Tutorials - Generic Polymorphism

Generic Polymorphism
by Rajaat

Introduction:

The idea of having an opcode emulator in a virus scanner to decrypt a very polymorphic virus isn't a novelty anymore. The more decent scanners can handle viruses that have a complex decryptor, using a generic decryptor to take the protective shell from the virus away, making the structures of the virus very visible to the scanner.

So far, every polymorphic engine that is designed is able to make a certain type of decryptor, with variations in it's general behaviour, but not a generic encryptor.

Since we, virus writers, also must adapt to newer programming technologies, a further step in virus advancement is generic programming. This generic programming can be used for any sort of virus related approach.

Background:

Every polymorphic engine is based on certain rules that will be used in a random order and trash instructions and instructions that compose another opcode. But what about a polymorphic engine you can feed a piece of CODE to, and the engine generates mutations of that code? This would truely be a real polymorphic engine and will polymorph everything, regardless of what it does. Take this for an example:

```
        MOV AX,3D02
        INT 21
```

If you feed this to the generic polymorpher, you might get the following result:

```
        SUB AX,AX
        ADD AX,3000
        CMP AX,78AF
        JE NEVER_JUMPS
        XOR AX,0D02
NEVER_JUMPS:
        INT 21
```

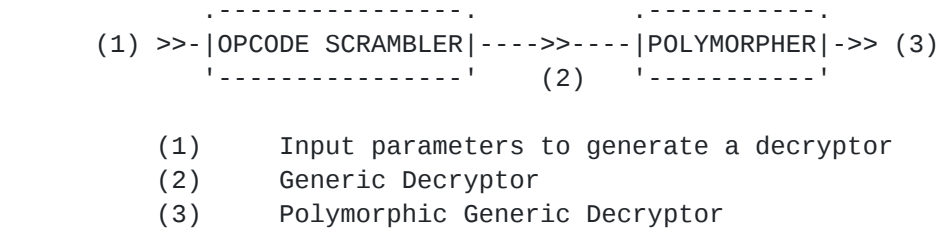
Or if you want to generate a decryptor, you can feed this to the polymorpher:

```
        MOV SI,OFFSET ENCRYPTED_VIRUS
        MOV AL,32
        MOV CX,VIRUS_LENGTH
  DECRYPT_VIRUS:
        XOR BYTE PTR [SI],AL
        ADD AL,4
        INC SI
        LOOP DECRYPT_VIRUS
```

You might get this as a result:

```
        MOV SI,OFFSET ENCRYPTED_VIRUS+145
        ADD SI,-145
        MOV AL,33
        DEC AX
        MOV CX,VIRUS_LENGTH XOR 1234
        XOR CX,1234
  DECRYPT_VIRUS:
        XCHG SI,DI
        XOR BYTE PTR [DI],AL
        XCHG SI,DI
        INC AL
        INC AL
        INC AL
        ADD AL,1
        SUB SI,-1
        DEC CX
        JNZ DECRYPT_VIRUS
```

As you have noticed, the polymorpher only puts random instructions in the place where it could compose more compact instructions, it doesn't swap opcodes. This should be generated by a separate module, because a the polymorpher doesn't know what it is allowed to swap and what's not allowed to swap. To generate a decryptor, you must call two engines, like in the diagram shown:




```
        (1)     Input parameters to generate a decryptor
        (2)     Generic Decryptor
        (3)     Polymorphic Generic Decryptor
```

You can make both the opcode scrambler and the polymorpher as complex as you want. In it's most simple implementation, the polymorpher has a little drawback: since it polymorphs one opcode at a time, the random opcodes and not be mixed with random opcodes from the next operand. The advantage is that you don't need to track the registers, but it's polymorphic structure is still somewhat predictable. There are a few problems considering the polymorpher:

- Recalculate branch jump opcodes

- How to handle registers set to pointer
- How to fully mutate a virus? You must keep the original metavirus code, otherwise it would polymorph the polymorphed virus.
- How to set aside data from code while morphing?

An approach to resolve the jump and register handling opcodes is making some sort of relocation table, as used in EXE files. If you want to polymorph a piece of program you feed the polymorpher the range of jumps it should adjust. You can put this on the stack while calling the engine, or setting some pointer to a predefined table, whichever approach you like most. Make sure you can relocate both 8 and 16 bits addresses, both direct or indirect.

You can also generate a range table which must not be morphed and to which references should be updates (pointer registers).

A relocation/exclusion/resolve table (RER table) can look like this:

```
enum RER_types {
  RERT_nil,           ; 0 - end of RER chain
  RERT_8rel,          ; 1 - 8 bits relative address (conditional jumps)
  RERT_16rel,         ; 2 - 16 bits relative address (call)
  RERT_16dir,         ; 3 - 16 bits direct address (pointers & jumps)
}
struct RER_table {
  RER_type    : unsigned char;
  RER_address : unsigned int;
}
```

After using this table on morphing the code you want, don't forget to update the table with the new addresses if you want to morph again the current result. You can, if programmed in a good style, call the morpher multiple times, and each time it will morph the routine specified.

A bit harder is including the possibility to shrink a routine that was generated by a previous morph. To expand and add instructions that do the same thing is easier than shrink them into a compact instruction, as you need to know how a few instructions can be optimized into less instructions or a shorter variant. Some simple algorithms are possible, like changing an ADD <reg>,1 into an INC <reg> or even more complicated things like this:

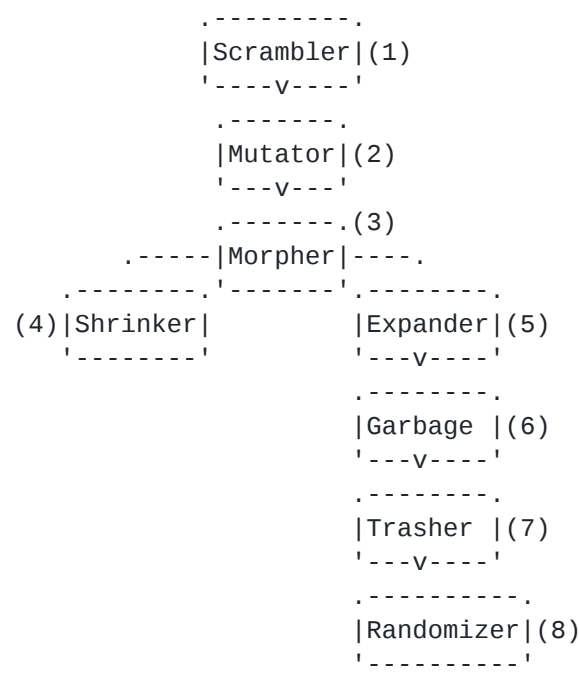
```
Unoptimized                                    Optimized
CMC                                            JC ERROR
JNC ERROR
CMC

XOR AX,AX                                      MOV AX,3D02
ADD AX,3D02
```

Maybe it's best to seperate the shrinker module too, so you first let the shrinker optimize the virus to what it thinks that is the shortest way, and next calling the morphing engine to enlarge it again. Both can use the same instruction table and RER table. You can also add an

overlapping module that randomly calls both the shrinker and morpher, thus optimizing some parts of the virus while expanding other parts.

Thus, the modules are like this hierarchy:




```
(1) Will put instructions in a random order
(2) Will choose other registers for the actions that must be done
(3) Will call randomly the shrinker and expander to mutate the code
(4) Shrinks code by optimizing instructions that it knows
(5) Expands code by creating multiple random opcodes with the same
    functionality
(6) Adds random do-nothing instructions
(7) Adds plain garbage (random numbers)
(8) Random number generator, must to able to do both slow-random and
    regular random (for slow and normal poly/mutating)
```

Rajaat 29A, November '98
