A by means of x, but it would go to S by means of (a,b,c,d,x,z). Just think about not beeing necessary to be kept in a state, but decide depending on the consequent entries. This is called NDFA, is an automaton, and sadly, is completely computable.

The second chance here is making x and z be a collection of the whole alphabet which doesn't follow any pattern, but this is almost impossible: also, real code follow patterns (you won't see mov ax,bx + mov ax,1200 although the coder is so dumb) which makes it much more difficult (but in the other hand it's also difficult to find all those patterns).

So, x and z, that is, our polymorphic decryption instructions, seem to be the key for all this stuff. The most logical solution to avoid an automaton to work needs two requirements:

```
 + x and z should contain the universal language. This means it has to
be able generate any kind of instructions.
 + x and z shouldn't follow a pattern.
```

Our problem is that x and z MUST decrypt the virus, so we're following a pattern. Even if we could have any kind of instructions, which is almost impossible, we would work with a pattern that is intended to decrypt the virus.

Conclussions. Are we condemned to death ?

I'm not saying polymorphism as it's done by now is an useless task: you make antivirus specialists work on discovering x, z, and your pattern, and you can even make it so difficult some AV dumbs won't even care on trying to decrypt your virus, except they can emulate it or make cryptanalysis in your code, of course.

The fact is that we have to invent different ways on polymorphism. Some new stuff is better, but still undetectable - and will never be -, as metamorphism technique is.

> **Exploring other methods**

My UVE (Uncleanable Virus Engine), though is poorly developed looking as uses just one instruction to make confussion, shows another way to avoid this time cleaning completely. Mixing legit instructions with garbage ones in a way the antivirus can't work out which one is the good one, makes it impossible a good cleaning; in terms related to this article, mixing legit instructions and virus ones making them from the same alphabet, has the consecuence of non-beeing able to distinguish legit from viric, so clean can be made impossible.

But maybe you've noticed I'm trying to distract you. This article was meant to talk about virus detection and avoiding it by polymorphism, and I am probably supposed to give you a solution: the only way you maybe think we could go near it is by non-regular grammars, but I'll have to anounce you that way is also blocked; explained in Appendix A. Anyway, we can still work in an uncleanable virus, mixing as I said legit code and garbage code so the antivirus can't difference among them.

The best advice I can just provide to try avoid detection into a poly, is that you include the decrypting instructions in the garbage generator, so the detecting automaton has difficulties making such a difference. In example, keep in your virus some host data and then operate the memory data segment as well as you do with your encrypted virus; but then again, this way doesn't seem completely useful, as a NFDA can easily avoid this

Appendix A: Non-regular grammars

The grammars I was talking above were always regular grammars, which have two kinds of productions: left and right linear. Left linear ones are when all the productions are like A ::= Aa, thas is, in "left" direction. Right linear is the opposite.

As you can see, most polymorphic engines can be reduced to a regular grammar when generating virus decryptors. You generate in ex. garbage this way; A ::= aA/B, where a represents garbage. When you don't want more, you just come to B, which i.ex is B ::= bC, be it 'b' a decryptor instruction. C, would be like A, and so we have a right linear, regular grammar.

But there are some different grammars which we call "non-regular". I would resume the kind of productions this way (call regular grammars type 3):

- Type 0 grammars: they're productions like u::=v (let the capitals be the non-ending symbols on the production); they contain types 1, 2 and 3 grammars (regular grammars are type 3).
- Type 1 grammars: called context sensitive grammars, when they're conditioned as xAy ::= xvy, like as habing a "precondition".
- Type 2 grammars: non-context sensitive, be it as A ::= xBy etc.

The good thing about this grammars is that they sometimes can't be translated to automatons. This could leave us a way into future developing although a limited decrypting alphabet restricts polymorphism as beeing detectable; you could think that using that kind of grammars (which cannot be converted to automatons) detection is impossible, or at least if they couldn't be converted to regular grammars, detection would be really difficult.

But again to dissapoint you, there are ways - different to automatons and maybe a bit more complicated in concept - to detect such words generated by even a type 0 grammar (which contains the other three). Stack automatons can detect types 2 and 3 of grammars, and finally Turing machines, for which there's not enough space to explain, can detect any kind of grammar, so they could detect any polymorphic engine regardless on which generation system it has on making their "words", that is, regardless on how did you make your decryptor.

Appendix B: Converting a production into an automaton

In this last appendix you'll know how to convert a left linear grammar into an automaton, deterministic or not. There are some simple rules that will make this an easy job.

Just take a left linear grammar (LLG) formed by Et, En, S and P (search line 54). The automaton will have as many states as non-ending symbols in the grammar plus two more.

Let's look at an example:

```
        { S ::= U0/V1
    P = { U ::= S1/1
        { V ::= S0/0
```

Now look at this as if S was a function on U with 0. Now we should write, f(U,0) = 0. Let's do that with every single production:

```
    f(U,0) = S
    f(U,1) = q
    f(V,0) = q
    f(V,1) = S
    f(S,0) = V
    f(S,1) = U
    f(p,1) = U
    f(p,0) = V
    f(q,0) = q
    f(q,1) = q
```

Most of this is clear, but you should wonder what p and q are; p is used (as another state) when there's no non-ending letter from the non-ending alphabet in the function (that is, 1 or 0 are alone). q, is empty state, that is when there's no function on that one; acts in the automaton telling the word isn't of the language.

So, here is the automaton (in a chart):

```
    f | 0 | 1
   ---+---+---
      |   |
   *S*| V | U       <= Ending state
      |   |
    U | S | q
      |   |
    V | q | S
      |   |
 -> p | V | U       <= Beggining state
      |   |
    q | q | q       <= Non-word state
      |   |
```

Another thing; p will be the initial state and the final one in the language (that is S) will be a final state (when the automaton reaches there, a complete alphabet word has been red). Q is another ending one, but meaning there isn't a word from that alphabet; in our case, tells us nothing was detected.

So let's make a try. I generated by that language the word 01101010 - and remember I generated it right to left. The grammar used was the one pasted above, where I made the automaton from: words look always as combinations of 2 long terms that are "01" or "10", never a 11 or a 00.

Now looking at the chart, starting from p we have a 0. That is the last we generated, as it's done left to right. Now beeing it a 0, we go to the "V" state in the automaton (just think each line in the table is a state). Now in the V, we do generate a 1 so we turn to the S state. Then the next component of the word is a 1 so the new state is U. Now we find a 0 so we get to S and so on. When the last 0 is red from the word, we finish in state S; that is, a *final state*. That means it's the state where our automaton recognizes that the word we sent him (be it a poly decryptor in example) is what it searched for.

Keep in mind also that if it turned to the "q" state, there's no possibility; the word isn't from the language we tried to recognize and can't be generated by the grammar.

I don't like to finish a depressing article ;) like this telling you there's no chance, but there you had that simple rules to recognize almost every polymorphic engine, as *every* regular grammar's words can be detected by a simple automaton specifically made to detect it.

Written by Qozah, Finland 99'
