J Comput Virol (2006) 2:149–161
DOI 10.1007/s11416-006-0023-z

INVITED PAPER

Detection of metamorphic computer viruses using algebraic specification

Matt Webster · Grant Malcolm

Published online: 31 August 2006

Abstract This paper describes a new approach towards the detection of metamorphic computer viruses through the algebraic specification of an assembly language. Metamorphic computer viruses are computer viruses that apply a variety of syntax-mutating, behaviour-preserving metamorphoses to their code in order to defend themselves against static analysis based detection methods. An overview of these metamorphoses is given. Then, in order to identify behaviourally equivalent instruction sequences, the syntax and semantics of a subset of the IA-32 assembly language instruction set is specified formally using OBJ – an algebraic specification formalism and theorem prover based on order-sorted equational logic. The concepts of equivalence and semi-equivalence are given formally, and a means of proving equivalence from semi-equivalence is given. The OBJ specification is shown to be useful for proving the equivalence or semi-equivalence of IA-32 instruction sequences by applying reductions – sequences of equational rewrites in OBJ. These proof methods are then applied to fragments of two different metamorphic computer viruses, Win95/Bistro and Win9*x*.Zmorph.A, in order to prove their (semi-)equivalence. Finally, the application of these methods to the detection of metamorphic computer viruses in general is discussed.

M. Webster (✉) · G. Malcolm
Department of Computer Science,
University of Liverpool,
Liverpool, L69 3BX, UK
e-mail: matt@csc.liv.ac.uk

G. Malcolm
e-mail: grant@csc.liv.ac.uk

1 Introduction

Computer viruses are typically segments of a stored program that when run are able to create a copy of themselves in another stored program. During this process of reproduction, it is possible for the virus to modify itself in some way. Metamorphic computer viruses essentially replace sequences of instructions with syntactically different (yet semantically equivalent) sequences of instructions in successive generations [10]. In this way the behaviour of each generation is the same, but the actual code is different. Typically, this is done in order to avoid static analysis based detection methods such as signature scanning, heuristic analysis and spectral analysis.

This paper describes an approach towards to the detection of metamorphic computer virus using an algebraic specification of the IA-32 assembly language. In Sect. 2 an overview of the specification is given, and the notions of equivalence and semi-equivalence of instructions and instruction sequences are defined formally. Using this formalism we prove that semi-equivalence can be extended to equivalence under certain conditions that can be checked using static analysis. The OBJ specification, when combined with the OBJ term rewriting engine, can be used as an interpreter for programs in IA-32, and this in turn can be used for dynamic analysis of computer viruses. In Sect. 3 this dynamic analysis is used to prove the equivalence and semi-equivalence of real-life metamorphic computer virus code fragments, and potential applications to metamorphic computer virus detection are discussed. In Sect. 4 some directions for future research are given.

1.1 Types of code metamorphosis

Metamorphic computer viruses conceal their code from anti-virus scanners using a variety of semantics-preserving, syntax-mutating methods [7]. Here a non-exhaustive list of the different kinds of code metamorphosis is given in order to demonstrate the many and varied ways in which metamorphic computer viruses can use syntactic camouflage to defend themselves against static analysis based detection. Several of these types were given by Lakhotia and Mohammed [7].

1.1.1 Junk code insertion

Junk code is code that is superfluous to the main function(s) of the virus, and is inserted to create syntactic variants. There are different types of junk code, including but not limited to:

- Code that reverses the effects of a previous instruction or instructions, thus making the previous instruction(s) and the inverse code into junk. For example, the instruction sequence `xchg eax,ebx ; xchg eax,ebx`—which swaps the values in registers `eax` and `ebx` twice—would fall under this category. (Note that, throughout this paper, we use a semicolon (`;`) to indicate sequential composition of assembly language instructions.)
- Code that performs a computation that is not utilised in any of the outputs of the program. For example, the first instruction in the following instruction list does nothing as the result is overwritten by the next instruction: `mov eax,0 ; mov eax,ebx`.

1.1.2 Variable renaming

Variables are renamed in successive generations of metamorphic computer viruses such as Win9*x*.Regswap [10]. For instance, `mov eax,0 ; push eax ; pop ebx` could be replaced by the equivalent instruction sequence `mov ecx,0 ; push ecx ; pop ebx`.

```
push ebp                          push ebp
mov ebp, esp                      push esp
                                  pop ebp

mov esi, dwordptr [ebp + 08]      mov esi, dwordptr [ebp + 08]
test esi, esi                     or esi, esi
je 401045                         je 401045

mov edi, dwordptr [ebp + 0c]      mov edi, dwordptr [ebp + 0c]
or edi, edi                       test edi, edi
je 401045                         je 401045
```

Fig. 1 Allomorphic fragments of Win95/Bistro [10]

1.1.3 Unconditional jump insertion

A block of instructions is broken up into more than one smaller blocks of instructions linked by unconditional jumps. For example:

```
pop edx                           pop edx
mov edi,0004h                     jmp label1
mov esi,ebp                       label2:
mov eax,000Ch                     jmp label3
                                  label1:
                                  mov edi,0004h
                                  mov esi,ebp
                                  jmp label2
                                  label3:
                                  mov eax,000Ch
```

1.1.4 Instruction reordering

Blocks of data-independent instructions are reordered to create syntactic variants. For example, `mov eax, ebx ; mov esi,edi` can be reordered to `mov esi, edi; mov eax,ebx`.

1.1.5 Equivalent sequence replacement

A sequence of instructions is replaced by equivalent sequences of instructions in order to generate syntactic variants. A good example of this can be seen in the Win95/Bistro virus (see Fig. 1, Sect. 3.1).

1.1.6 Pseudo-conditional jump insertion

A sequence of instructions ends in a conditional jump that depends entirely on information encoded in the preceding instructions. An example of this would be the following instruction sequence, `mov eax,20 ; sub eax,20 ; je label1`, in which the conditional jump `je` ("jump if the zero flag is set to 1") is effectively unconditional because the preceding instructions always set the zero flag to 1.

1.1.7 Arithmetical/Boolean mutation

Arithmetical and Boolean operations can be easily mutated into other, equivalent forms. A good example of this can be found in the Win9*x*.Zmorph.A virus (see Fig. 2, Sect. 3.2).