

HUNTING FOR METAMORPHIC

Péter Ször and Peter Ferrie

Symantec Corporation, 2500 Broadway, Suite 200, Santa Monica, CA 90404-3036, USA
Tel + 1 310 453 4600 • Fax +1 310 453 0636 • Email pszor@symantec.com,
pferrie@symantec.com

ABSTRACT

As virus writers developed numerous polymorphic engines, virus scanners became stronger in their defense against them. A virus scanner which used a code emulator to detect viruses looked like it was on steroids compared to those without an emulator-based scanning engine.

Nowadays, most polymorphic viruses are considered boring. Even though they can be extremely hard to detect, most of today's products are able to deal with them relatively easily. These are the scanners that survived the DOS polymorphic days. For some of the scanners DOS polymorphic viruses meant the 'end of days'. Other scanners died with the macro virus problem. For most products the next challenge to take is 32-bit metamorphosis.

Metamorphic viruses are nothing new. We have seen them in DOS days, though some of them, like ACG, already used 32-bit instructions. The next step is 32-bit metamorphosis under Windows environments. Virus writers already took the first step in that direction.

In this paper the authors will examine metamorphic engines to provide a better general understanding of the problem that we are facing. The authors also provide detection examples of some of the metamorphic viruses.

1 INTRODUCTION

Remember the first time we were faced with MtE (The Dark Avenger Mutation Engine)? Initial research showed that most products were not able to detect MtE-based viruses 100% correctly. The product tests carried out by Vesselin Bontchev at VTC showed that most scanners missed a certain percentage of infected files that was occasionally as high as 10%. If infected files are replaced from backups, sooner or later this initial 10% miss can build up to 100% on a particular system. Everything could be infected on the machine but the scanner would not be able to detect a single infection!

Virus writers developed numerous polymorphic engines and, as a result, virus scanners became stronger in order to handle them. Virus code started to become more and more complex from the beginning.

We will examine the various ways virus writers challenged our scanning products over the last decade. Although most of these techniques are used to obfuscate file infector viruses, we can surely expect similar techniques to appear in modern *Windows* worms in the future.

Evolution of code has come a long way in binary viruses over the years. If someone checks the development result of viruses it might appear that almost everything possible was already done and problems are not escalating. However, there are still computing distribution models that have not been seen in viruses.

New *Windows* worms do appear and seem to dominate the field in their complexity. Some of the recent worms support a plug-in mechanism to evolve and introduce new changes and functionality in their code. Certainly such update mechanisms are the basics of the future models.

In this paper we will try to predict the next steps taken by the dark side.

2 EVOLUTION OF CODE

Virus writers continuously challenge anti-virus products. In particular their biggest enemies are the virus scanner products that are the most popular of all anti-virus software today. Generic AV solutions such as integrity checking and behaviour blocking never managed to take over the popularity of the anti-virus scanner.

The fact is that such generic virus detection models need a lot more thinking and technology in place under *Windows* platforms. These technologies were beaten by some of the old DOS viruses in the DOS days. As a result, some people draw the incorrect conclusion that these techniques are not useful.

Scanning is the accepted solution of the market regardless of its drawbacks. Thus it needs to be able deal with the escalating complexity and emerging number of distributed and self-distributing malware.

While modern computing developed extremely quickly, binary virus code could not catch up with the technology challenges for a long time. In fact, the DOS viruses evolved to a very complex level until 1996. However, 32-bit *Windows* started to dominate the market from that point.

As a result, virus writers had to go back years in binary virus developments. The complexity of DOS polymorphism peaked when Ply was introduced in 1996 with a new permutation engine. These developments could not continue. New 32-bit infection techniques needed to be discovered by the pioneer virus writers on Win32 platforms.

Some virus writers still find the *Windows* platforms far too challenging, especially when it comes to *Windows NT/2000/XP*. However, the basic infection techniques were already introduced, and standalone virus assembly sources are highly distributed on the Internet. These sources provide the basis of new mass-mailing worms that do not take major skill, but rather cut and paste abilities.

In this section we will examine the basic virus code obfuscation techniques from encrypted viruses up to the modern metamorphic techniques.

2.1 32-bit Encrypted Viruses

Virus writers tried to implement virus code evolution from the very early days. One of the easiest ways to hide the functionality of the virus code was encryption. One of the first DOS viruses that implemented encryption was Cascade. The virus starts with a constant decryptor that is followed by the encrypted virus body. Such a simple code evolution method appeared in 32-bit *Windows* viruses very early also. Win95/Mad and Win95/Zombie use exactly the same technique as Cascade. The only difference is the 32-bit implementation.

Detection of such viruses is still possible without trying to decrypt the actual virus body. In most cases the code pattern of the decryptor of these viruses is unique enough for detection. Obviously such detection is not exact. However, the repair code can decrypt the encrypted virus body and deal with the minor variants easily.

2.2 32-bit Oligomorphic Viruses

Unlike encrypted viruses, oligomorphic viruses do change their decryptors in new generations. Win95/Memorial had the ability to build 96 different decryptor patterns. Thus the detection of the virus based on the decryptor's code was not a practical solution, though possible. Most products tried to deal with the virus by dynamic decryption of the encrypted code instead. Thus the detection is still based on the constant code of the decrypted virus body.

Interestingly, some products that we tested could not detect all instances of Memorial. This is because such viruses need to be examined to their smallest details in order to find and understand the oligomorphic decryptor generator. Without such a careful manual analysis the slow oligomorphic virus techniques are impossible to detect reliably. Obviously they are a great opportunity for automated virus analysis centres.

2.3 32-bit Polymorphic Viruses

Win95/Marburg and Win95/HPS were the first viruses that used real 32-bit polymorphic engines. Polymorphic viruses can create an endless number of new decryptors that use different encryption methods to encrypt the constant part (except their data areas) of the virus body (see

Figure 1).

Some of the polymorphic viruses such as Win32/Coke use multiple layers of encryption. Other newer polymorphic engines such as the Win32/Crypto, built by a Czech virus writer, used a random decryption algorithm (RDA)-based decryptor that implemented brute force attack against its constant but variably encrypted virus body in a multi-encrypted manner. Manual analysis of such viruses might provide great surprises. Often there are inefficiencies of randomness in such polymorphic engines that provide an easy solution for detection, as long as algorithmic scanning is an available scanning option. Sometimes even a single wildcard string can do the magic of perfect detection.

Back in those days, most virus scanner products already had a code emulator capable of emulating 32-bit PE (Portable Executable) files. Other virus researchers only implemented dynamic decryption to deal with such viruses. That worked just like in the previous cases because the virus body was still constant under the encryption. According to the various AV tests some vendors were still sorry not to have support for difficult virus infection techniques.

Virus writers used the combination of entry point obscuring techniques with 32-bit polymorphism to make the scanner's job even more difficult. In addition, they tried to implement anti-emulation techniques to challenge code emulators.

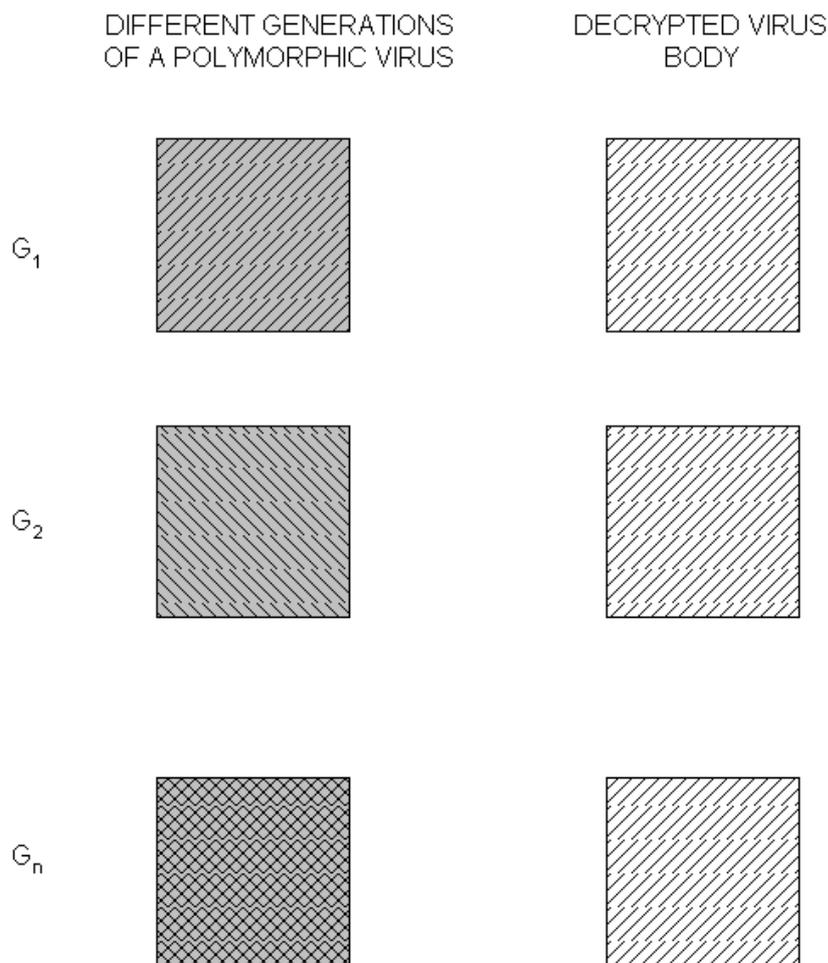


Figure 1. Generations of a polymorphic virus.

2.4 32-bit Metamorphic Viruses

Virus writers still need to waste weeks or months to create a new polymorphic virus that often does not have chance to appear in the Wild because of its bugs. On the other hand, a researcher might be able to deal with the detection of such a virus in a few minutes or few days. One of the reasons is that there are a surprisingly low number of efficient external polymorphic engines.

Obviously, virus writers try to implement various new code evolution techniques in order to make the researchers' job more difficult. Win32/Apparition virus was the first known 32-bit virus that did not use polymorphic decryptors to evolve itself in new generations. Rather, the virus carries its source and drops it whenever it can find a compiler installed on the machine. The virus inserts and removes junk code to its source and recompiles itself. This way a new generation of the virus will look completely different. It is fortunate that Win32/Apparition did not become a major problem. However, such a method would be more dangerous if implemented in a Win32 worm. Furthermore, these techniques are even more dangerous on platforms such as *Linux*, where C compilers are commonly installed with the standard system, even if the system is not used for development.

The technique of Win32/Apparition is not surprising. It is much simpler to evolve the code in source format instead of binary. Not surprisingly, many macro and script viruses use junk insertion and removal techniques to evolve themselves in new generations.

2.4.1 What is a metamorphic virus?

Igor Muttik explained metamorphic viruses the shortest possible way: 'Metamorphics are body-polymorphics.' Metamorphic viruses do not have a decryptor, nor a constant virus body. However, they are able to create new generations that look different. They do not use a constant data area filled with string constants but have one single code body that carries data as code.

Material metamorphosis (Figure 2) does exist in real life. For instance, shape memory polymers have the ability to transform back to their parent shape when heated. Metamorphic computer viruses have the ability to change their shape by themselves from one form to another, but usually they avoid generating instances that are very close to their parent shape. Figure 3 illustrates the problem of metamorphic virus bodies as multiple shapes.



Figure 2. Material metamorphosis: when heated, biodegradable shape-memory polymer transforms from a temporary shape (left) to its parent shape (right) within 20 seconds.

Although there are some DOS metamorphic viruses such as ACG (Amazing Code Generator), they did not become a significant problem for end users. In only a few months we will know more metamorphic 32-bit Windows viruses than metamorphic DOS viruses. The only difference between the two is the potential. The networked enterprise allows metamorphic binary worms to

cause major problems. As a result, we will not be able to turn a blind eye to them and say ‘we do not need to handle them since they are not causing problems to our users’. They will.

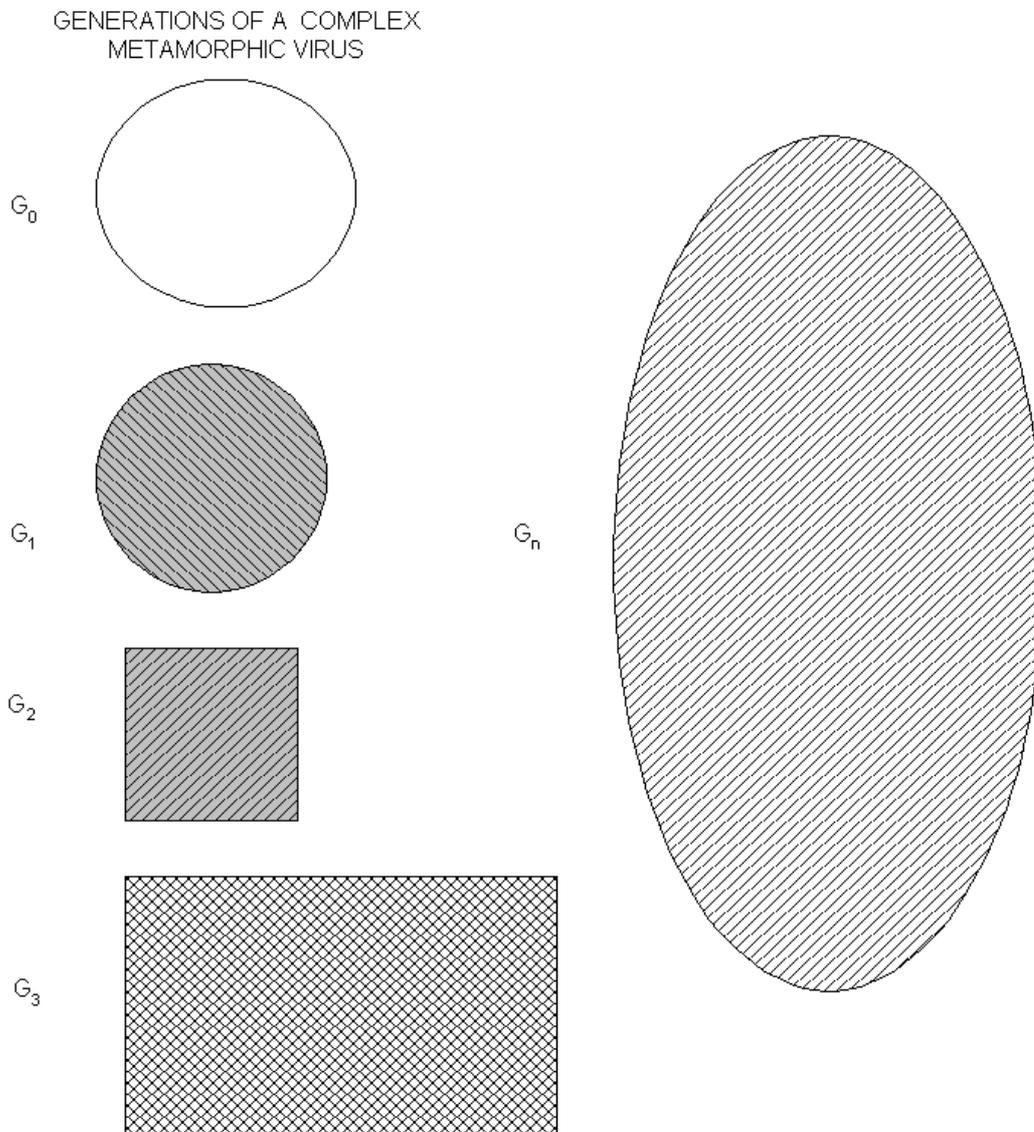


Figure 3: The virus body keeps changing in different generations of a metamorphic virus.

2.4.2 Simple metamorphic viruses

In December 1998, Vecna, (a notorious virus writer), created the Win95/Regswap virus. Regswap implemented metamorphosis via register usage exchange. Any part of the virus body will use different registers but the same code. Obviously the complexity of this is not very high. Figure 4 shows an example of code fragments selected from two different generations of Win95/Regswap.

```

5A          pop  edx
BF04000000 mov  edi,0004h
8BF5      mov  esi,ebp
B80C000000 mov  eax,000Ch
81C288000000 add  edx,0088h
8B1A      mov  ebx,[edx]
899C8618110000 mov  [esi+eax*4+00001118],ebx

58          pop  eax
BB04000000 mov  ebx,0004h
8BD5      mov  edx,ebp
BF0C000000 mov  edi,000Ch
81C088000000 add  eax,0088h
8B30      mov  esi,[eax]
89B4BA18110000 mov  [edx+edi*4+00001118],esi

```

Figure 4: Win95/Regswap uses different registers in new generations.

The bold areas show the common areas of the two code generations. Thus, a wildcard string could be useful to detect the virus. Moreover, support for half-byte wildcard bytes such as 5? B? (as described by Frans Veldman) could lead to an even more accurate detection. However, depending on the actual ability of the scanning engine such a virus might need an algorithmic detection because of the missing support of wild card search strings. If algorithmic detection is not supported as a single database update the product update might not come out for several weeks or months for all platforms!

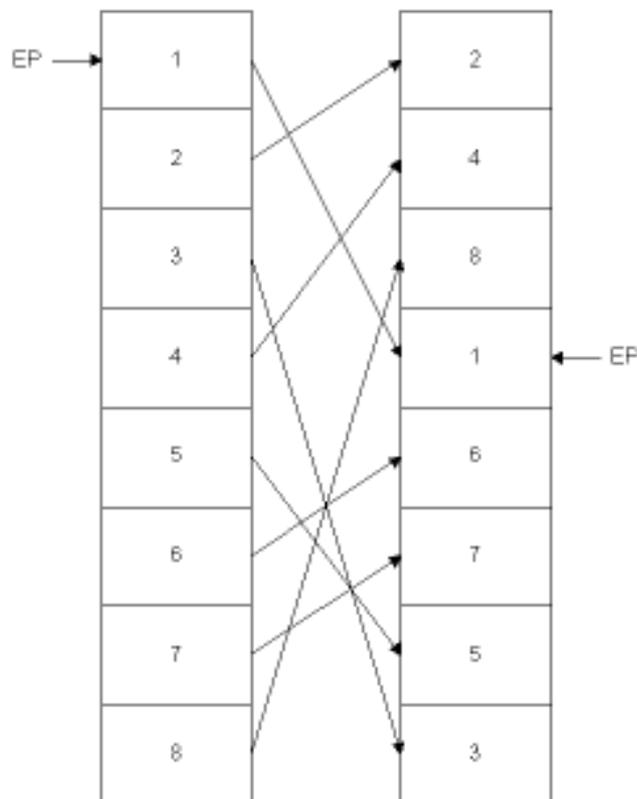


Figure 5. Example of module reordering with 8 modules.

Other virus writers tried to recreate older permutation techniques. For instance, the Win32/Ghost virus has the ability to reorder its subroutines similarly to the BadBoy DOS virus family (Figure 5).

The order of the subroutines will be different from generation to generation and this leads to $n!$ different virus generations, where n is the number of subroutines. BadBoy had eight subroutines, $8! = 40320$ different generations. Win32/Ghost (discovered in May 2000) had ten functions, $10! = 3628800$ combinations. Both of them can be detected with search strings, however, some scanners need to deal with such a virus algorithmically.

Two different variants of Win95/Zmorph virus appeared in January of 2000. The polymorphic engine of the virus implements a build and execute code evolution. The virus rebuilds itself on the stack with push instructions. Blocks of code decrypt the virus instruction by instruction and push them to the stack. The build routine of the virus is already metamorphic. The engine supports jump insertion and removal between any instructions of the build code. Regardless, code emulators can be used to deal with the virus easily. A constant code area of the virus will provide identification since the virus body is decrypted on the stack.

2.4.3 More complex metamorphic viruses and permutation techniques

The Win32/Evol virus appeared in early July, 2000. The virus implements a metamorphic engine. Evol is capable to run on any major Win32 platform. Figure 6 shows an example code fragment as mutated to a new form in a new generation of the same virus.

```

a. An early generation:

C7060F000055  mov     dword ptr [esi],5500000Fh
C746048BEC5151  mov     dword ptr [esi+0004],5151EC8Bh

b. And one of its later generations:

BF0F000055     mov     edi,5500000Fh
893E           mov     [esi],edi
5F            pop     edi
52            push   edx
B640           mov     dh,40
BA8BEC5151     mov     edx,5151EC8Bh
53            push   ebx
8BDA           mov     ebx,edx
895E04         mov     [esi+0004],ebx

c. And yet another generation with recalculated ("encrypted") "constant" data.

BB0F000055     mov     ebx,5500000Fh
891E           mov     [esi],ebx
5B            pop     ebx
51            push   ecx
B9CB00C05F     mov     ecx,5FC000CBh
81C1C0EB91F1   add     ecx,F191EBC0h ; ecx=5151EC8Bh
894E04         mov     [esi+0004],ecx

```

Figure 6: Example of Win32/Evol's code metamorphosis.

Even the ‘magic’ DWORD values (5500000Fh, 5151EC8Bh) are changed in newer generations of the virus as shown in Figure 6c. Therefore, any wild card strings based on them will not detect anything above the third generation of the virus. Win32/Evol’s engine is capable of inserting garbage in-between core instructions.

Variants of the Win95/Zperm family appeared in June and September of 2000, respectively. The method used is known from the Ply DOS virus. The virus inserts jump instructions into its code. The jumps will be inserted to point to a new instruction of the virus. The virus body is built in a 64K buffer that is originally filled with zeroes. The virus will not use decryption to decrypt itself. In fact, it will not regenerate a constant virus body anywhere. Instead, it creates new mutations by the removal and addition of jump instructions as well as garbage instructions. Thus there is no way to detect the virus with search strings in the files, or in the memory either.

Most polymorphic viruses decrypt themselves to a single constant virus body in memory. However, metamorphic viruses do not. Therefore, the detection of the virus code in memory needs to be algorithmic because the virus body does not become constant even there. Figure 7 explains the code structure changes of Zperm-like viruses.

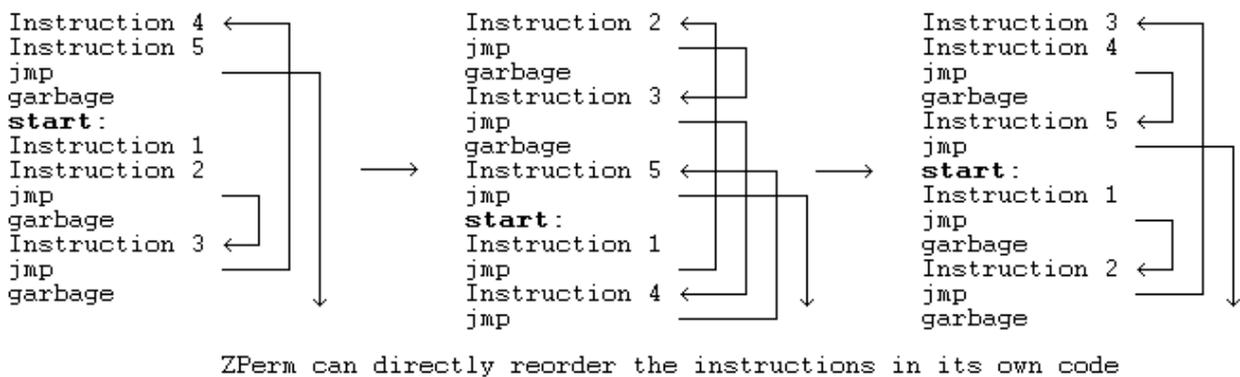


Figure 7. Zperm.A inserts JMP instruction into its code.

Sometimes, the virus replaces instructions with other equivalent instructions. For example, the instruction ‘xor eax, eax’ (which sets the eax register to zero) will be replaced by ‘sub eax, eax’ which also zeroes the contents of the eax register. The opcode of these two instructions will be different.

The core instruction set of the virus has the very same execution order; however, the jumps are inserted in random places. The B variant of the virus also uses garbage instruction insertion and removal such as ‘nop’ (a ‘do nothing’ instruction). It is easy to see that the number of generations can be at least n! where n is the number of core set instructions in the virus body.

Zperm introduced the Real Permutating Engine (RPME). RPME is available for other virus writers to create new metamorphic viruses. We should note here that permutation is only a single item on the list of metamorphic techniques. In order to make the virus truly metamorphic, instruction opcode changes are introduced. Encryption can be used in combination with anti emulation and polymorphic techniques.

In October 2000 two virus writers created a new permutation virus, Win95/Bistro, based on the

sources of Zperm virus and the RPME. To further complicate the matter, the virus uses a random code block insertion engine. A randomly activated routine builds a ‘do nothing’ code block at the entry point of the virus body prior to any active virus instructions. When executed, the code block can generate millions of iterations to challenge the code emulator’s speed.

Simple permutating viruses and complex metamorphic viruses can be very different in their implementation complexity. In any case, both permutating viruses and metamorphic viruses are different from the traditional polymorphic techniques.

In the case of polymorphic viruses, there is a particular moment when we can make a snapshot of the completely decrypted virus body, as illustrated in Figure 8. Typically, anti-virus software uses a generic decryption engine (based on code emulation) to abstract this process. It is not a requirement to have a complete snapshot to provide identification in a virus scanner, but it is essential to find a particular moment during the execution of virus code when a complete snapshot can be made, in order to classify a virus as a traditional polymorphic virus. It is efficient to have a partial result as long as there is a long enough decrypted area of each possible generation of the virus.

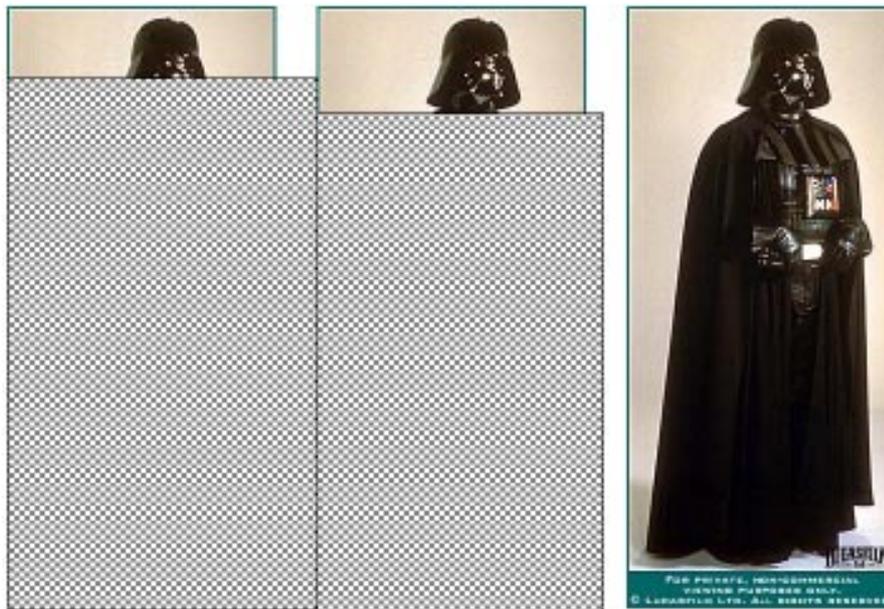


Figure 8: We can make a partial or complete snapshot of a polymorphic virus during its execution cycle.

On the contrary, a complex metamorphic virus does not provide this particular moment during its execution cycle. This is true even if the virus uses metamorphic techniques combined with traditional polymorphic techniques.

2.4.4 Mutating other applications: The ultimate virus generator?

Win95/Bistro not only mutates itself in new generations. It also mutates the code of its host by a randomly executed code morphing routine. This way the virus might generate new worms and viruses. Moreover, the repair of the virus cannot be done perfectly because the entry point code area of the application could be different. The code sequence at the entry point of the host application will be mutated for a 480 bytes long range. Figure 9 shows an original and a permuted

code sequence of a typical entry point code.

Original entry point code:	
55	push ebp
8BEC	mov ebp, esp
8B7608	mov esi, dword ptr [ebp + 08]
85F6	test esi, esi
743B	je 401045
8B7E0C	mov edi, dword ptr [ebp + 0c]
09FF	or edi, edi
7434	je 401045
31D2	xor edx, edx
Permutated entry point code:	
55	push ebp
54	push esp
5D	pop ebp
8B7608	mov esi, dword ptr [ebp + 08]
09F6	or esi, esi
743B	je 401045
8B7E0C	mov edi, dword ptr [ebp + 0c]
85FF	test edi, edi
7434	je 401045
28D2	sub edx, edx

Figure 9: Win95/Bistro entry point code permutation example.

Thus an instruction such as ‘test esi, esi’ can be replaced by ‘or esi, esi’, its equivalent format. A ‘push ebp, mov ebp, esp’ sequence (very common in high-level language applications) can be permutated to ‘push ebp, push esp, pop ebp’. Obviously it would be more complicated to replace the code with different opcode sizes, but it would be possible to shorten longer forms of some of the complex instructions and include ‘do nothing’ code as a filler.

This is a problem for all scanners. Heuristic scanners typically cannot deal with worms written in high-level languages yet. Obviously, some of these worms could be easily morphed to a new format. In a previous conference paper we introduced the problem of new virus variants being generated accidentally as a result of Portable Executable file repair. It is unfortunate that such mutations can appear, but it remains feasible to deal with that particular problem. On the other hand, code permutations of worms and viruses as done by Win95/Bistro will be much more difficult to deal with.

If a virus or a 32-bit worm were to implement a similar morphing technique, the problem could be major. New mutations of old viruses and worms would be morphed endlessly! Thus, a virtually endless number of not yet detectable viruses and worms would appear without any human intervention, leading to the ultimate virus generator.

An even more advanced technique was developed in the Win95/Zmist virus, which is described in the following sections.

At the end of 1999, the Win32/Smorph trojan was developed. It implements a semi-metamorphic technique to install a backdoor to the system. The standalone executable is completely regener-

ated during the installation of the trojan. The PE header will be recreated also, and will include new section names and section sizes. The actual code at the entry point is metamorphically generated. The code will allocate memory then decrypts its own resources which contain a set of other executables. The trojan uses API calls to its own import address table. The import table is filled with many non-essential API imports as well as some essential ones. Thus everything in the standalone trojan code will be different in new generations.

2.4.5 Advanced metamorphic viruses engines

2.4.5.1 Zmist

During VB 2000, Dave Chess and Steve White demonstrated their research result on Undetectable Viruses. Early this year the Russian virus writer Zombie released his *Total Zombification* magazine, with a set of articles and viruses of his own. One of the articles in the magazine was entitled 'Undetectable Virus Technology'.

Zombie has already demonstrated his set of polymorphic and metamorphic virus writing skills. His viruses have been distributed for years in source format and other virus writers have modified them to create new variants. Certainly this will be the case with Zombie's latest creation, W95.Zmist.

Many of us have not seen a virus approaching this complexity for a few years. We could easily call Zmist one of the most complex binary viruses ever written. W95.SK, One_Half, ACG, and a few other virus names popped to our mind for comparison. Zmist is a little bit of everything: it is an entry point-obscuring (EPO) virus that is metamorphic. Moreover, the virus randomly uses an additional polymorphic decryptor.

The virus supports a unique new technique: code integration. The Mistfall engine contained in the virus is capable of decompiling Portable Executable files to its smallest elements, requiring 32MB! of memory. Zmist will insert itself into the code: it moves code blocks out of the way, inserts itself, regenerates code and data references, including relocation information, and rebuilds the executable. This is something that has not been seen in any previous virus.

Zmist occasionally inserts jump instructions after every single instruction of the code section, each of which will point to the next instruction. Amazingly, these horribly modified applications will still run as before, just like the infected executables do, from generation to generation. In fact we have not seen a single crash during the test replications. Nobody expected this to work, not even its author Zombie. Although it is not foolproof it seems to be good enough for a virus. It takes some time for a human to find the virus in infected files. Because of this extreme camouflage Zmist is easily the perfect anti-heuristics virus.

They say a good picture tells a thousand words. The T-1000 model of Terminator 2 (Figure 10) is the easiest possible analogy to use. Zmist integrates itself into the code section of the infected application as the T-1000 model can hide itself on the floor.

2.4.5.1 Initialisation

Zmist does not alter the entry point of the host. Instead, it merges itself with the existing code, becoming part of the instruction flow. However, the random location of the code means that



Figure 10: T-1000 of Terminator 2.

sometimes the virus will never receive control. If the virus does run, then it will immediately launch the host as a separate process, and hide the original process (if the RegisterServiceProcess() function is supported on the current platform) until the infection routine completes. Meanwhile, the virus will begin searching for files to infect.

2.4.5.2 Direct Action Infection

After launching the host process, the virus will check if there are at least 16 MB of physical memory installed and that it is not running in console mode. If these checks pass, then it will allocate several memory blocks, including a 32 MB area for the Mistfall workspace, permute the virus body, and begin a recursive search for Portable Executable .EXE files. This search will take place in the *Windows* directory and all subdirectories, the directories referred to by the PATH environment variable, then all fixed or remote drives from A: to Z:. This is a rather brute force approach to spreading.

2.4.5.3 Permutation

The permutation is fairly slow because it is done only once per infection of a machine. It consists of instruction replacement, such as the reversing of branch conditions, register moves replaced by push/pop sequences, alternative opcode encoding, xor/sub and or/test interchanging, and garbage instruction generation. It is the same engine, Real Permutating Engine (RPME), used in several viruses, including W95.Zperm, also written by Zombie.

2.4.5.4 Infection of Portable Executable Files

A file is considered infectable if it smaller than 448 KB, begins with 'MZ' (*Windows* does not support 'ZM' format applications), is not infected already (the infection marker is 'Z' at offset 0x1C in the MZ header. This field is generally not used by *Windows* applications), and is a Portable Executable file.

The virus will read into memory the entire file, then choose from one of three possible infection types. With a one in ten chance, only jump instructions will be inserted between every existing instruction (if the instruction was not a jump already), and the file will not be infected; with a one in ten chance, the file will be infected by an unencrypted copy of the virus; otherwise, the file will be infected by a polymorphically-encrypted copy of the virus. The infection process is protected by Structured Exception Handling, which prevents crashes in the case of errors occurring. After the rebuilding of the executable is completed, the original file is deleted and the infected file is created in its place. However, if an error occurs during the file creation, then the original file is lost and nothing will replace it.

The polymorphic decryptor consists of ‘islands’ of code that are integrated into random locations throughout the host code section, and linked together by jumps. The decryptor integration is performed in the same way as for the virus body integration – existing instructions are moved to either side, and a block of code is placed in between them. The polymorphic decryptor uses absolute references to the data section, but the Mistfall engine will update the relocation information for these references, too. An anti-heuristic trick is used for decrypting the virus code: instead of making the section writable in order to alter its code directly, the host is required to have, as one of the first three sections, a section containing writable, initialised data. The virtual size of this section is increased by 32 KB, large enough for the decrypted body and all variables used during decryption. This allows the virus to decrypt code directly into the data section, and transfer control to there.

If such a section cannot be found, then the virus will infect the file without using encryption. The decryptor will receive control in one of four ways: via an absolute indirect call (0xFF 0x15), a relative call (0xE8), a relative jump (0xE9), or as part of the instruction flow itself! If one of the first three methods is used, the transfer of control will appear soon after the entry point. In the case of the last method, though, an island of the decryptor is simply inserted into the middle of a subroutine, somewhere in the code (including before the entry point). All used registers are preserved before decryption and restored afterwards, so the original code will behave as before. *Zombie* calls this last method ‘UEP’, perhaps an acronym for Unknown Entry Point, because there is no direct pointer anywhere in the file to the decryptor.

When encryption is used, the code is encrypted with one of ADD/SUB/XOR with a random key, and this key is altered on each iteration by ADD/SUB/XOR with a second random key. In between the decryption instructions are various garbage instructions, using a random number of registers, and a random choice of loop instruction, all produced by the Executable Trash Generator engine (ETG), also written by *Zombie*. Randomness features very heavily in this virus.

2.4.5.5 Code Integration

The integration algorithm requires that the host has fixups, in order to distinguish between offsets and constants, however after infection, the fixup data are not required by the virus. Therefore, though it is tempting to look for a ~20 Kb long gap in the fixup area, which would suggest that the virus body is located there, it would be dangerous to rely on this during scanning.

If another application (such as one of an increasing number of viruses) were to remove the fixup data, then the infection will be hidden. The algorithm also requires that the name of each section in the host is one of the following: ‘CODE’, ‘DATA’, ‘AUTO’, ‘BSS’, ‘TLS’, ‘.bss’, ‘.tls’, ‘.CRT’, ‘.INIT’, ‘.text’, ‘.data’, ‘.rsrc’, ‘.reloc’, ‘.idata’, ‘.rdata’, ‘.edata’, ‘.debug’, ‘DGROUP’.

These section names are produced by the most common compilers and assemblers in use, those of Microsoft, Borland, and Watcom. The names are not visible in the virus code, because the strings are encrypted.

A block of memory is allocated that is equivalent to the size of the host memory image, and each section is loaded into this array at the section's relative virtual address. The location is noted of every interesting virtual address (import and export functions, resources, fixup destinations, and the entry point), and then the instruction parsing begins.

This is used in order to rebuild the executable. When an instruction is inserted into the code, all following code and data references must be updated. Some of these references might be branch destinations, and in some cases the size of these branches will increase as a result of the modification. When this occurs, more code and data references must be updated, some of which might be branch destinations, and the cycle repeats. Fortunately – at least from *Zombie's* point of view – this regression is not infinite, so that, while a significant number of changes might be required, the number is limited. The instruction parsing consists of identifying the type and length of each instruction. Flags are used to describe the types, such as instruction is an absolute offset requiring a fixup entry, or instruction is a code reference, etc.

There are cases where an instruction cannot be resolved in an unambiguous manner to either code or data. In that case, *Zmist* will not infect the file. After the parsing stage is completed, the mutation engine is called, which inserts the jump instructions after every instruction, or generates a decryptor and inserts the islands into the file. Then the file is rebuilt, the relocation information is updated, the offsets are recalculated, and the file checksum is restored. If there are overlay data appended to the original file, then they are copied to the new file, too.

3 METAMORPHIC VIRUS DETECTION EXAMPLES

There is a level of metamorphosis beyond which no reasonable number of strings can be used to detect the code that it contains. At that point, other techniques must be used, such as examination of the file structure or the code stream, or analysis of the code's behaviour.

In order to detect a metamorphic virus perfectly, a detection routine needs to be written that is capable of regenerating the essential instruction set of the virus body from the actual instance of the infection. Other products use shortcuts to try to solve the problem but such shortcuts often lead to an unacceptable number of false positives. In this section we give an introduction to some of the techniques that can be useful.

3.1 Geometric Detection

Geometric detection is the technique of virus detection based on alterations that a virus has made to the file structure. It could also be called the 'shape heuristic', since it is far from exact, and prone to false positives. An example of a geometric detection can be demonstrated using *W95/ZMist*. This virus, when it infects a file using its encrypted form, increases the virtual size of the data section by at least 32 KB, but does not alter the physical size of the section.

Thus, a file might be reported as being infected by *W95/ZMist* if the file contains a data section

whose virtual size is at least 32 KB larger than its physical size. However, such a file structure alteration can also be an indicator of a runtime-compressed file. Very often file viruses do rely on a virus infection marker to detect already infected files and avoid multiple infections. Such an identifier can be useful for the scanner to use in combination with the other geometric changes of the file caused by the virus infection. This makes the geometric detection more reliable but the risk of false positive only gets smaller, it never gets nullified.

3.2 Disassembling Techniques

To assemble means to bring together, so to disassemble is to separate or take apart. In the context of code, to disassemble is to separate the stream into individual instructions. This is useful for detecting viruses which insert garbage instructions between their core instructions. Simple string searching cannot be used for such viruses because instructions can be quite long and there is a possibility that a string can appear 'inside' an instruction, instead of being the instruction itself. For example, suppose that one wished to search for the instruction `CMP AX, 'ZM'`. This is a common instruction in viruses, and is used to test if file is of executable type. Its code representation is:

```
66 3D 4D 5A
```

and it can be found in the stream

```
90 90 BF 66 3D 4D 5A
```

however when disassembled and displayed,

```
NOP
```

```
NOP
```

```
MOV EDI, 5A4D3D66
```

it can be seen that what was found is not the instruction at all. The use of a disassembler would prevent such mistakes, and if the stream were examined further

```
90 90 BF 66 3D 4D 5A 90 66 3D 4D 5A
```

when disassembled and displayed,

```
NOP
```

```
NOP
```

```
MOV EDI, 5A4D3D66
```

```
NOP
```

```
CMP AX, "ZM"
```

it can be seen that the true string follows shortly afterwards.

When combined with a state machine, perhaps to record the order in which are encountered 'interesting' instructions, and even combined with an emulator, it presents a powerful tool that makes a comparatively easy task of detecting viruses such as W95/ZMist, and the more recent W95/Puron which is based on the Lexotan engine.

Lexotan and W95/Puron execute the same instructions in the same order, with only garbage instructions and jumps inserted between the core instructions, and no garbage subroutines. This makes them easy to detect using only a disassembler and a state machine.

Sample detection of W95/Puron:

```
MOVZX    EAX, AX
MOV      ECX, DWORD PTR [EDX + 3C]    ;interesting
XOR      ESI, ESI
MOV      ESI, 12345678
CMP      WORD PTR [EDX], 'ZM'        ;interesting
MOV      AX, 2468
```

ACG, by comparison, is a quite complex metamorph that requires an emulator combined with a state machine. Sample detection is included in the next section.

3.3 Use of Emulators for Tracing

A CPU emulator is an application that simulates the behaviour of a CPU. It is very useful for working with viruses, as it allows virus code to execute in an environment from which it cannot escape. Code that runs in an emulator can be examined periodically or when particular instructions are executed. For DOS viruses, INT 21h is a common instruction to intercept.

3.3.1 Sample detection of ACG

A short example code of an instance of ACG:

```
MOV  AX, 65A1
XCHG DX, AX
MOV  AX, DX
MOV  BP, AX
ADD  EBP, 69BDAA5F
MOV  BX, BP
XCHG BL, DH
MOV  BL, BYTE PTR DS:[43A5]
XCHG BL, DH
CMP  BYTE PTR GS:[B975], DH
SUB  DH, BYTE PTR DS:[6003]
MOV  AH, DH
INT  21
```

When the INT 21 is reached, the registers contain ah=4a and bx=1000. This is constant for one class of ACG viruses. By trapping enough similar instructions is the basis for detection of ACG.

Surprisingly, not all anti-virus scanner products support such detection. ACG is used regularly in *Virus Bulletin* tests. Nevertheless, the majority of the anti-virus software misses it from time to time.

This shows that traditional code emulation logic in older virus scanner engines might not be used 'as is' to trace code on such a level. It is evident that all anti-virus scanners need to go in the direction of interactive scanning engine developments.

An interactive scanning engine model is particularly useful to build algorithmic detections of the

kind that ACG needs. We strongly recommend this technique to be used for those scanner developers that are currently unable to detect ACG with their existing technology.

3.3.2 Sample Detection of Evol

Previously we discussed the complexity of the Evol virus. Evol is a perfect example of a virus that deals with the problem of hiding constant data as variable code in itself from generation to generation. It is easy to see that code tracing can be particularly useful to detect even such level of changes. Evol builds the constant data on the stack from variable data, before it passes them to the actual function or API that needs them.

At a glance it seems that emulation cannot deal with such viruses effectively. However, the reality is that, with appropriate break points, emulators can be our best friend again. The only thing that it takes is a p-code language that can be used to write algorithmic detections. If the p-code is capable of emulating the applications from anywhere till a given set of break points, the stack can be analysed for the data that are built by the virus. Stack analysis can be very helpful to deal with complex metamorphic viruses that often decrypt data on the stack.

3.3.3 Use of Negative and Positive Features

In order to make the detection faster, the scanners can use negative detections. Unlike a positive detection that checks for a set of patterns that does exist in the virus body, negative detections check for the opposite. It is often enough to identify a set of instructions that do not appear in any of the instances of the actual metamorphic virus.

Such a negative detection can be used to stop the detection process once a common negative pattern is encountered.

3.4.3 Using Emulator-Based Heuristics

Heuristics were greatly discussed over the last decade. The method that covers ACG in our example (3.3.1) is essentially very similar to a DOS heuristics detector. If the DOS emulator of the scanner is capable to emulate 32-bit code (which is generated by ACG) it can easily cover that virus heuristically. The actual heuristics engine might track the interrupts or even implement a deeper level of heuristics using a Virtual Machine (VM) that simulates some of the functions of the operating system. Such systems can even ‘replicate’ the virus inside their Virtual Machine on a virtual file system built into the VM of the engine. Such a system was implemented in some of the AV scanner solutions and found to be very effective. They also provide a much better false positive ratio.

Nowadays it is easy to think of an almost perfect emulation of DOS. The actual computing speed of today’s processors and the relatively simple single-threaded OS allows this to happen. However, it is more difficult to emulate *Windows* on *Windows* built into a scanner! Emulating multi-threaded functionality and not having problems with synchronizations is a very challenging task. Such a system cannot be as perfect as a DOS emulation because of the complexity of the OS. Even if we can think to use a system such as VMWARE to solve most of the challenges there can be plenty of problems remaining. Emulation of third-party DLLs is one of the possible problems that can arise.

Performance is another problem. A scanner needs to be fast enough otherwise people will not use it. Fast is not always better when it comes to scanners. However the thing is that in real life people often get the impression that faster is better. Thus even if we have all the possible resources to develop such a perfect Virtual Machine to emulate *Windows* on *Windows* inside a scanner, we would need to compromise regarding speed. This will result in an imperfect system. In any case extending the level of emulation of *Windows* inside the scanner's system is a good idea and leads to better heuristics reliability.

Unfortunately, EPO viruses (such as Zmist) can easily challenge such a system.

Moreover, there is a full class of anti-emulation viruses. Even ACG used tricks to challenge emulators. The virus often does replicate only on certain days and in case of other similar conditions. Therefore, perfect detection is more difficult to be done by using pure heuristics without paying some attention to virus-specific details.

If an implementation ignores such details the virus could be missed in case of many different entities. Imagine running a detection test on a Sunday against a few thousand samples that only replicate during Monday to Friday. Depending on the heuristics implementations the virus can be easily missed. There are viruses like W32/Magistr that do not infect without an active internet connection.

What if the virus looks for www.antiheuristictrick.com? What would be the proper answer for such a query? Someone could claim that a proper real world answer could be provided for such a query, but could you really do that from a scanner during emulation?

Certainly it is not possible to be done perfectly. There will be viruses that cannot be detected with any emulated environments, no matter how good the emulator of the system. Some of these viruses will be metamorphic too. For such viruses only specific virus detection can provide a solution. Thus heuristics systems can only reduce the problem against masses of viruses. It is a good feeling if your product handles VCG (VCG is a DOS metamorphic without major anti-emulation tricks but buggy code) and many others.

Typically with metamorphic viruses we often see different infection methods implemented in the same virus. In the next section we discuss some other possible trends.

4 POSSIBLE FUTURE VIRUS DEVELOPMENTS

Over the last decade a couple of strong polymorphic engines were developed in the form of an external engine. We can expect that a number of metamorphic engines will be written during the next couple of years also and polymorphic virus development will continue in the foreseeable future.

Even though our paper focuses primarily on file infector viruses, we should not forget about the possibility of using metamorphic techniques in computer worms. The Hybris worm is a very good example how virus writing techniques developed to an updateable technology over the years.

Such concepts were already considered from the very early years of virus developments. For

instance, the old Yankee Doodle virus supports repair of older versions of itself. After removing its previous copy it infects the same file with the newer version of the virus. Basically the same is happening in case of computer worms such as Hybris but in a much faster way using networked models in a secure manner using public key infrastructure.

Hybris has a module that supports metamorphic file infections but this module was developed for testing purposes and has not been found in the Wild yet. There is major potential to deliver a metamorphic engine to thousands of already compromised systems, from one day to the next.

Most of the major virus creators such as Spanska, Zombie, and Sandman were involved in this worm project and helped Vecna. They are the same guys who developed DOS metamorphic and polymorphic viruses in the past.

Unfortunately, Hybris shows that virus writers can get together to form a project. Such projects are often more complicated to deal with than regular viruses written by a single person.

It is very likely that virus writers will develop models in which a set of viruses are able to communicate with each other, exchange information about compromised systems (exchange password and user information, IP addresses to remotely execute code via a backdoor, etc) or evolve each other by exporting and importing code modules.

We already encountered cases of dangerous viruses in a natural combination. For instance the W32/HLLW.QAZ got infected by the W32/FunLove virus. QAZ carries a backdoor feature that was used to break in to large US corporations. W32.FunLove is capable of reducing the security of IA32-based *Windows NT* systems, by patching the OS kernel file. When QAZ gets infected with FunLove and infects a *Windows NT* system the attacker will likely to have an even better control over the compromised system.

A new level of code evolution could appear in viruses via communication. It only takes a standard and an interface. In theory, viruses could evolve to a level where a virus would be able to export a polymorphic or metamorphic engine of itself for use in another virus or worm. Similarly, viruses would be able to exchange trigger routines and appear in newer and newer combinations. This sounds superficial but the technology is out there to support these kinds of models.

5 CONCLUSION

Evolution of metamorphic viruses is one of the great challenges of this decade. Clearly, virus writing is evolving to the direction of modern computer worms. From the perspective of the anti-virus researchers it is going to be a very interesting time.

It is more important than ever to realise how important team work is. Virus detection is not a one-man heroic job any more. The QA people need to understand the virus detection problems better than ever and work more closely with the virus researchers towards the goal of a faster scanning speed.

We have arrived in the times when detection of a single zoo virus can easily cause a 5–15% slowdown of scanning speed, assuming that the scanning engine and the virus researchers could handle the detection at the first place. We have not even mentioned the complexity of the repairs. This is a worrying result.

Metamorphic viruses are only one technique on the palette of virus writers. However, nobody in this business should ignore them for too long.

REFERENCES

- [1] Cohen, Frederick B., *A short course on computer viruses*, 1994.
- [2] Szor, Peter, 'The New 32-bit Medusa', *Virus Bulletin*, December 2000, pp.8–10.
- [3] Marinescu, Andrian, 'ACG in the Hole', *Virus Bulletin*, July 1999, pp.8–9.
- [4] Ferrie, Peter & Szor, Peter, 'Zmist Opportunities', *Virus Bulletin*, March 2001, pp.6–7.
- [5] Carey Nachenberg, personal communication.
- [6] Ferrie, Peter, 'Magisterium Abraxas', *Virus Bulletin*, May 2001, pp.6–7.
- [7] Szor, Peter, 'Attacks On Win32 – Part II', *VB Conference Proceedings*, September 2000, pp.101–121.
- [8] Bontchev, Vesselin, 'MtE Detection Test', *Virus News Int.*, January, 1992, pp.26–34.
- [9] Nikishhin, Andy, 'Harnessing Hybris', *Virus Bulletin*, January, 2001, pp.6–7.
- [10] 'Shape Shifters', *Scientific American*, May 2001, pp.20–21.

