Metamorphic Viruses

Sean O'Toole

Metamorphism Defined

- “Body-polymorphics” (Szor, 127)
- “Self Mutating Code” (Lord Julius)
- “The Art of Extreme Mutation” (Mental Drill)

Brief Description of Sections

- 1) Disassembler: used to disassemble host, most often into a linked list of op codes.
- 2) Depermutater: removes some of the jumps added by the permutater, and sometime also from the host, and therefore removes unreachable code.
- 3) Shrinker: changes op code clusters to most efficient op code.

Brief Description of Sections

- 4) Expander: randomly chooses code to change to equivalent op code or op code cluster.
- 5) Permutater: randomly “shuffles” groups of code and links the groups with JMPs.
- 6) Assembler: re-assembles code at the end of the infection process.

Metamorphic Programming Approach

"Do not think in code think in macros" (Mental Drill).

In other words, the best approach to this process it to approach it with software engineering in mind. Everything is a separate/independent module or macro.

1) Disassembler (Choices)

- Using a pseudo-language, which was the idea presented in “Metamorphism in Practice” by Mental Drill.
- Using a reverse-engineering tool, such as LDE (Length Disassembly Engine) and ADE (Advanced Disassembly Engine) by Zombie. To use these please check the manuals that accompany the module.

1.1) “Pseudo” Code Ex. by Mental Drill

Op Code Structure:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
OP *-------- instruction data -------*  LM *-pointer-*
```

Op Coding Ex. (Full List in Article):

MOV:= 40; Reg, Mem:= +2;

So, MOV Reg, Mem := 42

2) Depermutator (Choices)

- Integrate depermutator into the disassembly process. (This is the most common.)
- Create separate module for depermutator.

2.1) Pseudo-Code

Variables:

- ESI = Entrypoint.
- PathMarks: buffer that will contain the depermutated virus.
- LabelTable: list of elements that are each two DWORDs long. The first DWORD stores the real EIP where it points; the second stores a pointer to the depermutated code.
- FutureLableTable: list that contains pointers to the destinations of JMPs, CALLs, etc. that have not yet been depermutated. Each element is a DWORD.

2.1) Pseudo-Code

Initializations:

1) Initialize the PathMarks map (i.e. zeroing it) and the number of labels and future labels.
2) Translate the current EIP (in ESI) directly onto the PathMarks map.

2.1) Pseudo-Code

If it's JMP:

- If it points to an already depermutated address, write a JMP instruction, insert a label to the destiny and get a new EIP at FutureLabelTable. If the label already exists, use that label.
- If not, then write a NOP (just in case a label points directly to this JMP) and load a new EIP (in ESI) with the destination. In this way, we have eliminated a possible permutation JMP.

2.1) Pseudo-Code

If it's Jcc (conditional jump):

- * If it points to an already depermutated address, write the Jcc and insert a label to the destiny if the label doesn't exist (if not, use the label already inserted in the table).
- * If the destiny is not depermutated yet, then store it at FutureLabelTable and continue.

2.1) Pseudo-Code

If it's CALL, act as if it were a Jcc.

If it's RET, JMP Reg or JMP [Mem] (a final leaf in the code tree), store the instruction and get a new EIP from FutureLabelTable.

2.1) Pseudo-Code

Note:

When getting a new EIP from the FutureLabelTable, we check if the labels stored here are already depermutated. If they are, then we insert the corresponding labels at the LabelTable and eliminate the entry in FutureLabelTable. If not, we get that new EIP (i.e. we load ESI with that new entrypoint), we insert the new label at LabelTable and continue.

2.1) Pseudo-Code

If creating a depermutater as a stand-alone module. The Psuedo-Code is the same, except that when depermuating a jump the pointers in the elements of the list are manipulated instead.

2.2.1) Example (permutated code)

```
      xxx1
      xxx2
      xxx3
      jmp @A
      yyy1
      yyy2
@B:   xxx4
      xxx5
```

```
      xxx6
      jmp @C
      yyy3
      yyy4
@A:   xxx7
      xxx8
      xxx9
      jmp @B
```

```
@D:   xxx13
      xxx14
      RET
      yyy5
@C:   xxx10
      xxx11
      jz  @D
      xxx12
      RET
```

2.2.2) Example (depermutated code)

```
     xxx1
     xxx2
     xxx3
     xxx7
     xxx8
     xxx9
     xxx4
     xxx5
     xxx6
```

```
     xxx10
     xxx11
     jz  @D
     xxx12
     RET
@D:  xxx13
     xxx14
     RET
```

3)Shrinker

The shrinker is pretty much a stand alone module.

The only possible relation is if the expander uses a list to find code and choose an equivalent, the list can be reversed to find the shrunken equivalent to a cluster of op code.

3.1) Pseudo-Code

```
CurrentPointer = FirstInstruction
@@Loop:
if([CurrentPointer] == MATCHING_SINGLE){
    Convert it
    if (CurrentPointer != FirstInstruction) call DecreasePointer
    if (CurrentPointer != FirstInstruction) call DecreasePointer
    if (CurrentPointer != FirstInstruction) call DecreasePointer
    goto @@Loop
    }
 if ([CurrentPointer] == MATCHING_PAIR) {
     Convert it
    if (CurrentPointer != FirstInstruction) call DecreasePointer
    if (CurrentPointer != FirstInstruction) call DecreasePointer
    if (CurrentPointer != FirstInstruction) call DecreasePointer
    goto @@Loop
}
if([CurrentPointer] ==MATCHING_TRIPLET){
     Convert it
    if (CurrentPointer != FirstInstruction) call DecreasePointer
    if (CurrentPointer != FirstInstruction) call DecreasePointer
    if (CurrentPointer != FirstInstruction) call DecreasePointer
    goto @@Loop
}
 do (CurrentPointer++) while
    ([CurrentPointer] == NOP)
if(CurrentPointer != LastInstruction) goto
    @@Loop
 DecreasePointer: do (CurrentPointer--)
    while (([CurrentPointer] == NOP) &&
    ([CurrentPointer.Label == FALSE))
    return
```

4) Expander

The expander is most often it’s own module, with the only possible relationship being the previously mentioned relationship with the shrinker.

4.1) Pseudo-Code

```
CurrentPointer = FirstInstruction
AmountExpanded = 0
While(NotEndOfCode) {
    boolean isExpandable = Expandable ([CurrentPointer])
    RandomNum = 0
    if(0<isExpandable<4) RandomNum = random() % 2
    else if(3<isExpandable<7) RandomNum = (random() % 4) - 1
    else if(isExpandable == 7) RandomNum = (random() % 6) – 2
    if(RandomNum <= 0 || AmountExpanded == EXPANDEDENOUGH)
    { IncrementPointer(CurrentPointer) AmountExpanded = 0 }
    else { replace(CurrentPointer, expandOp([CurrentPointer], isExpandable, RandomNum)) Increment(AmountExpanded) }
}
```

4.1) Pseudo-Code (Notes)

Expandable checks if the current operation can be expanded to a larger amount of code. Return Values:

0: not expandable

1: can expand to one operation

2: can expand to two operations

3: can expand to three operations

4: can expand to one or two operations

5: can expand to one or three operations

6: can expand to two or three operations

7: can expand to one, two, or three operations

4.1) Pseudo-Code (Notes)

Function replace: replaces the instruction pointed at by argument one with the instruction(s) in argument two.

Function expandOp: expands the operation pointed at by current pointer with instructions based on the values of the second two arguments (isExpandable, RandomNum) Possible Arguments:

exchange current operation with the one listed: (1,1),(4,1),(5,1), or (7,1)

expand current operation to the two mentioned in the list: (2,1),(4,2),(6,1), or (7,2)

expand current operation to the three mentioned in the list: (3,1),(5,2),(6,1), or (7,3)

5) Permutater

Permutaters are self-contained modules, which can be seen in the Win32/Ghost and Win95/Zperm.

5.1.1) Pseudo-Code (Shuffling Code)

```
ESI = Initial address of instructions
EDI =Buffer for code
NumDivisions = User Sets
GroupSizes = (EDI-ESI)/NumDivisions
List = zeroed array size NumDivisions
for(c=0; c<NumDivisions; c++)
{
Redo:  randomNum = (rand() % NumDivisions)
        if(Unique(List, randomNum){
            List[c] = randomNum

            CopyGroup(
            (randomNum * GroupSize) + ESI,
             GroupSize, [EDI])

            EDI +=GroupSize
            [EDI] = insert jmp for later
            EDI += jmpSize
            }
        else        goto redo
}
```

5.1.2) Pseudo-Code (Adding JMPs)

```
EDI = Reset to Beginning of Buffer
IndexNum = find(list, 0)
EntryPoint = (IndexNum * GroupSize) + (IndexNum*jmpSize) + ESI
JmpPoint = EntryPoint + GroupSize
For(c=1; c<NumDivisions; c++)
{
    IndexNum = find(list, c)
    DestinationPoint = (IndexNum * GroupSize) +
            (IndexNum*jmpSize) + ESI
    [EDI+JmpPoint] = Jump to DestinationPoint
    JmpPoint = DestinationPoint + GroupSize
}
```

6) Assembler

In the assembler, the most prevalent problem that needs to be fixed during assembly is jump relocation.

6.1) Pseudo-Code

Initialized:

- eip_table (8 bytes per entry):
    - new_eip
    - old_eip (+4)
- jmp_table (4 bytes per entry):
    - ofset of referenced instructions.

6.1) Pseudo-Code

```
for (int y=0; still jumps to process; y++)
    for(int x=0; not end of jmp_table; x++)
        if( jmp_table[x*4] == eip_table[(y*8)+4])
            assign jump eip_table[y*8]
        endif
    endfor
endfor
```

7) Other Ideas

- Register Exchange (EBX becomes ECX), like AZCME32 by Zombie and W9x/Regswap by Vecna.
- Entry Point Obscuring (EPO) & Unknown Entry Point (UEP) Techniques
- Integrating other modules, such as garbage code generators and encryption, into metamorphism.

8) How Does This Technique Defeat AV Techniques?

The article “Zmist Opportunities” in Virus Bulletin March 2001 the authors stated, “Metamorphic creations will come very close to the concept of a theoretically undetectable virus.”

Zmist will be used as the test for the AV techniques.

8.1) Zombie's Ideas on Undetectable Viruses (29A #6).

- Variables:

  C := complexity of checking file for some virus.

  C[i] := complexity, caused by metamorphism, polymorphism, etc., of checking file for some virus at a specific address.

  I := number of possible addresses in file where execution of virus body (or part of this body could) start.

- Formula:

  C = C[i] * I.

8.2) Signature Scan

- Obviously, by having a continually variable body generation to generation, there will be no signature, unless the virus purposely leaves an unambiguous sign to mark already infected files.
- Zmist places a ‘Z’ at offset 0X1Ch as an infection mark.
- If an ambiguous marker is used false positives will be encountered by the virus and AV, if it uses it as a signature.

8.3) Geometric Scanning

- The Zmist virus causes at least a 32KB increase in the virtual size of the data section.
- If a geometric scanner, which looks for size changes, the scanner will often give false positives since this action is extremely similar to the actions of a runtime-compressed file.

8.4) Possible Answer

- 1) A combination of techniques that as a whole could be used in a heuristic concept.
- 2) Use of a shrinker so that a skeleton can be traced through and signature scanned.
- The flaws in these answers:
  - This would become very time consuming and therefore would often not be used by the public, as pointed out on many occasion by Ferrie & Szor.
  - Techniques that require emulation can be blocked by the latest anti-emulation technique being integrated into the code.

9) Why This Technique Is Superior To Previous Techniques:

- Trash generation: This technique will cause a constant growth in code size until the virus becomes to large and obvious.
- Polymorphism: The majority of polymorphic viruses decrypt into a constant code body that can be recognized.

10) Thank You

I will be happy to answer any questions to the best of my abilities that you have with the remaining time. If you still have questions, I’d be happy to speak to you later.