Advanced Metamorphic Techniques in Computer Viruses

Philippe Beaucamps

***Abstract*—Nowadays viruses use polymorphic techniques to mutate their code on each replication, thus evading detection by antiviruses. However detection by emulation can defeat simple polymorphism: thus metamorphic techniques are used which thoroughly change the viral code, even after decryption. We briefly detail this evolution of virus protection techniques against detection and then study the METAPHOR virus, today's most advanced metamorphic virus.**

***Keywords*—Computer virus, Viral mutation, Polymorphism, Metamorphism, MetaPHOR, Virus history, Obfuscation, Viral genetic techniques**

I. INTRODUCTION

When the first antiviral protections appeared in the late 80's to answer the nascent viral threat, they consisted of a mere binary scan of programs looking for known virus signatures. Never mind, virus writers adapted their code so that it would mutate its binary form on each replication: as early as in 1988 a first virus protected itself using encryption, followed in 1990 by the first *polymorphic* viruses which were able to mutate their code as well as their decryption method. Their ability to evade detection by the then antivirus software gave them immediate "popularity". Nevertheless antiviruses quickly adapted to this protection by letting viruses decrypt themselves and then only scanning the decrypted code looking for any known signature. This led, as early as in 1997, to the first *metamorphic* viruses which mutate their code *in its decrypted form*.

This article will therefore study polymorphism and its miscellaneous techniques and more particularly the most evolved ones, namely metamorphic techniques. In order to do so, we will study most notably the 2002 METAPHOR virus. For more details, the reader may consult Éric Filiol's books [Fil05], [Fil07] as well as the VX Heavens website, which is crammed with malware resources.

II. POLYMORPHISM – EARLY STAGES

This section shortly describes the evolution and techniques of viruses from the most basic techniques to simple polymorphic techniques and finally to advanced metamorphic techniques. The reader may refer to [Fil05], [Fil07], [Szo05], [Ayc06] for a more exhaustive and detailed study.

Philippe Beaucamps is with the Loria, Nancy, France, email: ph.beaucamps_at_loria_dot_fr,

and also with the Virology and Cryptology Lab of the École Supérieure et d'Application des Transmissions (Army Signals Academy), Rennes, France

A. First viruses

The first virus outbreak broke out in 1981 with the ELK CLONER virus, followed by BRAIN in 1986, the first virus to implement stealth techniques, and from then by numerous other viruses. The most commonly used techniques consisted in appending the viral code at the end of the executable file, modifying the entry point to point at the virus and then letting the virus spread among the system (fig. 1). Thus, a basic protection method is *form analysis* where each virus is identified by a specific signature: such a signature is a sequence of – not necessarily consecutive – bytes whose detection inside a program allows to identify as undeniably as possible infection by the virus. This method has the advantage of being non-greedy in its complexity as well as subject to a tiny rate of false alarms.

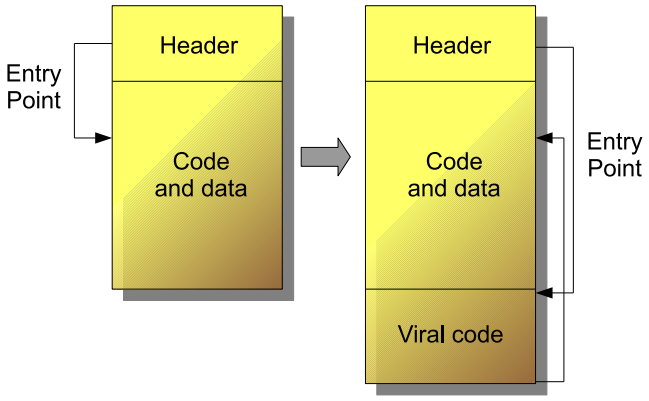

Fig. 1. Basic virus infection.

Back in time, as early as in 1984, F. Cohen had been the first one to study viruses from a theoretical point of view, christening them and defining them as programs which are able to infect other programs with a possibly evolved copy of themselves. Thus, this definition already suggested the existence of viruses which would alter their form when replicating. And indeed such viruses turned up quite quickly. Cohen also showed that the problem of virus detection was undecidable, meaning in other words that no algorithm would ever be able to determine with unquestionable certainty whether a given program is a virus or not [Coh84].

B. Polymorphic viruses

The first virus encrypting its code, CASCADE, appeared in 1988. Yet its decryption method remained unchanged from one replication to another and thus it was not really a polymorphic

virus per se. In 1990 however, the first family of polymorphic viruses appeared: the CHAMELEON viruses (or V2P) which were developped by Mark Washburn, were based on the CASCADE and VIENNA viruses and mutated the code of their decryption method (fig. 2). The shock they created shaked the antiviral community, since detection techniques using a fixed signature had suddenly become obsolete for this new brand of viruses.

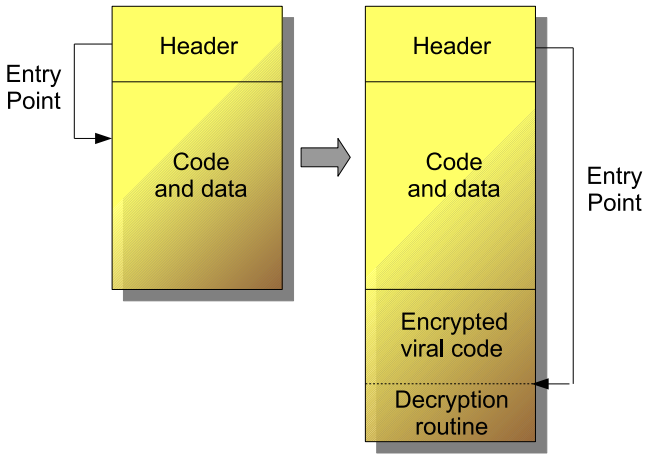

Fig. 2. Polymorphic virus infection.

The famous WHALE virus appeared the same year: it included polymorphism, stealth and armouring techniques and mutated in particular the code of its mutation function using obfuscation techniques (dead code, test repetition, redundant code, ...). Then "boards" appeared, where were shared viruses and e-zines, among which Phrack and 40Hex, and where were worked out and shared new viral techniques. Then in 1992 the first polymorphic engines appeared, like MTE, TPE, NED and DAME[1], which could be linked to the virus to produce a polymorphic variant. They were quickly followed by the first virus creation toolkits, like VCL, PS-MPC and G2[2], some of which including polymorphism features. This signalled the start of massive creation – in thousands – of simple and polymorphic viruses.

On the antiviral community side, the answer came in 1992 when Eugene Kaspersky worked out a technique now used by most antivirus products, namely *detection by code emulation*. Since one could not anymore rely on the static version of a program's code to detect a virus, the code was run in a controlled (emulated or sandboxed) environment on a given number of instructions, and periodically or in the end the affected memory was analysed to detect the (possibly partially) decrypted viral code. Indeed, and this is the base principle of metamorphism, polymorphic codes had the major drawback of always decrypting themselves into the memory into an invariant and thus detectable form. However this detection technique also has the disadvantage of being quite cpu-intensive.

Several techniques, called anti-emulation techniques, have been developped as a result by virus writers to hinder this kind of detection:

- Using unusual instructions which an emulator might not support and interpret, or similar tricks that would prevent the virus from decrypting itself correctly or that would betray the presence of an emulator.
- Inserting dead code that will loop long enough to have the emulator give up on detection, relying on the prohibitive cost of emulation (this technique is used by the BISTRO virus for instance).
- Random cancelling of decryption, thus running the viral code only a random basis.
- Entry Point Obscuring (EPO) techniques, which consist in avoiding executing the virus body at the very beginning of the host's execution, but rather executing it during the host execution or even in the end.
- Using several encryption layers.
- Decrypting and running the code chunk by chunk, some viruses decrypting and running only one instruction at a time (like the DARK PARANOID virus, in 2004).
- *Metamorphic* techniques, which transform the encrypted code.

These techniques are detailed in the literature [Fil05], [Fil07], [Ayc06]: some of these techniques are used by METAPHOR and we shall come back on them in the next section.

Finally, we state Spinellis's recent result [Spi03], which establishes the general complexity of the detection of such viruses. He shows that the problem of detecting polymorphic viruses, of bounded length, is NP-complete, by reducing to it the well-known SAT problem of satisfiability.

C. Metamorphic viruses

Metamorphic viruses are in a sense advanced polymorphic viruses: on each replication, the code to be executed completely mutates, without altering its functionality. Thus, encryption is not anymore necessary and, when used, the decryption method as well as the decrypted code of the virus are different for each new generation. Figure 3 presents a basic example of infection by a metamorphic virus, on its i^{eme} mutation: in practice, the code is often encrypted and the decryption routine is sometimes scattered among the host's code (ZMIST virus for instance).

The first metamorphic techniques made their appearance in 1997 with the TINY MUTATION COMPILER (TMC), by Ender. This virus had a compiler embedded in its body as well as its own sources in encrypted pseudocode. On execution, the virus decrypted its source code, inserted dead code, mixed up its code and data, and recompiled everything.

On the same year, Z0mbie developped his Z0MBIE'S CODE MUTATION ENGINE (ZCME), which did not use any encryption techniques but allocated a 16K buffer where it randomly copied out its instructions, linking them with each other with JMP instructions and filling the remaining space with dead code.

[1] MUTATION ENGINE (MTE) by Dark Avenger, TRIDENT POLYMORPHIC ENGINE (TPE), NUKE ENCRYPTION DEVICE (NED) and DARK ANGEL'S MULTIPLE ENCRYPTOR (DAME).

[2] VIRUS CONSTRUCTION LAB (VCL), PHALCON/SKISM MASS-PRODUCED CODE GENERATOR (PS-MPC) and PHALCON/SKISM'S G2 VIRUS GENERATOR (G2).

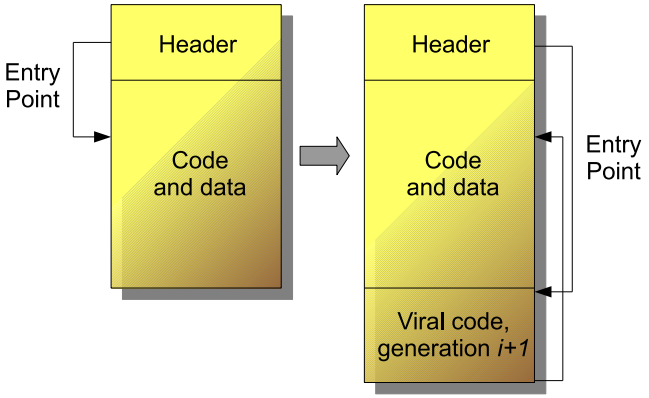

Fig. 3. Metamorphic virus infection on generation i.

In 1998, Vecna implemented MISS LEXOTAN, which disassembled itself, added some dead code and modified the form of its instructions, in a computational way most particularly (see later). To create dead code, it inserted most notably meta-instructions `XOR ebp, imm`, with no effect, but which defined which registers were used and thus should not be modified. He also implemented REGSWAP later, which shuffled the registers. Here is an excerpt from LEXOTAN:

```
xor  bp, __fill + __ax + __bx + __flag
   ; tells that registers ax, bx and
   ; the FLAGS are used by the code
add  ax, bx
xor  bp, __fill + __ax + __flag
add  ax, 10h
push ax
mov  ax, 0
```

After transformation, this code may look like this, with no jumps:

```
xor  bp, __fill + __ax + __bx + __flag
mov  dx, bx
xor  cx, cx                              ;
push cx                                  ; dead code
add  ax, dx
pop  cx                                  ;
xor  bp, __fill + __ax + __flag
mov  bx, 34h
push bx
mov  bx, ffCCh
pop  ax
add  ax, bx
xor  bx, bx
push ax
mov  bx, 10h
sub  ax, ax
```

In 2000, the BADBOY, ZMORPH, EVOL, ZPERM, BISTRO and ZMIST viruses enter the growing list of metamorphic viruses, using more or less complex techniques. ZPERM most notably introduces the REAL PERMUTATION ENGINE (RPME), which can be linked to other viruses, and enables random permutation of the virus code, with insertion of dead code and branching using `JMP` instructions.

ZMIST, by Z0mbie, is more particularly one of the most evolved (and most stable) metamorphic viruses until now. It uses the following techniques:

- Entry Point Obscuring (EPO).
- Metamorphism:
  - (Random) encryption with two keys.
  - Code integration: it's the first virus to use this method which consists in scattering the decryptor's code directly among the host's code, which makes the virus hard to detect and hard to disinfect. The MISTFALL engine is used for this technique.
  - Permutations (it uses ZPERM's RPME engine).
  - Dead code, generated by the EXECUTABLE TRASH GENERATOR (ETG).
  - Syntaxic modification of instructions.

The virus is analysed, along with other polymorphic and metamorphic viruses, in [Szo05].

Finally, METAPHOR, by Mental Driller, appears in 2002 and is certainly the most advanced metamorphic virus until today. It may infect both `Elf` (on Linux) and `PE` (on Windows) files, on the local file system and on mounted partitions (in Linux) or shared folders (in Windows).

Let's also mention the recent development of Java and MSIL[3] viruses, which are platform-independent. .NET assemblies infection is simplified by the presence of assembler libraries (`System.Reflection.Emit` namespace) and both technologies enclose standard high level cryptography libraries. Only one metamorphic MSIL virus is known as of today, —Gastropod—, and there still are very few Java and MSIL viruses. But given the ubiquity of both technologies, these viruses might well represent a threat in the near future for any platform that supports them.

The rapid evolution of viral techniques towards first polymorphic and then metamorphic techniques motivated the working out of new detection techniques, based on emulation and behaviour analysis allowing to identify suspect behaviours. However in the same time, they revealed two limitations that are inherent to antiviral defence and benefit virus writers. First, the efficiency of these methods relies on an often prohibitive complexity when iterated on a high number of files: defence cannot monopolize resources of the protected system whereas attack has a priori no cost nor time limits. Moreover a delay of a few hours or of a day is long enough for a well-implemented virus to spread on a very large scale, hence the interest for virus writers to complicate as much as possible analysis of their viruses. Although these weaknesses, combined with advanced metamorphic techniques, are not used yet in a lot of viruses (or these very viruses are often buggy and easily detected and stopped), they define a new age of viral detection, in which current protection methods will be thoroughly obsolete.

III. STUDY OF A METAMORPHIC VIRUS: METAPHOR

The cross-platform metamorphic virus METAPHOR[4] was written in 2002 by The Mental Driller and was the second highly advanced metamorphic virus (with ZMIST), and the first ever polymorphic, and metamorphic, Linux virus. It was published in 29A's magazine [MD02]: its sources can be found

[3] i.e. targetting .NET assemblies.

[4] METAPHOR is also known as SIMILE or ETAP.

on VX Heavens [MDa]. It uses highly advanced metamorphic techniques which combine the majority of the techniques used by its predecessors. They're used along with anti-heuristic and anti-emulation techniques.

A. Overview of the techniques used by METAPHOR

Here are the main polymorphic techniques used by METAPHOR:

- XOR / SUB / ADD encryption, with random key, or no encryption at all;
- *Branching technique*;
- Pseudo-Random Index Decryption (*PRIDE*);
- Metamorphic techniques:
  - Dead code insertion;
  - Instruction modification;
  - Random modification and permutation of registers;
  - Code permutation;
  - Mutation of the memory access profile.

B. Polymorphism in METAPHOR

1) Encryption techniques: First let's describe the miscellaneous encryption techniques which are commonly used in polymorphic viruses (see [Mid99] for some more details and for examples).

a) Basic encryption: The most simple ones, as well as the most common ones, use a mere XOR (as shown in the example), ADD or SUB encryption, with a key which is randomly generated on each replication and which is stored inside the virus data or directly inside the decryption method. The following code is a basic example of such an encryption:

```
  mov  esi, offset enc_code_start
                          ; start of encrypted code
  mov  edi, esi           ; start of decrypted code
  mov  ecx, (offset enc_code_end -
        offset enc_code_start) / 4  ; size in dwords
  mov  ebx, 6B3C728Ah       ; encryption key
start:
  lodsd                   ; load a dword in eax
  xor  eax, ebx           ; decrypt it
  stosd                   ; save it
  loop start
end:
  jmp  enc_code_start
```

b) Sliding key encryption: One drawback of the previous technique is that, once the key has been chosen, each character is encrypted in a unique way. Thus the sliding key encryption updates the key as the decryption progresses, either in a fixed way or for instance with the last encrypted character. For instance, the previous code could be modified in the following way:

```
  ...
  xor  eax, ebx
  add  ebx, eax
  ...
```

c) Flow encryption: This method uses a key to generate a keystream of the same size as the data to encrypt. For instance the generation of this pseudo-random keystream might use one or several linear feedback shift registers (LFSR, see section III-D1). Some basic implementations simply duplicate as much as needed the input key. The previous code can be easily adapted to this technique, in the case of a single register (`lfsr_init` initializes the register, and `lfsr_next` shifts the 32bits register, thus generating a new key):

```
  ...
  mov  ebx, 6B3C728Ah
  call lfsr_init ; init the register from the key
start:
  lodsd
  call lfsr_next ; ebx := 4 new bytes from keystream
  xor eax, ebx
  ...
```

d) Encryption with permutation: The input data is simply permutated. Permutation can occur on the scale of the whole data, of chunks of bytes (of fixed or variable length), or even of each byte (with the ROR instruction for instance).

e) Multiple encryption: Several encryption techniques are sequentially applied.

f) Random key encryption: The data is encrypted with a random key which is not stored for future decryption. Upon execution, the key (as well as the encryption technique) can only be recovered by brute force attack or cryptanalysis. This technique disables any code emulation analysis. The size of the key space (and possibly its properties) allows to control over the decryption time. This technique was introduced by DarkMan in 1999 in his RANDOM DECODING ALGORITHM ENGINE (RDAE), which implemented several encryption techniques without storing the key: only the code's CRC32 checksum was stored. These techniques are detailed in [BF07], [Kha07].

g) Code-dependent encryption: The binary code itself is used as the encryption key, or a combination of the code and a random key. This technique was usually used to ensure that the code had not been modified – during an antiviral analysis (where the code could be patched to disable some anti-debugging techniques).

Upon decryption, the virus needs access to the decryption key(s). This key is usually directly stored in the program: inside the decryption procedure, inside the virus data or simply related to the host program (for instance the key can be the host's filename). The case of RDA is different since the key is retrieved by brute force. However other scenarios exist where the key isn't stored in the code but is inferred from the environment. This technique is called environmental key generation [RS98]. Here are some examples:

- The key is forged from the local environment. For instance, the key is the hard disk serial number, combined with some random value stored in the code, etc.
- The key depends on activation factors. For instance, it depends on the current date and will only be valid during some predetermined period. In consequence, the virus itself will be disabled outside the valid periods.
- The key is stored on a web server, a news server, etc.

The most advanced implementation of this technique is the proof of concept BRADLEY virus [Fil04]. It uses several encryption layers, whose keys are retrieved from the environment. The interest of such viruses from their writer's point of view, is that they can restrict the activity of their

virus geographically as well as temporally. Filiol also shows in [Fil04] that, if the key is unknown during the analysis, the cryptanalysis's complexity is exponential (in BRADLEY's case).

As for METAPHOR, it encrypts its code with an initial probability of $15/16$ and uses an encryption method (with random key) of type XOR, ADD or SUB.

However, METAPHOR's decryption method is much more interesting. It uses techniques that The Mental Driller had already implemented into the TUAREG engine (TAMELESS UNPREDICTABLE ANARCHIC RELENTLESS ENCRYPTION GENERATOR) and that he describes in another issue of 29A's magazine [MD00], [MDb]. This engine combined most notably two novel techniques, with an anti-heuristic purpose, which also took part in the mutation of the decryption routine. Both techniques, the branching technique and the *PRIDE* technique, are used in METAPHOR. Finally, an EPO technique is used to give control to the decryption routine: METAPHOR changes all calls to the `exit` function into calls to this routine. Thus, the virus only gains control after execution of the program, which contributes to its stealth and protects it from the detection by emulation.

2) Branching technique: A basic decryption method has a structure that often follows a common template which will trigger an alarm in any heuristic engine, as one can see with the examples from last section. Thus the branching technique allows to simulate as much as possible the behaviour of an innocuous program. Such programs will usually sequentially test several conditions and, depending on the result, finally branch on distinct paths. This technique therefore creates several random tests, until a given recursivity level, that will define an execution tree with leaves representing distinct ways to decrypt the code. Figure 4 describes the execution tree for a maximum depth of recursivity equal to 2: each terminal branch has its own decryption code, though the final result is the same, whatever branch is taken. Thus for a depth of recursivity equal to n, 2^n decryption branches are generated.

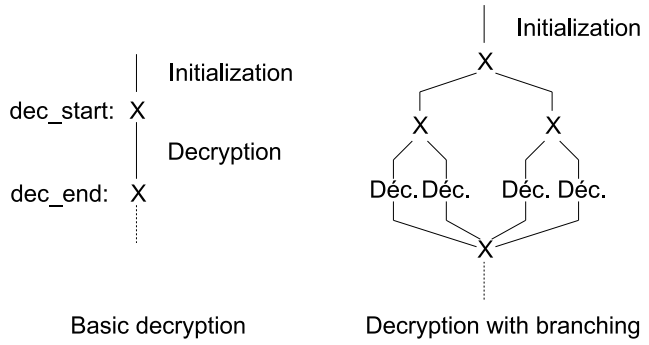

Fig. 4. Execution tree with and without branching technique.

Furthermore, to reduce the risk of an heuristic alert upon execution of a branch, terminal branches do not contain a decryption loop but only its body: once the body is executed, a jump is made to any one of the previous nodes in order to carry on decryption. Thus, upon execution, each branch makes the same computation and all branches are shared and alternatively used to implement the decryption loop. Here is the C algorithm used in METAPHOR (ll. 15750 – 16075):

```
void do_branching () {
  int i;

  make_branch ();
  for (i = 0; i < cnt_partial_jumps; i++)
    // redirect each jump at a random node
    complete_partial_jump (partial_jumps[i],
            get_random_node ());
}

void make_branch () {
  int jmp;

  if (recLevel >= maxLevel) {   // maximum depth?
    insert_code ();             // decryption code
    build_instr (OP_CMP, REG_ECX, code_len);
                                // CMP ecx, code_len
    jmp = insert_partial_jump (OP_JNZ);   // JNZ <?>
    partial_jumps [cnt_partial_jumps ++] = jmp;
                    // update the target in the end
    ...             // call the decrypted code
    return;
  }

  recLevel ++;
  add_node (insert_label ()); // save the new branch

  if (random_boolean ()) {    // test CMP or TEST?
    int reg, val, op;
    reg = get_random_register ();
    val = 0x80000000 | (random () & 0x3fffffff);
                              // 0x8XYYYYYY (X < 4)
    build_instr (OP_CMP, reg, val); // CMP reg, val
    op = OP_JB + (random () & 0x5); // JB/JA/JBE/JAE
    jmp = build_partial_jump (op);  // partial jump
  } else {
    int reg, val, op;
    reg = get_random_register ();
    val = 0x1 << (random () & 0x1f); // 2^X (X < 32)
    build_instr (OP_TEST, reg, val); // TEST reg,val
    op  = OP_JZ + (random () & 0x1); // JZ or JNZ
    jmp = build_partial_jump (op);   // partial jump
  }

  /* first branch: */
  make_branch ();
  complete_partial_jump (jmp, insert_label ());

  /* alternative branch: */
  make_branch ();

  recLevel --;
}
```

And here is an example code it could yield, for a recursivity depth of 2:

```
br0:
  cmp  reg1, val1       ; reg1, random register
                        ; val1 = 8XYYYYYYh (X < 4)
  jcc  alt0             ; jcc = jb / ja / jbe / jae
br1:
  test reg2, val2       ; reg2, random register
                        ; val2 = 2^X (X < 32)
  jcc  alt1             ; jcc = jz / jnz
  <Decryption code 1>
  cmp  ecx, code_len
  jnz  br1'
  ...
alt1:
  <Decryption code 2>
  cmp  ecx, code_len
```

```
  jnz  br1
  ...
alt0:
br1':
  cmp  reg3, val3
  jcc  alt1'
  <Decryption code 3>
  cmp  ecx, code_len
  jnz  br0
  ...
alt1':
  <Decryption code 4>
  cmp  ecx, code_len
  jnz  br0
  ...
```

As this will be detailed in section III-C about metamorphic techniques, this code is actually an intermediate representation of the final code: once the code has been created, METAPHOR generates the final `x86` code by rewriting each instruction into an equivalent sequence of instructions and by randomly inserting dead code.

3) PRIDE technique (Pseudo-Random Index DEcryption): The purpose of this technique is also to protect the virus from a heuristic detection. Indeed, even with the modification of the execution tree of the decryption procedure, it follows the following common mechanism (for a basic encryption):

1) `data :=` address of a buffer inside the data section of the virus.
2) Sequentially read `data` and create a new buffer, which will contain the decrypted data.
3) Give control to the new decrypted code.

The second stage of this procedure, which consists in sequentially reading a sequence of 1000 bytes or more in memory, presents a high risk of heuristic alert. Therefore, the `PRIDE` technique consists in decrypting `data` in a pseudo-random order and not anymore in a sequential order. Byte 10 will be decrypted, then byte 23, then byte 7, then byte 48, and so on. This memory access profile is much closer to an innocuous application's memory access profile. In the same time, this technique reinforces the polymorphism of the decryption code.

Here is the algorithm used for the `PRIDE` technique (ll. 15570 – 15652 and 15827 – 15984). `size_of_data` is the size of the data to be encrypted, rounded up to a power of 2. First the algorithm initializes its variables:

```
pride_start = (size_of_data - 4) & random ();
                    // aligned on a dword boundary
pride_step = (size_of_data - 8) & random ();
                    // aligned on a qword boundary
pride_key = get_random_key ();
```

Then it initializes the registers to be used by the decryption routine: `CR`, `IR` and `BR`. `CR` is the counter register and contains the sequential decryption index, `IR` is the index register and contains the pseudo-random decryption index (`XOR`'ed actually with `CR`), `BR` is the buffer register used as temporary storage for encrypted data. Compared to the decryption routine in section III-B1, we have: `CR` ≡ `ecx`, `IR` ≡ `esi` ≡ `edi` and `BR` ≡ `eax`. The following code is written at the beginning of the decryption routine:

```
  MOV CR, pride_start
  MOV IR, val  ; val = (size_of_data - 4) & random()
  MOV BR, val' ; val' = random()
```

Finally, when the decryption routine's body must be generated (call to `insert_code` inside the `make_branch` method), the algorithm writes:

```
  PUSH IR
  XOR  IR, CR
  MOV  BR, [IR + source]
  XOR  BR, key          ; or ADD BR, +/- key
                        ; or nothing (no decryption)
  ADD  IR, dest
  MOV [IR], BR          ; write the decrypted dword
  POP  IR
  ADD  CR, val          ; CR += [4;7]
  AND  CR, val'         ; val'  = ((random() &
       ;  ~size_of_data) | (size_of_data-4)) & -4
       ; (-> CR := (CR % size_of_code) & FFFFFFFCh)
  ADD  IR, pride_step
  AND  IR, val''        ; val'' = ((random() &
       ;    ~size_of_data) | (size_of_data-1)) & -1
       ;        (-> IR := IR % size_of_code)
  CMP  CR, pride_start
  JNZ  <?>              ; jump at a random branch
```

Furthermore, the last instructions which update registers `CR` and `IR` (`ADD CR, val` and `AND CR, val'` for the `CR` register) are permutated with each other, with the obvious requirement that the `AND` instruction is executed before its `ADD` counterpart. Also, as we can see, `pride_step` determines the "order" of decryption: when equal to 0, it simply corresponds to a sequential decryption (starting at index (`IR ^ pride_start`)).

This ends the study of polymorphic techniques in METAPHOR. Both techniques we described mainly aim to impede any detection by emulation: however, in a sense, they also have a mutation role, not anymore in the form but in the behaviour. This proximity between signatures used for form analysis and signatures used for behaviour analysis is studied into more details in [Fil07].

C. Metamorphism in METAPHOR

METAPHOR's metamorphic engine takes up 70% of the source code (11000 lines in all), the remaining 30% accounting for the infection routines (20%) and the decryptor's creation routine (10%). This proportion isn't uncommon: some metamorphic viruses devote up to 90% of their code to their metamorphic engine. The engine is used to mutate the virus body (more precisely the part to be encrypted) as well as the decryptor itself.

The engine works according to the following template, which The Mental Driller calls humorously *accordion model*:

1) Disassembly / Depermutation
2) Compression
3) Permutation
4) Expansion
5) Reassembly

One particularity of this engine, which conceptually differentiates it from its predecessors, is the use of an intermediate representation which allows to dissociate from the complexity of the underlying processor's instruction set and to simplify the miscellaneous transformations and the code manipulation and creation. For instance, equivalences between `x86` instructions

are deferred until the reassembly stage, jumps at other code instructions are translated into a pointer perspective (that are much easier to manipulate, compared to offsets), etc.

1) Description of the pseudo-instruction set: METAPHOR uses a limited instruction set. It only considers instructions that are actually used by the code. Since this intermediate representation isn't used when modifying the host code, this restriction is natural. This instruction set is organized as follows:

- Base instructions with 2 operands: `ADD`, `OR`, `AND`, `SUB`, `XOR`, `CMP`, `MOV` and `TEST`.
- Base instructions with 1 operand: `PUSH`, `POP`, `Jcc`, `NOT`, `NEG`, `CALL` and `JMP`.
- Other instructions: `SHIFT`, `MOVZX`, `LEA`, `RET` and `NOT`.
- Macro-instructions:
  - `APICALL_BEGIN`, `APICALL_END`, `APICALL_STORE`, which represent the instruction sequences which are used when calling a Windows API (in the case of a `PE` infection): since the registers to be used by these calls are predefined, these macro-instructions ensure their protection from register swapping transformations.
  - `SET_WEIGHT` which is used for "genetic" evolution (see section III-D2).
  - `LINUX_GETPARAMS`, which is similar to `APICALL_BEGIN`, and represents the loading of parameters into general purpose registers.
  - `LINUX_SYSCALL` which represents a syscall (`int 80h` – used to call a system function); and `LINUX_SYSCALL_STORE` which represents a syscall followed by the result's saving.
- Instructions used only by internal operations: `Mov Mem, Mem`, used during the compression stage, and `INC` and `LITERAL_BYTE` (unencoded byte to be inserted as it is) which are used during the reassembly stage.

The opcode choices are motivated by the equivalent `x86` opcode organization and by the sake of simplifying the manipulation of instructions and the coding of transformations. In particular, for the first type of opcodes, the opcode itself (for instance `ADD`) is encoded into bits `6..3`, and the operand types into bits `2..0` and `7`: bit `7` specifies whether the operands are 8 bits (for instance `mov al, 12h`) or 32 bits (for instance `mov eax, 12h`) whereas bits `2..0` specify the type of operands (`Reg, Imm`, etc.).

Finally, a pseudo-instruction is encoded in 16 bytes:

```
XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX
OP *--------- operands --------* LM *- instr -*
```

`OP` contains the instruction opcode, on one byte. Then the operands are encoded (register index, memory address or immediate value) on the following 10 bytes. Then `LM` ("Label Mark") is a flag on 1 byte telling whether this instruction is the target of a branching instruction: when this is the case, the instruction can neither be deleted nor compressed with instructions preceding it. The last 4 bytes contain a pointer which has miscellaneous significations along the execution: during the disassembly, it contains the address of the initial `x86` instruction, during the permutation, it contains the instruction's address inside the non-permutated code, etc.

Once the virus decrypted its code, it gives control to it. After initialization of the variables and possible payload activation, it defines the form of next generation (internal organization of the code – where to put code, where to put data, etc.). Then it starts the code transformation process.

2) Disassembly: The `x86` code is first disassembled into an intermediate representation which uses the previous instruction set. This procedure loads the intermediate code into the buffer pointed by variable `InstructionTable`. It also creates an array of labels which contains all instructions which are the target of a branching instruction. In the end, the computed intermediate code has been depermutated and the inaccessible code (dead code) removed: this is actually a direct consequence from the routine's algorithm.

The `x86` code is disassembled by *following the execution flow*. The algorithm uses an array, `FutureLabelTable`, which contains instructions which are waiting for their disassembly (namely these are the targets of conditional jumps and direct calls). Here is the algorithm:

- If the current instruction was already disassembled, simply add a `JMP` instruction which points at the disassembled instruction. Then carry on disassembly with an instruction from `FutureLabelTable` (if any) or terminate.
- Otherwise:
  1) If previous instructions did point at the current instruction, update them in order to point at the new disassembled instruction.
  2) Create the new pseudo-instruction. The following cases are more specifically distinguished:
     - `INC` and `DEC` instructions are replaced by their `ADD` and `SUB` counterparts: during the reassembly stage, the opposite transformation will be applied (or not).
     - If this is a `JMP` instruction: either its target was already disassembled and we simply insert a `JMP` instruction pointing at that instruction (by creating a label), or the target has not been disassembled yet and we insert a mere `NOP`.
     - If the instruction is a conditional jump or a direct call: if the target has been disassembled yet, add it to the wait array `FutureLabelTable`. Then insert the corresponding branching instruction (pointing at the disassembled target, if it exists, or at the `x86` target instruction).
  3) Finally, if this was a `JMP` instruction whose target had not been disassembled yet, continue with this target. If the target was already disassembled, or the instruction is a `RET`, continue with an instruction from `FutureLabelTable` (if any). Otherwise continue with the next instruction.

Code permutation is carried out, as we will see, using unconditional jumps (no "opaque predicates" or similar tricks): during the disassembly, the `JMP` instruction used to join two permutated blocks is replaced by a `NOP` instruction and the

disassembly continues with the new block. Given that the pseudo-code is built in a linear way, its final shape will be that of the depermutated code. Similarly, inaccessible code that was inserted will never be disassembled.

3) Compression: After disassembly and depermutation, the generated pseudocode is compressed. This cancels the expansion effects of the previous generations, since the compression rules are exactly the inverse of the expansion rules. There are five kinds of rules:

1) `Instr -> Instr` rules:
   - `XOR Reg, -1 -> NOT Reg`
   - `SUB Reg, Imm -> ADD Reg, -Imm`
   - `OR Reg, 0 -> NOP`
   - `AND Reg, Reg -> CMP Reg, 0`
   - `...`
2) `Instr / Instr -> Instr` rules:
   - `PUSH Imm / POP Reg -> MOV Reg, Imm`
   - `MOV Mem, Imm / PUSH Mem -> PUSH Imm`
   - `OP Mem, Imm / OP Mem, Imm2 -> OP Mem, (Imm OP Imm2)`
   - `NOT Reg / NEG Reg -> ADD Reg, 1`
   - `TEST X, Y / !=Jcc -> NOP`
   - `Jcc @xxx / !Jcc @xxx -> JMP @xxx`
   - `...`
3) `Instr / Instr / Instr -> Instr` rules:
   - `MOV Mem, Reg / OP Mem, Reg2 / Mov Reg, Mem -> OP Reg, Reg2`
   - `...`
4) `Instr / Instr / Instr -> Instr / Instr` rules:
   - `MOV Mem, Reg / TEST Mem, Reg2 / Jcc @xxx -> TEST Reg, Reg2 / Jcc @xxx`
   - `...`
5) Macro-operations identification rules:
   - `PUSH eax / PUSH ecx / PUSH edx -> APICALL_BEGIN`
   - `POP edx / POP ecx / POP eax -> APICALL_END`
   - `POP edx / POP ecx / POP ebx / POP eax -> LINUX_GETPARAMS`
   - `CALL Mem / MOV Mem2, eax -> CALL Mem / APICALL_STORE Mem2`
   - `INT 80h -> LINUX_SYSCALL`
   - `INT 80h / MOV Mem, eax -> LINUX_SYSCALL_STORE`
   - `PUSH Reg1 / MOV Reg1, Imm1 / MOV Reg2, Imm2 / MOV Mem, Reg2 / POP Reg1 -> SET_WEIGHT Mem, Imm1, Reg1, Reg2`

Notation `!=Jcc` denotes "any opcode that is not a conditional jump" and the notation `!Jcc` denotes the inverse of the last `Jcc` (for instance, `JA` and `JBE`). Furthermore, some of these rules might not be verified in the general case, but they are in the case of METAPHOR's code.

The algorithm is simple. It compresses the code as much as possible. When it looks up the next instruction, it skips any `NOP` instruction that is not the target of jump or a call (flag `LM` is set). As long as it did not reach the end of the code, it tries to compress chunks of one, two or three instructions starting from the current instruction: if a compression occurs, it makes a three instructions step-back and continues. This allows to take into account any new reduction opportunity that might have appeared with an instruction created by the last reduction. For the sake of simplicity, instructions that are deleted are simply replaced by `NOP` instructions. In the end, the algorithm identifies all sequences of instructions that correspond to macro-instructions (`APICALL_*`, `LINUX_SYSCALL*`, `LINUX_GETPARAMS`, `SET_WEIGHT`) and replaces them accordingly. Also note that, for a reduction – of any type – to occur, no instruction, except the first one, shall be the target of a jump (flag `LM`).

The algorithm also allows to reduce sequences of operations into a unique operation. For instance, `ADD Reg, X / SUB Reg, Y` will be reduced into `ADD Reg, (X - Y)`: these decompositions are created during the expansion. Finally, when a `Jcc` instruction is replaced by a `JMP` instruction, the following code is deleted (`NOP`ed) until reaching a label (instruction with `LM = 1`).

Here is an example of compression (this code represents a basic decryption routine):

```
  test esi, val1       |     nop
  mov  [Mem], val2     |     mov  esi, (val2 + val3)
  add  [Mem], val3     |     nop
  push [Mem]           |     nop
  pop  esi             |     nop
  mov  [Mem2], esi     |     mov  edi, esi
  and  esi, -1         |     nop
  push [Mem2]          |     nop
  pop  edi             |     nop
  push val4            |     mov  ecx, val4
  pop  [Mem3]          |     nop
  or   [Mem3], 0       |     nop
  mov  ecx, [Mem3]     |     nop
  mov  ebx, val5       |     mov  ebx, val5 XOR val6
  xor  ebx, val6       |     nop
label:                 |
  push [esi]           |     mov  eax, [esi]
  or   esi, 0          |     nop
  pop  eax             |     nop
  mov  [Mem4], eax     |==>  xor  eax, ebx
  push [Mem4]          |     nop
  pop  [Mem5]          |     nop
  xor  [Mem5], ebx     |     nop
  mov  eax, [Mem5]     |     nop
  mov  [Mem6], eax     |     mov  [edi], eax
  push [Mem6]          |     nop
  pop  [edi]           |     nop
  not  esi             |     add  esi, 4
  neg  esi             |     nop
  add  esi, 3          |     nop
  sub  edi, 0          |     nop
  add  edi, 4          |     add  edi, 4
  mov  [Mem10], 4      |     sub  ecx, 4
  and  [Mem10], -1     |     nop
  add  ecx, [Mem10]    |     nop
  mov  [Mem11], ecx    |     cmp  ecx, 0
  sub  [Mem11], 5      |     jnz  label
  add  [Mem11], 5      |     nop
  jnz  label           |     nop
```

4) Variable reorganization: METAPHOR aims to mutate at the semantic level (instructions expansion / compression) and at the code level (permutation) as well as at the code behaviour level. We already mentionned previously that it was mutating the internal organization of the viral code. When the virus gains control, it allocates into memory a space of

(340000h + X) bytes, where X is a random value between 0h and 01F000h. This space is then organized into 5 sections (see figure 5):

- Section Code contains the decrypted x86 code.
- Section Buffers contains the miscellaneous arrays and buffers used by the virus.
- Section Data contains the virus global variables.
- Section Disasm contains the disassembled code and then the result of the expansion of the permutated code. When creating the decryption routine, it will contain its pseudocode as well as the reassembled code.
- La section Disasm2 is first used as a buffer, then it contains the result of the permutation of the compressed pseudocode, and finally it contains the reassembled code.

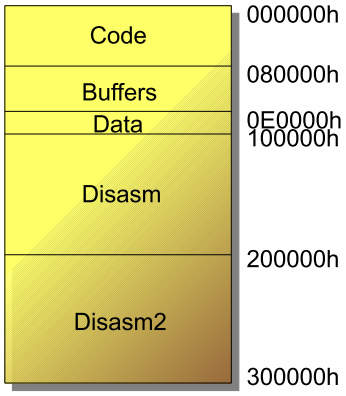

Fig. 5. METAPHOR's memory organization (generation 0).

Before starting the mutation and replication process, sections are randomly permutated and each section is shifted by a random value between 0h and 7FFFh. In the end, the maximum required size (into memory) is: 300000h + 5 * 7FFFh = 340000h. Thus, upon execution, METAPHOR never has a unique memory access profile.

The virus contains about 200 global variables, each of these variables being allocated 8 bytes inside the Data section. These variables are accessed by their offset in that section. A register is specifically assigned, which isn't modified during the virus execution, and which contains that section's address. During generation 0, this base register is ebp. Thus, to access to the contents of variable InstructionTable, which is at offset 10h of the Data section, one uses:

```
mov eax, [ebp + 10h]
```

Given that this register (ebp) is strictly reserved to data access, it is sufficient to spot all instructions that use it to identify read and write accesses to a variable and to list these very variables. Method IdentifyVariables does this job and replaces in each one of those instructions the offset by the index of the associated variable. Then the variables are shuffled: their organization inside the Data section is thus completely modified. Then, during reassembly, when an instruction uses one of these variables, the instruction is updated to contain the new base register (initially ebp) and the new offset of the referenced variable.

Thus the memory access profile is modified. This kind of transformation isn't however taken to extremes. For instance, the code often reads the contents of pseudo-instructions, as in the following code excerpt (where esi and edi contain pseudo-instructions addresses):

```
mov     ecx, [esi+1]  ; Get the value in ECX
mov     eax, [esi]
add     esi, 5
and     eax, 7        ; Get the register in EAX
mov     [edi+1], eax  ; Set the register
mov     [edi+7], ecx  ; Set the value
```

This kind of access can be profiled, since the internal organization of an instruction does not mutate. However The Mental Driller could have taken memory access profile mutation to extremes by modifying this very internal organization of pseudo-instructions. Given the massive use of instructions accessing the contents of these pseudo-instructions, impact would have been even stronger, even though the mutation of the organization of pseudo-instructions is quite limited (might we add a few padding bytes to increase mutation possibilities).

Let's note that, in this transformation's implementation, variables are aligned on 8 bytes boundaries so that they can be randomly positionned on any one of the first 4 bytes: finally, only 4 bytes are used by a variable.

5) Permutation: Once the compression is over, the engine permutates the code by splitting it into blocks of random sizes, between F0h and 1E0h. When doing the splitting, the following breaks are avoided:

- between a CALL instruction and the associated APICALL_STORE instruction;
- before a JMP or a RET instruction, to avoid two consecutive jumps;
- before a JMP or a Jcc instruction, in order for the compression process to correctly compress any Jcc + JMP or CMP/TEST + Jcc instructions.

Once the code blocks have been computed and shuffled, the new code is built (and its address saved into variable PermutationResult). A jump at the first code block is inserted at the very beginning of the code and the code blocks are linked with each other using JMP instructions, except in the following cases:

- The target block directly follows the current block.
- The block's last instruction is an unconditional jump or a return instruction.

The final result shall look like:

```
  jmp @block1
@block4:
  ;-------------;
  ;   block 4   ;
  ;-------------; (ends with a ret)
@block2:
  ;-------------;
  ;   block 2   ;
  ;-------------;
@block3:
  ;-------------;
  ;   block 3   ;
  ;-------------;
  jmp @block4
@block1:
  ;-------------;
```

```
;   block 1   ;
;-------------;
jmp @block2
```

6) Expansion: The expansion stage consists in applying the inverse rules from the compression stage. This method is called on the virus compressed pseudocode and, later, on the decryption routine's code.

The first step consists in randomly modifying the used registers. A bijective transformation is applied, which takes into account the following requirements:

- No register should of course be transformed in ESP.
- The base register (initially EBP) used to store the Data section's address (see section III-C4) should not be any of EAX, ECX or EDX (which are used by system calls).
- The 8 bits register used by 8 bits operations in the code must be related to a general purpose register (EAX, EBX, ECX or EDX).

Then, the expansion can start: it will update registers as well as accesses to the virus' global variables. The expansion's result is stored in variable ExpansionResult. To control the size of expansion, a maximum level of recursivity is first chosen: it cannot be larger than 3. Then, for each instruction, we increment the recursivity level and we randomly transform it, by using the inverse compression rules. Intermediate instructions which are generated are also transformed. NOP instructions are ignored in the compressed code (to avoid an uncontrolled increase of size, after several generations).

When an instruction is generated, which uses a *temporary memory address*, this memory address points at the Data section and should not have been allocated for the virus global variables nor by any previous instruction in the current expansion chain. The VarMarksTable array is used to mark which addresses have been allocated. As for global variables, the allocated address is randomly aligned on one of the first 4 bytes. However, this is different in the case of the decryption routine since the memory has not been allocated yet (with a call to malloc): the space to be used by intermediate operations is then the data section that was allocated inside the host file for the decryption operations.

When an instruction uses an *immediate value*, this value is computationally decomposed into a sequence of operations that finally yield the expected immediate value. This expansion is managed by method Xp_MakeComposedOPImm. It uses operators ADD, AND, OR and XOR (the SUB operator is randomly generated when transforming ADD instructions). Here is for instance the algorithm used to generate a MOV Dest, Imm instruction:

```
int v1 = random (), v2 = random ();

choose randomly among:
* MOV Dest, v1
  ADD Dest, Imm - v1

* MOV Dest, v1 & Imm
  OR  Dest, ((v2 & Imm) ^ (v1 & Imm)) | (v2 & Imm)

* MOV Dest, (v2 & ~v1) | Imm
  AND Dest, v1 | Imm

* MOV Dest, ~v1 | Imm
  AND Dest, v1 | Imm

* MOV Dest, v1
  XOR Dest, v1 ^ Imm

* MOV Dest, Imm
```

In addition, *dead code* is inserted, with probability 1/16, after each expansion of an instruction of the compressed code (if this instruction's opcode was a CMP, TEST, CALL or APICALL_STORE, a mere NOP is inserted):

- Instructions that do nothing, like:
  ```
  MOV Reg, Reg
  ADD Reg, 0
  AND Reg, -1
  ...
  NOP
  ```
- Tests that always fail, like:
  `CMP Reg, Reg / JNZ [RandomLabel]`
- Useless x86 instructions: STC, CLC.

7) Reassembly: The last stage consists in assembling the pseudo-code into valid x86 code. When several translations are possible, the algorithm chooses one at random. Also, short jumps and long jumps are randomly used (when a short jump is possible), and jumps at subsequent addresses are stored in array JmpRelocationTable, in order to be updated in the end. After completion of this stage, the code is ready for encryption and copy out in the host.

D. Randomness techniques

1) Pseudo-Random Numbers Generator (PRNG): METAPHOR makes a heavy use of random numbers. It uses its own pseudo-random numbers generator, with two seeds, seed1 and seed2, which are initialized depending on the UNIX date for seed1 and on the code's first bytes for seed2. Then a random value is generated using the following algorithm (ror_X denotes right rotation by X bits):

```
int random () {
  seed1 ^= (seed2 + ror_13 (seed1 + seed2));
  seed2 = (seed1 + ror_17(seed2)) ^ (seed1 + seed2);
  return seed1 + ror_17 (seed1 ^ seed2);
}
```

Though this may not be obvious at first sight, the second seed is very weak, given furthermore that it is initialized depending on the code's first bytes which have a low randomness: thus we get, in the worst case, a cyclic generator of 32 pseudo-random numbers (as soon as seed2 reaches value 0x00000000 or value 0xFFFFFFFF). For a random seed2, a few tests allow to compute the PRNG's period about 40000, which is barely better that the glibc's generator (random () function), whose statistical properties are particularly weak and whose period is in the order of 30000.

Polymorphic viruses usually have their own pseudo-random generator, often of poor quality, which protects them at least from a heuristic alert due to a strong utilization of a system's PRNG. Yet, some generators exist that are quite powerful and have a small cost, but their use in polymorphic viruses is scarce. Here are some of them:

- *Linear Congruential Generator* (LCG), of which the following code is an implementation:

```
unsigned int lcg_next (void) {
  seed *= 1664525u;
  seed += 1013904223u;
  return seed;
}
```

- *Registers generaztors*, like *Xorshift* generators (the following example code comes from [Mar03]) and generators with linear feedback shift registers (LFSR):

```
/* Galois' LFSR, with taps 32 31 29 1 */
unsigned int lfsrg_next (void) {
  static unsigned int seed = time (NULL);
  int i;
  for (i = 0; i < 32; i++)     // shift 32 times
    seed = (seed >> 1) ^
        (-(signed int)(seed & 1) & 0xd0000001u);
  return seed;
}
unsigned int xorshift128_next (void) {
  /* initialization with random values */
  static unsigned int
      x = 123456789, y = 362436069,
      z = 521288629, w = 88675123;
  unsigned int t;
  t = x ^ (x << 11);
  x = y; y = z; z = w;
  return w = (w ^ (w >> 19)) ^ (t ^ (t >> 8));
}
```

2) "Genetic" techniques: METAPHOR combines genetic characteristics to its generator. Here is the principle. The virus contains some sort of genetic material which will have a tendency to favour some behaviours rather than others. On each replication, this genetic material is updated with a small random variation from the preceding material.

For instance, a gene contains the current propension of the virus to encrypt its code or not: the virus initially encrypts its code with probability 1/16. Depending on its decision, the gene will be altered in favour or in disfavour of encryption: if the virus encrypts its body, it will have next time a higher probability to encrypt again its body, and conversely. Thus after a few generations, either the code will have a strong propension to encryption, or a strong propension to absence of encryption. The propension strengh is related to the survival time (and to the number of replications) of the virus. Thus, if the virus has a strong propension to encryption, this means that most of the previous generations chose encryption and survived: this is kind of an implementation of natural selection, where viruses are preys and antiviruses are predators. Thus, let's imagine that the antivirus easily detects encrypted replications of the virus (using statistical entropy analysis for instance) but not unencrypted replications. In this case, encrypted replications will be detected before being able to replicate and increase their propension to encryption, and in the end, most of the survivors will come from unencrypted ancestors, with a high propension to no encryption.

METAPHOR contains a genetic material of 24 genes. In other words, 24 of its choices depend on its genetic history and its survival abilities. These genes are used for instance for:

- Number of files to infect: initially, only 50% are infected.
- Choice of the method of infection: position of the viral code, EPO type, type of the system calls, etc.
- Encryption of the viral code, or no encryption: initially, the code is encrypted with probability 1/16.
- Encryption method (`ADD`, `XOR`, `SUB`): initially, all methods have the same probability of being chosen.
- Decryption routine's code: form of the instructions, obfuscation type, use of anti-heuristic methods, etc.

Given that the virus does not store any information in its host other than its code, it must still be able to update its genetic material, from one generation to another. This is where `SET_WEIGHT` macro-instructions come into play: they're located on disassembly and, on reassembly, the "evolved" gene is used.

Here is the algorithm used to update the genes (function `CheckForBooleanWeight`). We notice that the genes values cannot exceed a minimal and a maximal threshold (thus the associated probability never reaches 1 or 0).

```
/*
  Returns 1 or 0, depending on the gene's contents.
*/
int query_gene (int gene) {
  int val = get_gene (gene);

  if ((random () & 0xFF) >= val) {
    // return 1 and increase propension to 1
    do {
      // minimal threshold reached?
      if (val < 0x08) return 1;
      if ((random () & 0x0F) > 0)
        // increase propension to 1:
        set_gene (gene, -- val);
    } while ((random () & 0x0F) == 0);
    // repeat with probability 1/16
    return 1;
  } else {
    do {
      // maximal threshold reached?
      if (val >= 0xF8) return 0;
      if ((random () & 0x0F) > 0)
        // increase propension to 0
        set_gene (gene, ++ val);
    } while ((random () & 0x0F) == 0);
    // repeat with probability 1/16
    return 0;
  }
}
```

For a more detailled analysis of genetic viruses, one may refer to M. Ludwig's books [Lud95], [Lud93].

E. Detection of METAPHOR

Analysis of METAPHOR comes to an end. As we saw, several advanced techniques of polymorphism and of anti-emulation / anti-heuristic protection are implemented in this virus. Nevertheless they're not taken to their extremes and thus this mutation model is still detectable, mainly because of the following "weaknesses":

- The viral code's encryption can always be identified by a statistical analysis of the code [Fil07]. Indeed, a program usually has a predefined entropy profile, which shows few variations when comparing miscellaneous executable files. Encrypted data, however, have a specific entropy profile which is much more uniform, depending on the underlying encryption system, and thus is characteristic of an encrypted content. Same goes for compressed data.

Any antivirus using this kind of analysis will most likely consider as suspect a program that contains a lot of encrypted content. However, several legitimate applications use encrypted data, for the purpose of intellectual property protection. This is the case of "packed" applications (even though malware also uses packers on a regular basis), and this is also the case of Skype for instance.

- When the virus is executed, it compresses its code into a form that is roughly the same from one generation to another, by conception: METAPHOR is therefore vulnerable to any form analysis that monitors memory. As we might have expected, this weakness can be corrected to some extent, using miscellaneous techniques that are preferably not described here but easy to find out. Another weakness is also the immutability of METAPHOR's mutation grammar.
- METAPHOR's mutation grammar is globally simple and does not use any sophisticated obfuscation tricks – this is by conception given that the virus wants to be able to revert effects of mutation. In other words, using more advanced obfuscation techniques, possibly along with the addition of metadata into the code (as is the case with MISS LEXOTAN – see section II-C), would lead to a virus, which would be much harder to detect (speaking of its mere detectability as well as of the complexity of its detectability).
- Except during decryption, METAPHOR does not protect itself from behaviour analysis.

É. Filiol studies into more details some aspects of METAPHOR in [Fil07], from a theoretical point of view, and most notably regarding the detection barrier on which METAPHOR sits astride: if it mostly inclines towards detectability, some modifications would be sufficient to have it incline towards the other side (see the POC virus PBMOT). To sum up, METAPHOR is a highly advanced virus, which could be really dangerous with a few improvements (PBMOT certainly is the most appropriate proof). Other advances, as on the field of functional polymorphism, would also give metamorphic viruses more sophisticated means of defence against detection.

IV. CONCLUSION

If polymorphic and above all metamorphic techniques described in this document enable viruses to protect themselves in a more efficient way against detection, their sophistication mostly stems from antiviral protection. For antiviral protection is in fact eternally submitted to two paradoxes:

- The more it develops, the more viruses, worms and other malwares use advanced protection techniques which get harder and harder to bypass. In a sense, it sentences itself indirectly to its own impotence (wrt these advanced techniques). Yet currently it still remains efficient, thanks to the mediocre quality of most malwares or to the complexity of the mentionned protection techniques, which discourages most malware writers.
- And secondly, if on one side the increase of RAM size and CPU speed, as well as the upcoming of multi-core processors, seem to be in favour of antiviral protection, it also enables malwares to use more and more complex techniques, without having to worry about their cost. And this is all the more true as, as we told previously, antiviruses will always be limited in time and CPU cost, unlike malwares.

Also, it should be noted that the state of the art of current metamorphic techniques (with viral protection purpose) is not representative of the threat they represent. Some antiviral experts sweep blatantly away – recently again [She07] – this threat on the pretext that it never actually proved itself except for proving its own uselessness. And as a matter of fact, the history of metamorphic viruses tends to corroborate this: there are few of them, most of which are poorly accomplished and contain critical flaws (bugs or conception flaws which make detection easy). In the same time, development of rootkit techniques draws away attention. Yet, both threats are real, with different maturities, but none of them should be overlooked. Even though the second one is mostly implemented in worms, which currently represent the most important infectious threat, and even though it is more technical than the first one, and thus within the means of more hackers.

All in all, if virus writers were a bit less "in a hurry" and refined their techniques, the antiviral community could be quickly overtaken. An advanced use of syntactic and functional polymorphism techniques, combined with advanced stealth techniques, would theoretically make the complexity of the detection problem prohibitive or even undecidable [Fil07] (POC virus PBMOT).

REFERENCES

[Ayc06] John Aycock. *Computer Viruses and Malware*. Springer, 2006.

[BF07] Philippe Beaucamps and Éric Filiol. On the possibility of practically obfuscating programs – towards a unified perspective of code protection. *Journal in Computer Virology*, 3(1), April 2007.

[Coh84] Fred Cohen. Computer viruses - theory and experiments, 1984.

[Fil04] Éric Filiol. Strong cryptography armoured computer viruses forbidding code analysis: the BRADLEY virus. In *Proceedings of the 14th EICAR conference*, May 2004.

[Fil05] Éric Filiol. *Computer viruses: from theory to applications*. Springer Verlag, 2005.

[Fil07] Éric Filiol. *Advanced viral techniques*. Springer Verlag France, 2007. An english translation is pending, due mid 2007.

[Kha07] Kharn. Exploring RDA. *.aware eZine*, 1, January 2007.

[Lud93] Mark Ludwig. *Computer Viruses, Artificial Life and Evolution*. American Eagle Publications, Inc., 1993.

[Lud95] Mark Ludwig. *The Giant Black Book of Computer Viruses*. American Eagle Publications, Inc., 1995.

[Mar03] George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14), 2003.

[MDa] The Mental Driller. METAPHOR source code. Version 1D available at: http://vx.netlux.org/src_view.php?file=metaphor1d.zip.

[MDb] The Mental Driller. TUAREG details and source code. Available in 29A#5: http://vx.org.ua/29a/29A-5.html.

[MD00] The Mental Driller. Advanced polymorphic engine construction. *29A*, 5, December 2000. Available at: http://vx.netlux.org/lib/vmd03.html.

[MD02] The Mental Driller. Metamorphism in practice or "how i made METAPHOR and what i've learnt". *29A*, 6, February 2002. Available at: http://vx.netlux.org/lib/vmd01.html.

[Mid99] MidNyte. An introduction to encryption, April 1999. Available at: http://vx.netlux.org/lib/vmn{04,05,06}.html.

[RS98] James Riordan and Bruce Schneier. Environmental key generation towards clueless agents. In *Lecture Notes In Computer Science*, volume 1419, pages 15 – 24, 1998.

[She07] Alisa Shevchenko. The evolution of self-defense technologies in malware. Available at: http://www.net-security.org/article.php?id=1028, July 2007.

[Spi03] Diomidis Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1):280 – 284, January 2003.

[Szo05] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley Professional, 2005.

Philippe Beaucamps is a PhD student at the CNRS / LORIA in Nancy, France. He works on the modelization of viral infections, and on formal and practical malware detection and protection.

Contact address: Loria, Équipe Carte, 615 rue du Jardin botanique, CS 20101, 54603 Villers-ls-Nancy Cedex France