J Comput Virol (2009) 5:199–207
DOI 10.1007/s11416-009-0121-9

ORIGINAL PAPER

Code mutation techniques by means of formal grammars and automatons

Pavel V. Zbitskiy

Received: 21 November 2008 / Revised: 23 January 2009 / Accepted: 25 March 2009 / Published online: 15 April 2009

Abstract The paper describes formalization of existing code mutation techniques widely used in a viruses (polymorphism and metamorphism) by means of formal grammars and automatons. New model of metamorphic viruses and new classification of this type of viruses are suggested. The statement about undetectable viruses of this type is proved. In that paper are shown iterative approach toward construct complex formal grammars from the simplest initial rules for building metamorphic generator. Also there are some samples of applied usage of formal grammar model. The experiment for system call tracing of some viruses and worms is described. Possibility of using system call sequences for viruses detecting is shown.

1 Introduction

The aims of this work are considering of existing models of polymorphic and metamorphic viruses and perfecting this models in keeping with really existing variations of these types of viruses. The statement of undetectable metamorphic viruses is discussed.

For the first time, formal grammars and automatons used for description of code mutation techniques in [1]. Simple polymorphic generator has been described and automaton for detect any exits from this generator has been written. Let's introduce this sample in use into formal grammar and code transformation.

Grammar $G = (N, T, P, S)$ is quad where $T = \{a, b, c, d, x, z\}$ is terminal alphabet which consists of x86 instructions. a, b, c, d is garbage instructions, x, z is decryptor instructions. $N = \{A, B, S\}$ is non-terminal alphabet. S is initial state. Symbols of non-terminal alphabet uses for rules linkage of rewriting system

$$P = \begin{cases} S \to aS|bS|cS|dS|xA \\ A \to aA|bA|cA|dA|zB \\ B \to aB|bB|cB|dB|\varepsilon \end{cases}$$

$aabcxddbazbdac$ is sample of the output generator. The [1] also provides mechanism for detect these polymorphic engine. That is a building of an appropriate automaton. Figure 1 illustrates it.

Detecting procedure works as follows: we start from initial state S and move into A when detect x instruction and move on into terminal state B when detect z instruction on automaton input. If terminal state is reached we assume that valid decryptor is detected. But false-positive matches possible if instructions garbage set is incomplete.

The [2] is an establish links between metamorphism and formal grammars by implementation of POC_PBMOT metamorphic engine. Metagrammar, which underlies POC_PBMOT, describes rules of transformation by propagation. Undetectable of POC_PBMOT is proven formally.

But [1,2] do not contains formalization of "classic" metamorphism (equivalent instructions replacement and code compression). This paper fully describes this technique by means of formal grammars and automatons.

2 Semi-metamorphic viruses

Let's consider polymorphic virus, which uses code obfuscation technique such as equivalent instruction, replaced by propagation. It means that virus contains encrypted skeleton and this skeleton uses when new virus copy is produced.

P. V. Zbitskiy (✉)
Chelyabinsk State University, 129 Bratiev Kashirinih st, Chelyabinsk, Russia
e-mail: pavel.zbitskiy@gmail.com

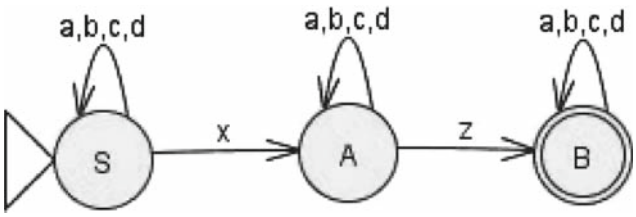

Fig. 1 Automaton for detecting simple polymorphic engine

This virus can be called "semi- metamorphic" because it uses metamorphic attributable technique and also uses encrypted part like polymorphic viruses.

The next sample illustrates it. Let we need to create following program:
push 0
push 4
call ExitWindowsEx
push 0
call ExitProcess

Now we will construct a grammar which will describe metamorphic transformation of this code. For simplicity, let addresses to use API-functions which has been already resolved.

Note $G = (N, T, P, S)$—formal grammar (base concepts of grammars and automatons can be found, for example, in [3]), T – terminal alphabet, consisted of instruction x86 processor (for instance) and a, b, c, d—some garbage instructions. N—non-terminal alphabet and S—start symbol. Let $x \oplus y$—concatenation of x and y instructions. EW—address of *ExitWindowsEx* function, EP—address of *ExitProcess* function. Thus, our grammar can be written as a set of rules:

1. $S \to aS|bS|cS|dS|(push\,0)A|(xor ebx, ebx \oplus push\ ebx)A|\ (sub\,esp, 4 \oplus mov\,[esp], 0)A$
2. $A \to aA|bA|cA|dA|(push\,4)B|(mov\,eax, N \oplus xor\ eax, < N\ xor\ 4 > \oplus push\,eax)B$
3. $B \to aB|bB|cB|dB|(call\ EW)C|(push\,\$)+10 \oplus jmp\ EW)C|\ (mov\,esi \oplus call\,esi)C$
4. $C \to aC|bC|cC|dC|(push\,0)D|(xor\ ebx, ebx \oplus push\ ebx)D|(sub\,esp, 4 \oplus mov\,[esp], 0)D$
5. $D \to aD|bD|cD|dD|(call\ EP)E|(push\,\$+11 \oplus push\ EP \oplus ret)E|(mov\,esi \oplus call\,esi)E$
6. $E \to aE|bE|cE|dE|\varepsilon$

We can see two main defects of generator built by this grammar: a big size of generator (one instruction—one rule) and "simplicity" of grammar. Let's rewrite some rules:

1. $S \to XA$
4. $C \to XD$
7. $X \to aX|bX|cX|dX|(push\,0)|(xor\ ebx, ebx \oplus push\ ebx)|\ (sub\,esp, 4 \oplus mov\,[esp], 0)$

This is non-regular grammar, but language decision problem (either $w \in L(G)$ or not, where $L(G)$—language) for this grammar can be resolved. Let's complicate model.

$G = (N, T, P, S)$—grammar, which describes algorithm of a virus. Rewriting system of this grammar is

$$P = \begin{cases} S \to A_1 A \\ A \to B_1 B \\ \quad \dots \\ X \to X_1 X \end{cases}.$$

In this case output program seems as sequence $A_1 B_1 \dots X_1$ and $A_1, B_1, \dots, X_1$ can be interpreted as internal language by which program has been written. Thereby, rules of this rewriting system sets a skeleton of program. Now we introduce a second grammar $G_1 = (N_1, T_1, P_1, S_1)$ which describes translation of skeleton symbols into a processor instructions or a block of instructions. For example,

$$P_1 = \begin{cases} A_1 \to push\,0|(xor\ ebx, ebx \oplus push\ ebx) \\ \quad \dots \\ X_1 \to (mov\ eax, EP \oplus call\ eax)|(mov\ ebx, EP \oplus call\ ebx) \end{cases}$$

Grammar G_1 describes mutation from skeleton into concrete instructions at the first step of evolution. Grammar G_2—at the second step and etc. How to complicate this model from practical point of view? First of all, we can change grammars G_1, G_2 etc by each mutation. On the one hand, it can be reached, for instance, by deleting of some rules from G_1 for getting G_2. But on the other hand, when we write our program by internal language two levels of code transformation exists: each of internal commands can be interpreted of different instructions sets of real processor and each instruction can be interpreted of these equivalents.

3 General metamorphic engine

Classic metamorphic generator can be presented as bulky, non-deterministic automata, because, all possible input characters are specified for each state of automata. Figure 2 illustrates it.

There are formal description of the automata $A = (Q, \Sigma, \delta, q_0)$. $Q = \{q_0\} \cup \{\text{x86 instructions}\}$—set of states, $\Sigma = \{\text{x86 instructions}\}$—input alphabet, $\delta : Q \times \Sigma \to Q$—the state-transition function. Input program is a some word (chain) from Σ^*. Mutation in this case is a path of automata $q_1 q_2 \dots q_n$ is visited states by processing of input word.

However, function δ describes some formal grammar, this grammar, as an automata, must be linked. Namely any sequences of instruction could be deduced from initial state q_0.

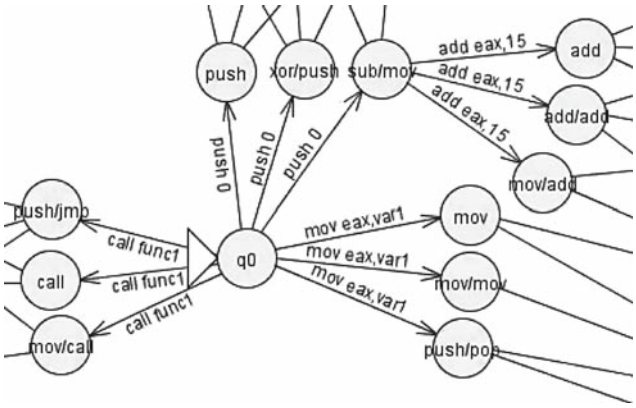

Fig. 2 Metamorphic generator modeled by automata

This grammar $G = (N, T, P, q_0)$ can be described as following. $N = \{q_0, A, B, C \ldots\}$ is non-terminal alphabet, $T = \{a_1, a_2, a_3, \ldots, z_1\} = \{\text{x86 instruction}\}$ is terminal alphabet and q_0 is start symbol. We can rewrite system in the following form:

$$P = \begin{cases} q_0 \to AA|BB|\ldots|ZZ \\ A \to BB|CC|\ldots|ZZ \\ \quad \ldots \\ Z \to AA|BB|CC|\ldots|YY \\ A \to a_1|a_2|\ldots|a_n \\ \quad \ldots \\ Z \to z_1|z_2|\ldots|z_m \end{cases}$$

This rewriting system is build up in keep with following reasons:

1. Each non-terminal symbol presents all variants of translation for some commands.
2. Double-entering of non-terminals into right side of rules provides linkage of grammar (any code sequences can be deduced).

This generator works with two steps: at first, chain of non-terminals gets and secondly, the chain translated into processor instructions. For example:

$$q_0 \to BB \to BDD \to BDKK \to BDKLL \to \cdots \to$$
$$\to BDKLMN \to b_1 d_3 k_2 l_7 m_9 n_{13}$$

To summarize, we get context-free grammar, because we build it over automata. Main problem of considered generator is growing up of mutated code. Let's consider one approach to code compression. Assume X, Y, Z - x86 commands and $XY \equiv Z$. Than rule $XY \to Z$ means that instruction sequence XY compresses into Z. For example,

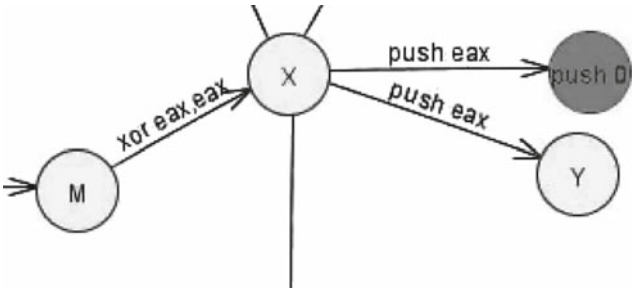

Fig. 3 Code compression part of automaton chart

$X \equiv xor\, eax, eax$, $Y \equiv push\, eax$, $z = push\, 0$ and $P = \begin{cases} M \to XX \\ X \to YY \\ Y \to NN \\ XY \to z \end{cases}$

In this case generator output is of the following form:

$$M \to XX \to XYY \to \begin{cases} zY \to zNN \to \cdots \\ XYNN \to \cdots \end{cases}$$

Mark, that both branches are semantically equivalent—moving a zero at stack top.

How to interpret it by term of automaton? Consider a certain automaton with current state M. Symbol $xor\, eax, eax$ is input symbol. After that, automaton branchs move to X state, which matches to some translation of $xor\, eax, eax$ instruction. The next input symbol is $push\, eax$. Automaton could branchs to Y state or z state, which matches to $push\, 0$ instruction. State z is special state: when automaton gets to z, automaton must discards previous state X from its path. This idea matches to imaginary edge from M to z. Figure 3 illustrates it.

Thus, after adding rules of a new type, our grammar gets type 0 of Chomsky classification. For grammars of this type, language decision problem is undecided. That is if we have an instance of viral code we couldn't determinate predecessor of this instance. Given fact confirms a possibility of making undetected viruses.

4 Method limitations

At practice some limitation of discussed model exists. At first, all used instructions (or kinds of instructions) must predicted in grammar rules. Therefore, metamorphic generator will be huge. Secondary, rules of described grammar presumes random garbage generation. So, after translation input instruction engine saves "context" (register values) and generates various garbage instructions so as "contexts" at the beginning and ending of garbage code block are equal. Unfortunately, these dummy code blocks can be found by static analysis, as well as, algorithm of string search works. And

thirdly, these code mutation techniques are effective only for "clean" code, without any hard points. I am speaking about system calls.

5 Practical grammar usage

This chapter describes practical building of polymorphic generator and its detector; both based on formal grammars and automatons. Lets build simple polymorphic decryptor such as following:

1. mov R_1, len
2. mov R_2, beg
3. xor [R_2], key
4. add R_2, 4
5. sub R_1, 4
6. jnz step_3

Now we are going to describe rewriting system of a grammar. Let rules X_1 and X_2 are corresponds to two first instructions. Order of these instructions is unimportant. For XOR instruction let's use following well-known equivalents: $x_1\ xor\ x_2 \equiv \neg(\neg x_1\ xor\ x_2) \equiv (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$. Additionally let that generator can uses two forms of XOR instruction: base addressing mode W_1 and base-indexed addressing mode W_2. H_1 and H_2 rules describes cycle organization and G is garbage generation rule. So rewriting system looks as:

$$
\begin{aligned}
A &\to XB\\
B &\to Y_4\varepsilon\\
X &\to X_1X_2|X_2X_1\\
X_1 &\to GX_1|mov\ R_1, len|push\ len \oplus pop\ R_1|xor\ R_1,\\
&\quad R_1 \oplus lea\ R_1, [R_1 + len]|sub\ R_1, R_1 \oplus add\ R_1, len\\
X_2 &\to GX_2|mov\ R_2, beg|push\ beg \oplus pop\ R_2|xor\ R_2,\\
&\quad R_2 \oplus lea\ R_2, [R_2 + beg]|sub\ R_2, R_2 \oplus add\ R_2, beg\\
Y_4 &\to GY_4|W_1|S_4W_4\\
W_1 &\to GW_1|xor\ [R_2], key\ H_1\\
W_1 &\to not\ [R_2] \oplus xor\ [R_2], key \oplus not[R_2]\ H_1\\
W_1 &\to mov\ R_3, [R_2] \oplus not\ R_3 \oplus and\ R_3, key \oplus and\ [R_2],\\
&\quad \neg key \oplus or\ [R_2], R_3\ H_1\\
H_1 &\to GH_1|add\ R_2, 4\ H_2|sub\ R_2, -4\ H_2\\
S_4 &\to GS_1|sub\ R_2, 4|add\ R_2, -4\\
W_2 &\to GW_2|xor\ [R_1][R_2], key\ H_2\\
W_2 &\to not\ [R_1][R_2] \oplus xor\ [R_1][R_2], key \oplus not[R_1][R_2]\ H_2\\
W_2 &\to mov\ R_3, [R_1][R_2] \oplus not\ R_3 \oplus and\ R_3, key \oplus and\\
&\quad [R_1][R_2], \neg key \oplus or\ [R_1][R_2], R_3\ H_2\\
H_2 &\to GH_2|sub\ R_1, 4 \oplus jnz\ xxx|sub\ R_1, 4 \oplus jz\ yyy \oplus jmp\ xxx\\
H_2 &\to add\ R_1, -4 \oplus jnz\ xxx|add\ R_1, -4 \oplus jz\ yyy \oplus jmp\ xxx\\
H_2 &\to sub\ ecx, 3 \oplus loop\ xxx \Leftrightarrow R_1 \equiv ecx
\end{aligned}
$$

```
00 PUSH 44554433
01 POP ESI
02 SUB EBX,EBX
03 ADD EBX, 124
04 XOR [ESI],d20b9a65
05 ADD ESI,4
06 SUB EBX,4
07 JZ $ + 2
08 JMP $ + f0
```

```
00 XOR EDI,EDI
01 LEA EDI,[EDI+124]
02 PUSH 44554433
03 POP ESI
04 MOV EDX,[ESI]
05 NOT EDX
06 AND EDX,d75d40bc
07 AND [ESI],28a2bf43
08 OR [ESI],EDX
09 ADD ESI,4
10 SUB EDI,4
11 JZ $ + 2
12 JMP $ + e4
```

Fig. 4 Samples of generated decryptors

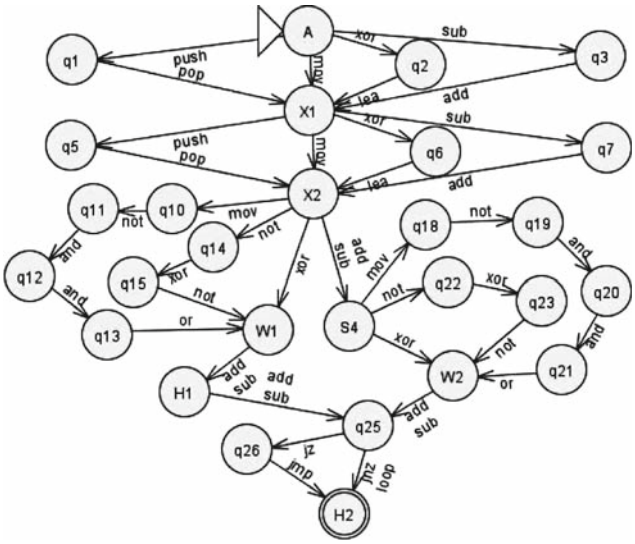

Fig. 5 Automaton-detector

Let realize this generator with empty garbage rule. So, this engine can generate $4^2 \cdot (3 \cdot 2 + 2 \cdot 3) \cdot 5 = 960$ different decryptors without regard any register replacement. Disassembled generator outputs seems as following (Fig. 4):

Automaton-detector that can detect any output of our generator, presented in Fig. 5. This automaton analyzes input instruction and moves into next state. Automaton recognizes code sequence as decryptor when it reaches finish state.

The automaton contains 28 states and can recognizes any word-build by our simple context-free grammar. So, than more complex originative grammar than more and more complex automaton-detector. This is one way to design "undetectable" code sequences. Another way is garbage generation. When garbage rules are the same with payload rules than automaton can loses main execution stream. Let's add following rules to generator:

$$
\begin{aligned}
G &\to GRBG_1|GRBG_2|GRBG_3|GRBG_4\\
G &\to mov\ R, imm|push\ imm \oplus pop\ R|xor\ R,\\
&\quad R \oplus lea\ R, [R + imm]|sub\ R, R \oplus add\ R, imm
\end{aligned}
$$

```
00 GRBG                        [0] = GRBG  state: 0 => 0
01 MOV EDX,124                 [1] = MOV   state: 0 => 4
02 GRBG                        [2] = GRBG  state: 4 => 4
03 GRBG                        [3] = GRBG  state: 4 => 4
04 XOR ESI,ESI                 [4] = XOR   state: 4 => 6
05 LEA ESI,[ESI+44554433]      [5] = LEA   state: 6 => 8
06 MOV EBX,b69bcb88            [6] = MOV   state: 8 => 10
07 XOR EBX,EBX                 [7] = XOR   state: 10 => 10
08 LEA EBX,[EBX+a40b14ee]      [8] = LEA   state: 10 => 10
09 GRBG                        [9] = GRBG  state: 10 => 10
10 GRBG                        [10] = GRBG state: 10 => 10
11 GRBG                        [11] = GRBG state: 10 => 10
12 GRBG                        [12] = GRBG state: 10 => 10
13 XOR ECX,ECX                 [13] = XOR  state: 10 => 10
14 LEA ECX,[ECX+ddb3fe99]      [14] = LEA  state: 10 => 10
15 GRBG                        [15] = GRBG state: 10 => 10
16 GRBG                        [16] = GRBG state: 10 => 10
17 GRBG                        [17] = GRBG state: 10 => 10
18 PUSH 154f25e1               [18] = PUSH state: 10 => 10
19 POP EBX                     [19] = POP  state: 10 => 10
20 XOR [ESI],b79275ab          [20] = XOR  state: 10 => 10
21 GRBG                        [21] = GRBG state: 10 => 10
22 ADD ESI,4                   [22] = ADD  state: 10 => 10
23 SUB EDX,4                   [23] = SUB  state: 10 => 10
24 JNZ $ + f1                  [24] = JNZ  state: 10 => 10
                               NOT DETECTED
                               ...
                               Total code sequences:   100
                               Detected sequences:     68
                               Non-detected sequences: 32
```

Fig. 6 Sample of generated decryptor and detector output

$GRBG_i$ is simple garbage instructions such as *mov Reg*, *Reg* and etc. Now try to detect generator output (Fig. 6).

As we can see insignificant grammar change entails are decreasing of detecting probability from 1 to $\frac{2}{3}$. But in spite of this any polymorphic virus under like this decryptor can be simply detected by system calls analysis.

6 Experiment: system calls tracing

This chapter describes experiment results of watching famous virus behavior on computer with Windows XP SP2 with the use of special program which intercepts system calls of specified process and writes log. This is dynamic research method. Propagation copies of one strain and different strains of some viruses are considered. The next viruses and worms had been analyzed: Bagle.a-z, Mimail.a-u, Tanatos.a-n and Zmist. Really existing machine has been cloned and user activity has been emulated. Additional information about some of these viruses can be found in [4,5].

System calls log by default is very verbose. Appendix A shows part of a log for process prologue. In the log can be found process name, process ID, thread ID, system call name, two return addresses, parameter names list and values lists.

For research let's use more compact view of system calls log with syscall name and return addresses. These addresses needed for determinate start of program without any OS services actions (process creation and etc). To this effect used the next fact: any process under Windows XP begins from startup code from kernel32.dll. Operation system calls *NtSetInformationThread* before transfer control to executable file entry point. Figures 7 and 8 illustrates it.

For comparison of system calls sequences let's introduce measure of distinction of these sequences. Function $\mu(x, y)$ is number of different blocks of system calls in the log for viruses x and y. Measure $\mu(x, y) = 0$ means that viruses x and y uses identical system calls. Of course, $\mu(x, x) = 0$. Computation of $\mu(x, y)$ can be based on any algorithm of text file comparison such as realized in xdiff utility. In this

```
.text:77E814B4  push    4
.text:77E814B6  lea     eax, [ebp+8]
.text:77E814B9  push    eax
.text:77E814BA  push    9
.text:77E814BC  push    0FFFFFFFEh
.text:77E814BE  call    ds:NtSetInformationThread
.text:77E814C4  call    dword ptr [ebp+8]
```

Fig. 7 Main thread startup code in kernel32.dll

```
21:47:11 bagle.a.exe(976.320)
    NtSetInformationThread[229](16) 77F5C294<=77E814C4
    0012FFB8:  ThreadHandle FFFFFFFE
    0012FFBC:  ThreadInformationClass   9
    0012FFC0:  ThreadInformation    0012FFF8
    0012FFC4:  ThreadInformationLength  00000004
```

Fig. 8 Appropriate system call for main thread startup in the log

instance algorithm of searching maximum subsequence from first log in second log was implemented. Thus, block is a system calls sequence which presence in first log but absence in second log or vice versa. So measure $\mu(x, y)$ that shows number of different blocks is express method of distinction evaluation of system calls sequences.

Let's begin research with Bagle. At first, we simply start different strains of Bagle and save its activity into logs. Namely we trace a first worm penetration on target system. Table in Appendix B presents $\mu(x, y)$ values for Bagle.X. As can seen Bagle.h is functional equivalent to Bagle.k, Bagle.l is functional equivalent to Bagle.v, Bagle.n—Bagle.r are nearly equivalent and Bagle.t is based on Bagle.a. Also functional signature for detecting Bagle can be build. Classic approach presumes one signature per one strain of virus. In functional signature case we can use one signature for few strains. Experimental data shows that different strains of Bagle.X contains a lot of similar system calls subsequences when $\mu(x, y) < 40$. So we can build up 1 functional signature for Bagle.a, Bagle.b and Bagle.t strains, 1 signature for f, g, h, k, l, m, v strains, 1 signature for i, j strains and 1 signature for n, o, p, q, r, y strains. Thus we have 4 functional signature instead of 18 classic signatures. Bagle.X permanent activity also has been traced and results can be found at Appendix C.

The next step is analogous system calls tracing for different strains of Mimail worm. Measure values can be found in Appendix D. This results confirms an efficiency of functional signatures: 4 signatures (1 for Mimail.a-p except Mimail.i, 1 for Mimail.i, 1 for Mimail.r and 1 for Mimail.q-u except Mimail.r) instead of 21 classic signatures. Also polymorphic copies of Mimail.q has been researched. So all 50 gotten copies has μ-values zero or two that matches to short blocks replacing.

Appendix E presents measure values for different strains of Tanatos worm. As can be seen all Tanatos strains has invariable system call sequences. All examined 50 polymorphic copies of Tanatos.b has $\mu = 0$.

Appendix F shows results of system calls tracing for Zmist virus. Eight letters a-h matchs to eight different virus samples. Notepad.exe has been infected and the same application activity has been performed. So, these measure values tells that chosen measure is abortive or the simplest for code integration technique used by Zmist.

Thus system calls sequences can be used for detecting poly- and metamorphic copies of one strain of viruses. In this case usage of code mutation techniques there is no point because of detecting occurs on operation system level. System calls sequences logically uses to detecting all strains of one virus family because unionization virus exemplars into one virus family occurs by functional similarity.

However, the problem is separation virus system calls from general sequence of program system calls. Unique arguments of system calls can be used but in this case functional signature will be huge and this method is not always applicable. Usage of single calls in functional signature is not usable because false-positive operates is possible.

7 Conclusion and future work

The main method of detecting metamorphic viruses is behavior analysis. Sorry to say, this method have a restriction – a necessity of running dubious code. Alternative approach is considered in [6–9]. These methods based on possibility of disassemble viral code. Authors also note that some obfuscation tricks may be barriers for using of this methods. Semi-metamorphic viruses, which considered above, don't require self disassemble possibility, because skeleton is contained in the metamorphic generator.

Considered formalization very well describes code mutation. But this is partially unfull because it descries only code transformation without code executing environment. We can perform any code transform, but system-depended points are exist. There are sequences of system calls. This sequence can be used for successful detection of formal undetected metamorphic viruses. Some methods for bridge over this restrictions are exists. For example, it may be garbage system calls or function mutation technique.

A general application of metamorphism technique is created of undetectable viruses. But peaceable adaptation for metamorphism exists. Firstly, it may be a software watermark or fingerprint at processor instruction level for tracing program owners. Secondly, metamorphism allows creating fully different copy of a program when it expands by Internet. In this case cracker could not create patch because each user have unique copy of a program.

Appendix A: System calls log sample

```
21:47:10 bagle.a.exe(976.320)
    NtOpenKey[119](12) 77F5BBB4<=77F61BD3
    0012F950:  KeyHandle 0012FC94 -> 00000000
    0012F954:  DesiredAccess    80000000
    0012F958:  ObjectAttributes 0012FC24 ->
        0012FC24:  OBJECT_ATTRIBUTES
        0012FC24:  Length   00000018
        0012FC28:  RootDirectory    00000000
        0012FC2C:  ObjectName   0012FC3C ->
            0012FC3C:  UNICODE_STRING
            0012FC3C:  Length   00CE
            0012FC3E:  MaximumLength    02C6
            0012FC40:  Buffer   0012F95C ->
    \Registry\Machine\Software\Microsoft\Windows NT\
    CurrentVersion\Image File Execution Options\bagle.a.exe
        0012FC30:  Attributes   00000040
        0012FC34:  SecurityDescriptor   00000000
        0012FC38:  SecurityQualityOfService 00000000

21:47:10 bagle.a.exe(976.320)
    NtQuerySystemInformation[173](16) 77F5BF14<=77F559C2
    0012FA4C:  SystemInformationClass   8
    0012FA50:  SystemInformation 0012FA9C
    0012FA54:  SystemInformationLength  0000002C
    0012FA58:  ReturnLength 00000000 ->

21:47:10 bagle.a.exe(976.320)
    NtAllocateVirtualMemory[17](24) 77F5B554<=77F55BD5
    0012FA44:  ProcessHandle    FFFFFFFF
    0012FA48:  BaseAddress  0012FB00
    0012FA4C:  ZeroBits 00000000
    0012FA50:  AllocationSize   0012FB2C -> 00100000
    0012FA54:  AllocationType   00002000
    0012FA58:  Protect   00000004
```

Appendix B: Bagle.X functional correlation (for first execution on target computer)

X	a	b	f	g	h	i	j	k	l	m	n	o	p	q	r	t	v	y
a	0	26	121	121	119	126	126	121	120	123	112	116	116	111	115	3	120	113
b	26	0	135	134	134	128	129	134	136	128	135	125	126	136	126	24	136	130
f	121	135	0	1	4	51	52	4	22	28	76	70	70	76	71	128	21	64
g	121	134	1	0	5	52	53	5	22	29	77	71	71	77	70	128	22	63
h	119	134	4	5	0	52	54	0	20	25	73	78	68	73	69	127	20	63
i	126	128	51	52	52	0	2	53	53	61	47	45	46	48	47	128	53	44
j	126	129	52	53	54	2	0	57	54	61	35	33	32	36	35	128	54	32
k	121	134	4	5	0	53	57	0	20	25	73	68	68	73	69	127	20	63
l	120	136	22	22	20	53	54	20	0	15	70	63	65	71	66	126	0	66
m	123	128	28	29	25	61	61	25	15	0	89	85	85	87	86	128	15	84
n	112	125	76	77	73	47	35	73	70	89	0	4	6	2	7	116	64	20
o	116	125	70	71	78	45	33	68	63	85	4	0	2	6	3	117	59	19
p	116	126	70	71	68	46	32	68	65	85	6	2	0	4	1	119	61	20
q	111	136	76	77	73	48	36	73	71	87	2	6	4	0	5	118	63	21
r	115	126	71	70	69	47	35	69	66	86	7	3	1	5	0	119	62	19
t	3	24	128	128	127	128	128	127	126	128	116	117	119	118	119	0	127	115
v	120	136	21	22	20	53	54	20	0	15	64	59	61	63	62	127	0	66
y	113	130	64	63	63	44	32	63	66	84	20	19	20	21	19	115	66	0

Appendix C: Bagle.X functional correlation (part of permanent activity)

X	a	b	f	g	h	i	j	k	l	m	n	o	p	q	r	t	v	y
a	0	5	138	130	139	132	136	139	139	132	135	137	137	135	137	3	139	138
b	5	0	131	133	133	133	130	134	132	125	130	132	132	130	133	16	132	134
f	138	131	0	5	11	48	48	14	10	17	90	93	91	90	93	125	10	109
g	130	133	5	0	14	48	48	11	8	17	93	94	94	93	94	125	8	112
h	139	133	11	14	0	47	47	5	13	20	87	90	89	87	90	126	13	107
i	132	133	48	48	47	0	6	49	49	51	30	35	36	31	36	130	49	38
j	136	130	48	48	47	6	0	49	49	51	18	23	24	19	24	112	49	26
k	139	134	14	11	5	49	49	0	13	22	92	93	91	92	93	127	12	113
l	139	132	10	8	13	49	49	13	0	15	53	54	53	53	54	125	0	56
m	132	125	17	17	20	51	51	22	15	0	70	76	63	70	78	132	15	76
n	135	130	90	93	87	30	18	92	53	70	0	9	11	2	11	119	47	22
o	137	132	93	94	90	35	23	93	64	76	9	0	8	11	2	120	49	22
p	137	132	91	94	89	36	24	91	53	63	11	8	0	9	6	121	48	21
q	135	130	90	93	87	31	19	92	53	70	2	11	9	0	9	120	48	23
r	137	133	93	94	90	36	24	93	54	78	11	2	6	9	0	121	49	23
t	3	16	125	125	126	130	112	127	125	132	119	120	121	120	121	0	128	126
v	139	132	10	8	13	49	49	10	0	15	47	49	48	48	49	128	0	55
y	138	134	109	112	107	38	26	113	56	76	22	22	21	23	23	126	55	0

Appendix D: Mimail.X functional correlation

X	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
a	0	6	28	6	36	26	22	34	107	17	23	33	17	7	7	46	92	130	88	63	101
b	6	0	16	1	17	23	22	24	113	17	26	34	21	3	5	37	93	140	84	69	98
c	28	16	0	18	135	193	84	155	250	19	128	245	153	18	14	337	198	206	228	81	210
d	6	1	18	0	24	22	20	24	118	18	24	32	21	3	5	35	93	140	84	69	101
e	36	17	135	24	0	179	38	81	321	11	72	105	64	26	38	305	182	275	188	86	189
f	26	23	193	22	179	0	123	164	264	12	157	162	135	25	17	259	167	240	185	104	187
g	22	22	84	20	38	123	0	29	199	11	35	107	53	20	24	282	143	201	148	70	156
h	34	24	155	24	81	164	29	0	276	11	77	166	57	27	25	390	186	229	203	84	189
i	107	113	250	118	321	264	119	276	0	103	203	253	198	114	116	611	160	203	233	46	179
j	17	17	19	18	11	12	11	11	103	0	8	16	10	18	17	16	91	136	74	73	99
k	23	26	128	24	75	157	35	77	203	8	0	132	142	22	20	441	186	282	205	94	185
l	33	34	245	32	105	162	107	166	253	16	132	0	113	36	27	644	223	256	231	96	207
m	17	21	153	21	64	135	53	57	198	10	142	113	0	23	20	323	159	214	90	84	129
n	7	3	18	3	26	25	20	27	114	18	22	36	23	0	6	37	93	141	84	69	101
o	7	5	14	5	38	17	24	25	116	17	20	27	20	6	0	50	92	140	84	71	98
p	46	37	337	35	305	259	282	390	611	16	441	644	323	37	50	0	325	442	416	144	404
q	92	93	198	93	182	167	143	186	160	91	186	223	159	93	92	325	0	167	20	20	26
r	130	140	206	140	275	240	201	229	203	136	282	256	214	141	140	442	167	0	172	142	156
s	88	84	228	84	188	185	148	203	233	74	205	231	90	84	84	416	20	172	0	16	19
t	63	69	81	69	86	104	70	84	46	73	94	96	84	69	71	144	20	142	16	0	40
u	101	98	210	101	189	187	156	189	179	99	185	207	129	101	98	404	26	156	19	40	0

Appendix E: Tanatos.X functional correlation

X	a	b	d	e	g	i	k	l	n
a	0	11	11	16	8	8	19	17	17
b	11	0	6	8	6	6	4	5	15
d	11	6	0	3	10	9	6	7	17
e	16	8	3	0	5	6	8	9	78
g	8	6	10	5	0	1	18	18	25
i	8	6	9	6	1	0	18	18	25
k	19	4	6	8	18	18	0	1	20
l	17	5	7	9	18	18	1	0	20
n	17	15	17	78	25	25	20	20	0

Appendix F: Zmist functional correlation

X	a	b	c	d	e	f	g	h
a	0	75	117	27	53	79	293	171
b	75	0	125	134	123	56	128	225
c	117	125	0	71	40	75	129	41
d	27	134	71	0	50	113	52	93
e	53	123	40	50	0	75	30	53
f	79	56	75	113	75	0	66	86
g	293	128	129	52	30	66	0	37
h	171	225	41	93	53	86	37	0

References

1. Qozah. Polymorphism and grammars, 29A E-zine, 1999, #4
2. Filiol, E.: Metamorphism, formal grammars and undecidable code mutation. In: Proceedings of World Academy of Science, Engineering and Technology (PWASET), vol. 20 (2007)
3. Jones, N.D.: Computability and Complexity. MIT Press, Cambridge (1997)
4. Filiol, E.: Computer viruses: from theory to applications, 405 p. Springer, France (2005)
5. Szor, P.: The Art of Computer: Virus Research and Defense, 744 p. Symantec Press, USA (2005)
6. Bruschi, D., Martignoni, L., Monga, M.: Using Code Normalization for Fighting Self-Mutating Malware, Security & Privacy, IEEE, vol. 5, pp. 46–54 (2007)
7. Lakhotia, A., Kapoor, A., Uday E.: Are metamorphic viruses really invincible? Virus Bulletin, pp. 5–7 (2004)
8. Lakhotia, A., Kapoor, A., Uday E.: Are metamorphic viruses really invincible? Virus Bulletin, pp. 9–12 (2005)
9. Zhang, Q., Reeves, D.: MetaAware: identifying metamorphic malware. In: Proceedings of the 23rd Annual Computer Security Applications Conference, Miami Beach, Florida (2007)