FEATURE

ARE METAMORPHIC VIRUSES REALLY INVINCIBLE? PART 1

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

In the game of hide and seek, where a virus tries to hide and AV scanners try to seek, the winner is the one that can take advantage of the other's weak spot. So far, the viruses have enjoyed the upper hand since they have been able to 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. The 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 behaviour.

So, are metamorphic viruses invincible?

INTRODUCTION

When you consider all the tricks that a virus writer can use to break AV scanners, metamorphic viruses, such as Win32/Evol, Metaphor (aka W32/Simile, see *VB*, May 2002, p.4) and W95/Zmist (see *VB*, March 2001 p.6) appear invincible. These viruses transform their code as they propagate, thus evading detection by analysers that rely on static information extracted from previously observed virus code. The viruses also use code obfuscation techniques to hinder deeper static analysis. Such viruses can also beat dynamic analysers by altering their behaviour when they detect that they are executing under a controlled environment.

Lakhotia and Singh have discussed at length how a virus can fool AV scanners, even those based on the most advanced formal techniques (see *VB*, September 2003, p.15). 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 the antagonist.

Metamorphic viruses enjoy their apparent invincibility because the virus writer has the advantage of knowing the weak spots of AV technologies. However, we could turn the tables if we could identify similar weak spots in metamorphic viruses. Indeed, Lakhotia and Singh close their otherwise gloomy article with one optimistic thought: '*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 article 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 analyse its own code. Thus, it too must face the limits of static and dynamic analysis. Beyond that a metamorphic virus has another constraint: it must be able to re-analyse the mutated code that it generates. Thus, the analysis within the virus, of how to transform the code in the 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? They are surely not as invincible, as they first appear. A metamorphic virus's need to analyse itself is its Achilles' heel. If a virus can analyse itself, then an AV scanner should also be able to analyse it by using whatever method the 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 the virus in reverse, thus undoing its attempt to hide from scanners.

Is there a catch? Before one can use a virus's methods on the virus itself, one has to extract those methods. One 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 behaviours. The important thing is that, once a set of tricks has been identified and countered by the AV software, the virus writer is forced to invent new tricks, thus raising the bar for the virus writer. Because of the additional constraints, the virus writer has to be more imaginative than the makers of AV scanners.

The rest of this two-part 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. In the second part of the article (which will appear in the January 2005 issue of *Virus Bulletin*) we present a case study by analysing the metamorphic engine of Win3/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.

MUTATION ENGINES

At 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 the virus code and data. If a virus cannot distinguish between code and data, it may transform the data, leading to incorrect behaviour. There are two known strategies for disassembly: linear scan and recursive traversal (Schwarz *et al.* 2002, Ninth Working Conference on Reverse Engineering). Each of these strategies has its own limitations (Linn, Debray 2003, Conference on Computer and Communications Security).

The third module, transform, generates a transformed version of the original program. A program must be transformed significantly in order to avoid being detected by a signature-based AV scanner. In the simplest case, the module may transform one instruction at a time. At the other extreme the module may analyse blocks of code and replace them with equivalent code fragments. To ensure accuracy 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, the reverse engineering (RE) module, supports the transformation module. The challenges for this module depend upon the technique chosen for transformation. As the transformations become more complex, so does the work of reverse engineering. If the transformation module works on one instruction at a time then the RE module does not need to do anything. However,

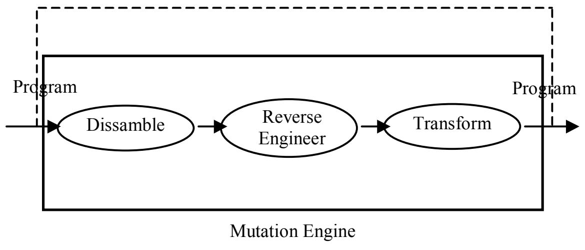


Figure 1. Stages of program transformation.

if the transformation module works on blocks of code, the RE module must identify blocks. Similarly, if the transformation module works on CFGs, the RE module should identify CFGs.

THE ACHILLES' HEEL

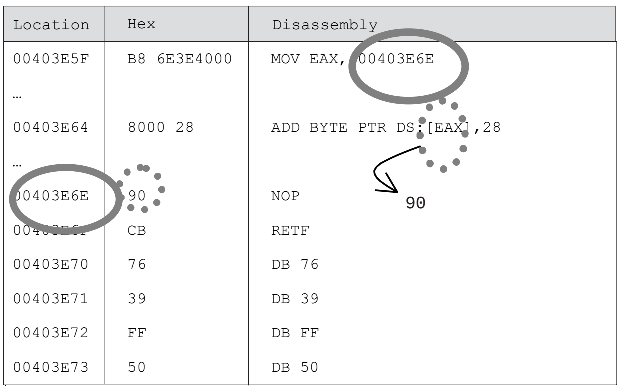
Lakhotia and Singh argued that virus writers enjoy the upper hand because they can exploit the limitations of static analysis as well as dynamic analysis to hide their code. Junk byte insertion, jump into the middle of instruction and self-modifying codes are a few obfuscation techniques that make it even harder to distinguish statically between data and code in a binary executable. Insertion of large loops and anti-debugging techniques test the patience and speed of dynamic analysis. A mutation engine that changes the virus code with every few generations as well as adding 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 analyse its own mutated code in order to mutate it further. If, after transformation, the virus introduces obfuscations that prevent its disassembly, then in the next generation the virus may not be able to diassemble itself. If it introduces obfuscations that prevent reverse engineering of the virus code – say, for instance, identifying its program blocks, then the virus will also not be able to detect its own blocks. Thus, the virus cannot introduce obfuscations that prevent those analyses that are performed by the virus itself.

To understand the problems faced in writing a metamorphic virus, let us analyse an obfuscation technique introduced by a non-metamorphic virus, W32/Netsky.Z.

The virus Netsky.Z introduces an obfuscation using a technique known as self-modifying code. Here, the virus is modifying code at location 00403E6E at run time. It adds 28h to the opcode 90h, which converts the NOP instruction to MOV instruction, thus modifying the code, as shown in Figure 2b. If we try to analyse it using a static technique we get the wrong analysis, as shown by Figure 2a.



Location	Hex	Disassembly
00403E5F	B8 6E3E4000	MOV EAX, 00403E6E
…		
00403E64	8000 28	ADD BYTE PTR DS:[EAX],28
…		
00403E6E	90	NOP
00403E6F	CB	RETF
00403E70	76	DB 76
00403E71	39	DB 39
00403E72	FF	DB FF
00403E73	50	DB 50

Figure 2a. Obfuscation through runtime code modification.

Location	Hex	Disassembly
00403E5F	B8 6E3E4000	MOV EAX, 00403E6E
…		
00403E64	8000 28	ADD BYTE PTR DS:[EAX],28
…		
00403E6E	B8 CB7639FF	MOV EAX, FF3976CB
00403E6F		
00403E70		
00403E71		
00403E72		
00403E73	50	PUSH EAX

Figure 2b. Modified code.

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

This 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 utilise obfuscation techniques that make it harder or impossible for its code to be disassembled or reverse engineered by itself.

WIN32/EVOL

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. Part two of this article (in next month's *VB*) will describe the details of these methods and discuss the development of reverse morphers to undo the mutations performed by a mutation engine.