ARE METAMORPHIC VIRUSES REALLY INVINCIBLE?

Arun Lakhotia, Aditya Kapoor, Eric Uday
University of Louisiana at Lafayette

SUMMARY

In the game of "hide and seek," where a virus tries to hide and the AV scanners tries to seek, the winner is the one that can take advantage of the other's weak spot. So far a virus writer has enjoyed the upper hand for she could exploit the limitations of AV technologies. Metamorphic viruses are particularly insidious in taking such advantage. A metamorphic virus thwarts detection by signature-based (static) AV technologies by morphing its code as it propagates. A virus can also thwart detection by emulation-based (dynamic) technologies. To do so it needs to detect whether it is running in an emulator and change its behavior. So are metamorphic viruses invincible?

This paper uncovers the Achilles' heel of a metamorphic virus.

INTRODUCTION

When you consider all the tricks that a virus writer can use to break AV scanners, metamorphic viruses, such as Win32.Evol, Metaphor, and Zmist, appear invincible. These viruses transform their code as they propagate, thus evading detection by analyzers that rely on static information extracted from previously observed virus code. The viruses also use code obfuscation techniques to hinder deeper static analysis. Such a virus can also beat dynamic analyzers by altering its behavior when it detects that it is executing under a controlled environment.

Lakhotia and Singh have discussed at length how a virus writer can fool AV scanners, even those based on the most advanced formal techniques (Virus Bulletin, September 2003). The limits of an AV scanner stem directly from the limits of static and dynamic analysis techniques, the foundation of all program analysis tools, including optimizing compilers. For AV scanners, the limits are debilitating for they operate in an environment where a programmer is its antagonist.

Metamorphic viruses enjoy the apparent invincibility because a virus writer has the advantage of knowing the weak spots of AV technologies. We could turn the tables if we can identify similar weak spots in a metamorphic virus. Indeed, Lakhotia and Singh close their otherwise gloomy article with one bright spot: "*The good news is that a virus writer is confronted with the same theoretical limit as anti-virus technologies… It may be worth contemplating how this could be used to the advantage of anti-virus technologies.*"

This paper investigates the above remark and identifies what promises to be the Achilles' heel of a metamorphic virus.

The key observation is that in order to mutate its code, generations after generations, a metamorphic virus must analyze its own code. Thus, it too must face the limits of static and dynamic analyses. Beyond that a metamorphic virus has another constraint: it must be able to re-analyze the mutated code that it generates. Thus, the analysis within the virus, of how to transform the code in current generation, depends upon the complexity of transformations in the previous generation. To overcome the challenges of static and dynamic analyses, the virus has the following options: do not obfuscate the transformed code in any generation; use some coding conventions that can aid it in detecting its own obfuscations; or develop smart algorithms to detect its specific obfuscations.

So, are metamorphic viruses really invincible? No, surely not as invincible, as they first seem to appear. A metamorphic virus' need to analyze itself is its Achilles' heel. If a virus can analyze itself then an AV scanner should also be able to analyze the virus by using whatever method a virus uses to work around its own obfuscations. It is then conceivable that one could create a reverse morpher that applies the transformation rules of a virus in reverse, thus undoing its attempt to hide from scanners.

Is there a catch? Before one can use a virus' methods on the virus itself, one has to extract those methods first. You must first have a sample of the virus in order to extract its transformation rules, assumptions, and algorithms. This chicken-and-egg problem is no different from that faced by the current AV technologies for extracting signatures and behaviors. The important thing is that once a set of tricks are identified and countered by the AV software, the virus writer is forced to invent new tricks, thus raising the bar

Published in *Virus Bulletin*, December 2004 and January 2005.

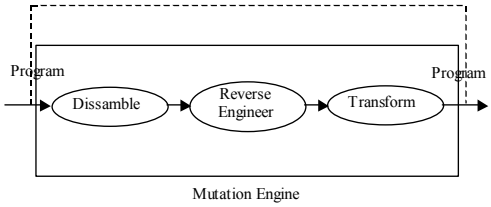

Figure 1. Stages of program transformation

for the virus writer. Because of the additional constraints, a virus writer has to be more imaginative than the makers of AV scanners.

The rest of the article is organized as follows. The next section provides an overview of mutation engines. It is followed by a discussion on the Achilles' heel of a metamorphic virus. We then present a case study by analyzing the metamorphic engine of Win32.Evol. This leads to a discussion on developing reverse morphers to undo the mutations performed by a mutation engine. The article closes with our conclusions and some notes in the appendix.

MUTATION ENGINES

The heart of a metamorphic virus is a mutation engine, the part of the virus code responsible for transforming its program. A mutation engine takes an input program and morphs it to a structurally different but semantically equivalent program.

Figure 1 identifies the three modules of any mutation engine: disassembly module, reverse engineering module and transformation module. Development of each of these modules poses different challenges and limitations.

In order to mutate its program, the virus must first disassemble it. One of the important tasks of disassembly is to differentiate between its code and data. If a virus cannot distinguish between code and data, it may transform the data, leading to incorrect behavior. There are two known strategies for disassembly: linear scan and recursive traversal (Schwarz et. al. 2002, Ninth Working Conference on Reverse Engineering, 2002). Each of these strategies has their own limitations (Linn, Debray 2003, Conference on Computer and Communications Security).

The third module, *Transform*, generates a transformed version of the original program. It must transform a program significantly to avoid being detected by a signature-based AV scanner. In the simplest case, the module may transform one instruction at a time. On the other extreme the module may analyze blocks of code and replace them with equivalent code fragments. To ensure correctness of transformation a block must be a single entry single exit piece of code. That means, that control should not jump into the middle of the block, or else it becomes harder to create semantic preserving transformations. One could also imagine transformations that replace segments of control flow graphs (CFGs) with other control flow graphs.

The second module, *Reverse Engineer* (RE), supports *Transform*. The challenges posed to this module depend upon the technique chosen for transformation. As the transformations become more complex, so does the work of reverse engineering. If *Transform* works on an instruction at a time then the RE module does not need to do anything. However, if *Transform* works on blocks of code, the RE module must identify blocks. Similarly, if *Transform* works on CFGs, the RE module should identify CFGs.

THE ACHILLES' HEEL

Lakhotia and Singh argue that virus writers enjoyed the upper hand because they can exploit the limitations of static analysis as well as dynamic analysis to hide their code. Junk byte insertion, jump into the middle of instruction and self-modifying codes are few obfuscation techniques that makes it even harder to statically distinguish between data and code in a binary executable. Insertion of large loops and anti-debugging techniques tests the patience and speed of dynamic analysis. A mutation engine that changes the virus code with every few generations and as well adds the complex obfuscation techniques to the newly created virus body might create a virus that is close to invincible.

Figure 1 shows that the steps involved in mutating a program are very similar to the steps outlined by Lakhotia and Singh for checking whether a program is malicious using program analysis techniques. There are two differences. First, a metamorphic virus uses the analysis of the first two steps for creating a transformed program. A scanner would use similar information to determine whether a program is malicious. Second, the output of the last step of a metamorphic virus becomes its input, albeit in a different execution of the program.

The feedback loop in Figure 1 has catastrophic consequences for a virus. A metamorphic virus has to analyze its own mutated code in order to

further mutate it. The complexity of analyzing its own code in the next generation increases with the complexity of mutations and obfuscations in the current generation. This increased complexity most likely will increase the virus' size or its runtime, thus making it vulnerable to detection.

To understand the problems faced in writing a metamorphic virus, let us analyze an obfuscation technique introduced by a non-metamorphic virus Netsky.Z. The virus Netsky.Z introduces an obfuscation technique called self-modifying code. The code is shown below. Here the virus is changing code at location 00403E6E at run time. It is adding 28h to the opcode 90h, which converts the NOP instruction to MOV instruction thus changing the code, as shown in Figure 2(b). If we try to analyze it statically we get the wrong analysis as shown by Figure 2(a).

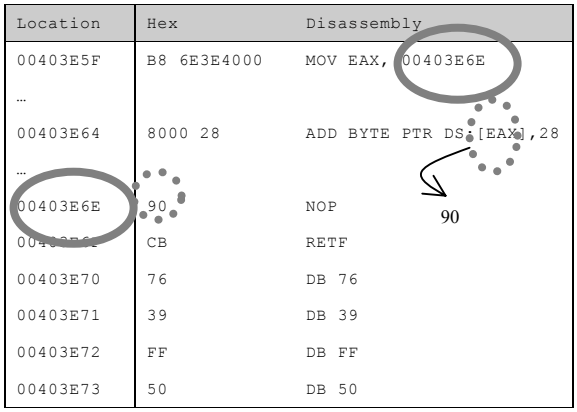

Figure 2 (a). Obfuscation through runtime code modification

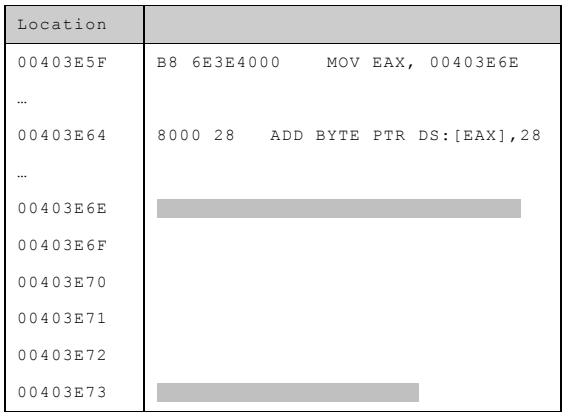

Figure 2(b). Modified Code

Now suppose a metamorphic virus writer has mutated its code such that the current generation is self-modifying, to further mutate its code it has to statically know the instruction that is changing at runtime. This challenge poses a serious limitation to the obfuscation techniques a metamorphic virus can impose during mutation.

This then highlights the Achilles' heel of a metamorphic virus: *A metamorphic virus must be able to disassemble and reverse engineer itself*. Thus, a metamorphic virus cannot utilize obfuscation techniques that make it harder or impossible for its code to be disassembled or reverse engineered by itself.

WIN32.EVOL: CASE STUDY

Win32.Evol is a relatively simple metamorphic virus. Nonetheless, it is a good example for a case study since the virus demonstrates properties common to metamorphic viruses, i.e., it obfuscates calls made to system libraries and it mutates its code before propagation.

The rest of this section describes the details of these methods.

Obfuscating System Calls

In order to perform any malicious act, a program would access the disk or the network. Access to these resources is controlled by the operating system. Thus, a quick way to determine whether a program is malicious is to look at the system calls it makes.

Win32.Evol does not use 'normal' procedure to make system calls. Thus, a disassembler, such as IDAPro, cannot directly determine the system calls it makes. It uses the following strategies to obfuscate its calls:

1) It computes the address of the kernel32.dll function GetProcAddress() by searching for the 8 byte sequence [0x55 00 01 F2 51 51 ec 8b] on Windows 2000[1].

2) Keeps the address of GetProcAddress() in its stack-based global data store, maintained at a certain distance from a magic marker pushed on the stack.

3) Uses a 'return' instruction to make a call to GetProcAddress().

[1] The Win32.Evol binary at http://vx.netlux.org looks for the byte sequence [0x55 00 00 0f 51 51 ec 8b], probably for a different version of Windows.

4) Maintains names of functions to be called as immediate, double-word operands of multiple instructions, not as strings in data store.

Mutation Engine

The mutation engine of Win32.Evol is a function consisting of the Disassembly and Transform modules of Figure 1. It does not have a Reverse Engineering module since it transforms an instruction at a time.

The mutation engine is located at address 00401FD7. It takes three inputs:

1. The Relocatable Virtual Address (RVA) of loaded virus code. RVA = 401000†
2. The length of the original virus code. LEN = arg_4 (1847)†
3. Pointer to buffer (BUF1) to store the transformed code. (Max Size buffer = 4 * LEN = arg_8 (7F0000)†

The output of the engine is the transformed program, which is placed in the buffer BUF1.

Disassembly module

The disassembly module of Win32.Evol uses the linear sweep algorithm. It checks whether a byte starts an instruction, if it does then it gets the size of the instruction, and disassembles the byte following the instruction.

If during disassembly the program comes across some byte that is not an instruction, the mutation process is abandoned.

Location	Instruction
0040227A	cmp al, 0FEh
0040227C	jz short loc_402282
	; If the byte under analysis is FE
	; goto 00402282
0040227E	cmp al, 0FFh
	; If the byte is FF goto 00402282
00402280	jnz short loc_4022B5
	; compare al with next opcode.
00402282	mov al, [esi+1]
	; If byte is either 0xFE or 0xFFload ModR/M
	; byte in al
00402285	and al, 38h
00402287	ror al, 3
0040228A	cmp al, 7
0040228C	jz loc_402532
	; If value of bits 3, 4, 5 of ModR/M byte are
	; 1 the instruction does not exist
	; Exit mutation process

Figure 3. Invalid instruction check

The mutation engine processes only a limited range of opcodes of the x86 instruction set. For instance, does not process floating-point instructions. The mutation is abandoned if an instruction outside its accepted range is encountered.

Figure 4 shows the code fragment from Win32.Evol doing the instruction range check.

Location	
00402118	cmp al, 0Fh
0040211A	jnz short loc_402152
0040211C	mov cl, [esi+1]
0040211F	cmp cl, 80h
00402122	jb loc_402532
00402128	cmp cl, 90h
0040212B	jnb loc_402532
	then exit mutation process

Figure 4. Invalid instruction 'range' check

Transform Module

The Transform module maps an instruction into one or more instructions. A detailed list of all transformations is given in Appendix A and B.

The transformation rules can be classified into two categories: deterministic and nondeterministic. A deterministic rule always transforms an instruction to the exact same sequence of instructions. For example, the following rule for transforming instruction *movsb* (opcode *0xA4*) is a deterministic transformation rule.

```
movsb   ➔   push eax
            mov  al, [esi]
            add  esi, 1
            mov  [edi], al
            add  edi, 1
            pop eax
```

Figure 5 shows the procedure of generating a fixed transformation for byte 0xA4 representing movsb.

Location	
004023B0	cmp al, 0xA4
004023B2	jnz 004023CE
004023B4	add esi,1
004023B7	mov eax, 83068A50
004023BC	stos dword ptr es:[edi]
004023BD	mov eax, 78801C6
004023C2	stos dword ptr es:[edi]
004023C3	mov eax, 5801C783
004023C8	stos dword ptr es:[edi]
004023C9	jmp 00401FF8

Figure 5. Transformation of byte 0xA4.

† Value during test run of Win32.Evol in debugger.

A non-deterministic rule may transform an instruction to different sequence of instructions. The following two rules demonstrate non-deterministic rules.

```
mov  eax, [ebp+4]      →      push   ecx
(8B 45 04)                    mov    ecx, ebp
                              add    ecx, 41h
                              mov    eax, [ecx-3Dh]
                              pop    ecx

mov  eax, [ebp+4]      →      push   esi
(8B 45 04)                    mov    esi, [ebp+4]
                              mov    eax, esi
                              pop    esi
```

Whenever the code introduced by a rule modifies a register, say *reg*, which was not modified by the original instruction, the mutated code is wrapped between a 'push *reg*' and 'pop *reg*' instructions.

Patching Relocatable Addresses

Win32.Evol does not contain any jump and call instructions that use absolute addresses, rather all the branching instructions uses relative jumps. It also does not contain indirect jumps and calls, where target address is available in a register or some other memory location. Since the transformations replace one instruction by multiple instructions, the mutation engine must also modify the relative addresses of the jump and call instructions.

To update the relative addresses, the mutation engine maintains another buffer, BUF2 of size 16 * *length of virus code*. For each instruction of the virus program, BUF2 has four entries as shown in Table 1. The first entry of table is *Source,* it points to the address of the n^{th} instruction in the virus code. The second entry, *Dest*, points to the address in BUF1 where the transformed virus code is stored. (N*ote that mutation engine takes BUF1 as input*). The other two entries are zero unless the instruction carries a relocatable offset. In case the instruction carries a relocatable offset, the third entry points to the address where the calculated offset is to be stored. The last entry stores the value of the current offset.

Table 1. A record in the buffer BUF2

Entry 1 (DWord)	Entry 2 (DWord)	Entry 3 (DWord)	Entry 4 (DWord)
Source	Dest	*next address following opcode*	*Original offset.*

The change in length of code results in change of relative addresses. To update the relative offsets, the algorithm searches for all the non-zero 'Entry 3' locations i.e. instructions having offsets. If an instruction ***I*** with a non-zero offset is found, it adds the original offset (Entry 4) to *Source* (Entry 1), to get address **a**. The address **a** is original destination address in Win32.Evol code. Since this destination address should start a valid instruction, there should be a valid record in BUF2 such that *Source* is equal to **a**. *Note that BUF2 has records corresponding to each valid instruction in virus code.* The difference between values of *Dest* at location of instruction ***I*** and *Dest* at location **a** gives us the new offset. This calculated offset gives number of bytes added in the transformed code. The offset is then patched back to the location pointed by Entry 3 at location of instruction ***I***.

DEFEATING WIN32.EVOL

Win32.Evol is no longer considered a major threat since most current AV scanners can catch it because of its relatively simple morphing engine. Yet, it may be worth contemplating how it could be defeated. The insights could lead to development of methods for defeating other such viruses.

Win32.Evol uses some very interesting techniques to obfuscate system calls. It is probably beyond the scope of current static analysis techniques to undo these obfuscations and identify the system functions being called by the virus. It appears to be futile to follow that direction.

However, the limitations of the metamorphic engine of Win32.Evol are clearly its weaknesses.

- It uses linear sweep for disassembling itself. Hence, it can be disassembled by most disassembler.
- It cannot use indirect jumps and calls because it cannot correctly transform them. Thus, its control flow graph can be created easily. Thereby simplifying its reverse engineering.
- Its deterministic transformation rules essentially replace a certain byte with a certain fixed sequence of bytes. These rules can be applied in reverse.
- The code generated by non-deterministic transformation rules follows the pattern: push *reg*, *instructions*, pop *reg*, where the *instructions* does not contain *push* or *pop*. The *push* and *pop* instructions form a pair of parenthesis. All such pairs are properly matched in the generated code. It should be

possible to undo the transformation using a parenthesis matching algorithms.

Now consider a program Undo.Evol that does the following: *It disassembles a program using linear sweep and then applies the transformations of Win32.Evol in reverse. The program continues to apply the transformations until none of the transformations can be applied.*

Will Undo.Evol program help in detecting versions of Win32.Evol?

Since the transformations of Win32.Evol always increase the code size, when applied in reverse they will always decrease the code size. Thus, Undo.Evol will always terminate. It is a matter of further study whether Undo.Evol will always terminate on a single program. If it can be shown that Undo.Evol terminates on a single program, say Min.Evol, then to detect Win32.Evol one may apply Undo.Evol on a binary and check for the signature of the Min.Evol.

CONCLUSIONS

Anti-virus scanner technology is constrained by the theoretical limits of program analysis techniques. A metamorphic virus is a manifestation of these limits. It turns out that to enjoy its advantage, a metamorphic virus too depends on program analysis techniques, because in order to mutate, a metamorphic virus must analyze its own code. Thus a metamorphic virus cannot use tricks that will fool its own analyzer. This handicap of a metamorphic virus can potentially be exploited to develop AV scanners. However, to revert the mutations in order to defeat a virus, the AV research community faces several key questions, such as: *How does one extract the assumptions of a virus and the transformations it performs? Will reverting the transformations lead to a single result? Will the reverse transformations terminate in polynomial time? How does one separate virus code from the code of the host?* Answers to some of these questions would be crucial in developing technology that takes advantage of a virus' Achilles' heel.

APPENDIX

This appendix summarizes the transformations performed by Win32.Evol. The description uses the following symbols.

- *imm → byte | word | dword*
- *A → <reg> | [<reg>] | [<reg>+imm]*
- *reg → al | ah | ax | eax | cl | ch | cx | ecx | dl | dh | dx | edx | bl | bh | bx | ebx | sp | esp | bp | ebp | si | esi | di | edi*
- *B → A – {<reg>}*
- *imm(i) → imm*

The meaning of these symbols follows from x86 architecture descriptions.

Deterministic Mutations

The following table summarizes the deterministic mutations. The first column of the table gives the opcode(s), the second column gives the mnemonic of the instruction(s) or describes the instructions, and the third column gives the result of transformation.

Opcodes (HEX)	Instruction	Deterministic Mutations
70 – 7F	Short displacement jump on condition.	Long displacement jump on condition. (0x0F) 0x80 – 0x8F two byte opcode
EB	jmp byte	jmp word/dword (0xE9)
FF, FE	push A	mov eax, A push eax
68, 6A	push imm	mov eax, imm push eax
AA/AB	stos(b/d)	mov [edi], eax add edi, (1/4)
AC/AD	loads(b/d)	mov al, [esi] add esi, (1/4)

A4/A5	movs(b/ w)	push eax
		mov al, [esi]
		add esi, (1 or 4)
		mov [edi], al
		add edi, (1 or 4)
		pop eax

map([0-3] [4-5]), map([0-2] [C-D]) A8, A9	(add/adc/and /xor/or/sbb/sub, test) <reg>, imm	Length of instruction increases due to addition of ModR/M byte. Instruction remains same.
	Instruction	**Identity Mutations**
E9, E8	jmp word/dword call word/dword	Unchanged
81 C4,	add (e)sp, …	Unchanged

81 EC	sub (e)sp, ….	
C0, C1, D0, D3	rol, ror, shl, sar, shr, rcr,sal, rcl	Unchanged
C2	ret near word / ret near	Unchanged
CD	int <byte>	Unchanged
8B EC	mov ebp,esp	Unchanged
F3	rep	Unchanged
F6, F7	test byte/(d)word	Unchanged
50 – 5F	push/pop	Unchanged
90	Nop	Unchanged

Non-Deterministic Mutations

The non-deterministic mutations replace an instruction by one of many alternative sequences of instructions. The specific sequence is chosen at random. We have extracted some sample transformations performed by the virus.

Win32.Evol transforms the following instructions are non-deterministically.

mov A, <reg>
mov <reg>, A
lea <reg>, B
add/adc/and/xor/or/sbb/sub A, <reg>
add/adc/and/xor/or/sbb/sub <reg>, A
inc/dec <reg>

All of the above are register-modifying instructions. The following strategy is used to generate the transformed code: transfer the register whose value is modified to another register, perform the original computation on the new register, transfer the value back to the new register. To ensure that the above does not change behavior, the new register value is saved by pushing on the stack before changing it and is popped back after the computation is complete.

Here are some example transformations:

```
mov   [ebp+8], eax → push   ecx
                     mov    ecx, ebp
                     add    ecx, 12h
                     mov    [ecx-0Ah], eax
                     pop    ecx

mov   al, [eax-0Dh] → push   edx
                      mov    dh, [eax-0Dh]
                      mov    al, dh
                      pop    edx

mov   eax, [ebp+4] → push   ecx
                     mov    ecx, ebp
                     add    ecx, 41h
                     mov    eax, [ecx-Dh]
                     pop    ecx
```