Detection of Metamorphic and Virtualization-based Malware using Algebraic Specification

Matt Webster and Grant Malcolm*

Abstract

We present an overview of the latest developments in the detection of metamorphic and virtualization-based malware using an algebraic specification of the Intel 64 assembly programming language. After giving an overview of related work, we describe the development of a specification of a subset of the Intel 64 instruction set in Maude, an advanced formal algebraic specification tool. We develop the technique of metamorphic malware detection based on equivalence-in-context so that it is applicable to imperative programming languages in general, and we give two detailed examples of how this might be used in a practical setting to detect metamorphic malware. We discuss the application of these techniques within anti-virus software, and give a proof-of-concept system for defeating detection counter-measures used by virtualization-based malware, which is based on our Maude specification of Intel 64. Finally, we compare formal and informal approaches to malware detection, and give some directions for future research.

1 Introduction

In this paper we present the latest developments on the detection of metamorphic and virtualization-based malware using an algebraic specification of a subset of the Intel 64 assembly language instruction set. Both metamorphic and virtualization-based malware present serious challenges for detection: undetectable metamorphic computer viruses are known to exist [4, 9], and virtualization-based malware seem able to create a virtual computational platform which is indistinguishable to the user under normal circumstances, but which is completely under the control of the malware [20, 13]. We will now give an overview of the existing detection methods for metamorphic and virtualization-based malware.

There are currently many avenues of research into the detection of metamorphic computer viruses, both academic and industrial. Lakhotia & Mohammed describe an algorithm for imposing order on high-level language programs based on control- and data-flow analysis [17, 14]. Bruschi et al [1] describe a similar method for malware detection to the one described by Lakhotia & Mohammed, which uses code normalisation. Christodorescu et al describe a formal approach to metamorphic computer

Postprint. To appear in the proceedings of the 17th Annual EICAR Conference (EICAR 2008), Laval, France.

*Department of Computer Science, University of Liverpool, Liverpool, L69 3BX, UK. Email: {matt,grant}@csc.liv.ac.uk.

virus detection using a signature-matching approach, in which the signatures contain information regarding the semantics, as well as the syntax, of the metamorphic computer virus [6]. In a later paper Preda et al [18] are able to prove the correctness of this approach with respect to instruction reordering, variable renaming and junk code insertion. Bruschi et al describe a normalisation procedure based on program rewriting [2, 3]. Chouchane & Lakhotia describe an approach to metamorphic computer virus detection based on the assumption that metamorphic computer often use the same metamorphism engine, and that by assigning an *engine signature* it ought to be possible to assign a probability that a suspect executable is an output of that engine [5]. Yoo & Ultes-Nitsche [26, 27] present a unique approach to metamorphic computer virus detection, which involves training a type of artificial neural network known as a self-ordering map (SOM). Recent work by Ször [22, 23] describes some of the industrial techniques for the detection of metamorphic computer virus detection.

As virtualization-based malware is a relatively recent phenomenon [20, 13], there is less in the literature on the problem of its detection. King et al give a detailed overview of the state of the art in virtual machine-based rootkits (VMBRs) through the demonstration of proof-of-concept systems, and explore strategies for defending against VMBRs [13]. Garkinkel et al [10] describe a taxonomy of virtual machine detection methods, and describe a fundamental trade-off between performance and transparency when designing virtual machine monitors. Rutkowska describes a technique for detecting VMBRs called Red Pill, in which the Intel 64 instruction SIDT is used to reveal the presence of a virtual machine monitor through an altered interrupt descriptor table [19].

Algebraic specification has been applied to the problem of metamorphic malware detection previously [24]. Using a formal specification in OBJ of a subset of the Intel 64 assembly language instruction set, it was shown that it was possible to prove the equivalence and semi-equivalence of programs using a reduction — a sequence of equational rewrites. When combined with the OBJ term rewriting engine, the algebraic specification becomes an interpreter for the programming language, and can be used to prove the equivalence of assembly language programs. Notions of equivalence and semi-equivalence were defined formally, and it was shown that it is possible to extend semi-equivalence to equivalence under certain conditions, known as “equivalence-in-context”. This paper builds upon this approach.

In Section 2 we describe a translation of the Intel 64 specification from OBJ to Maude, a successor to OBJ which allows proofs based on rewriting logic. In the earlier work, the technique of proving equivalence-in-context was only applicable to certain assembly language instructions for which we could prove (using a reduction in OBJ) that keeping one set of variables constant would ensure that another set of variables would have the same values after executing the instruction within two different states [24]. In Section 3 we improve this result by extending showing that equivalence-in-context is applicable to all instructions in imperative programming languages, regardless of whether we can prove the above condition using a reduction in OBJ or Maude. We then give concrete examples of how equivalence-in-context can be used in practice to detect metamorphic malware. In Section 4 we discuss the applicability of the algebraic approaches given in Section 3 and [24] to the practical problem of detection of metamorphic malware based on formal static and dynamic analysis, and in Section 5 we give a proof-of-concept system for generating metamorphic variants of virtualization-detection programs (such as Red Pill [19]), based on the additional proof tools available in Maude.

2 Specifying Intel 64 Assembly Language

In this section we summarise Webster and Malcolm’s approach [24] to specifying the syntax and semantics of the Intel 64 assembly language, and describe how algorithmic techniques can use this specification to reason about programs written in the language. The Intel 64 and Intel Architecture 32-bit (IA-32) instruction set architectures [12] are used by the vast majority of personal computers worldwide, and it follows that the majority of computer viruses will (at some point in their reproductive cycle) be manifest as a sequence of Intel 64 instructions. The full Maude specification, which is described below, can be found online [25].

2.1 Specifying the Syntax of Intel 64

The Intel 64 assembly language itself can be specified in Maude (see [7] for details of the Maude language; the present discussion does not, however, require any knowledge of Maude) by declaring sorts for instructions, expressions, variables, etc., and declaring the constructs of the language as operations. For example, the `mov` instruction is used in Intel 64 to assign the value of an expression (either a program variable name or a value) to another program variable, i.e., it "`mov`es" the value of the expression in its right-hand (*source*) operand to the program variable in its left-hand (*destination*) operand. We can specify the syntax of the `mov` instruction as follows:

```
mov_,_  :  Variable Expression -> Instruction .
```

The variables of the language are the registers `eax`, `ebx`, `ecx`, and `edi`, together with various "flags", such as the instruction pointer `ip`, and the stack, which can also be declared as a constant `stack`.

An important feature of the language is that instructions can be composed and put together to form programs. It is convenient to declare this composition operation using a semi-colon notation rather than the standard juxtaposition. In Maude this notation is declared as an operation

```
_;_  :  Instruction Instruction -> Instruction
```

(throughout this paper we shall blur the distinction between sequences of instructions and individual instructions).

The significance of specifying the syntax of the language in Maude is that programs can then be represented as terms such as

```
mov ecx, eax ; mov eax, ebx ; mov ebx, ecx .
```

This can then be used as a basis for a formal specification of the semantics of the language.

2.2 Specifying the Semantics of Intel 64

Following the approach of Goguen and Malcolm [11], the semantics of a programming language can be specified by describing the effect of programs upon the *state* of the machine that executes those programs. This state is effectively captured by the values stored in the variables of the language: programs update this state by manipulating these values. Webster and Malcolm's specification declares a sort `Store` to represent these states, together with operations that capture how stores and programs interact.

For example, evaluation of an expression in a given state is done by declaring an operation

```
_[[_]]  : Store Expression -> EInt
```

(where `EInt` represents integers together with "error values" that might arise through, for example, stack overflows). Expressions may include variables, and for a store `S` and variable `V`, the term `S[[V]]` is intended to denote the value stored in `V` in the state `S`.

The action of a program upon a state is captured by an operation

```
    _;_   :  Store Instruction -> Store
```

so that for a store `S` and instruction `P`, the term `S ; P` denotes the store that results from executing `P` in the "starting state" `S`. Putting all the above together, a term such as

```
    s ; mov ecx, eax ; mov eax, ebx ; mov ebx, ecx [[ ebx ]]
```

is intended to denote the value in the `ebx` register after the program has executed. Equations are used in the Maude specification to stipulate exactly what such values should be. For example, the three equations

```
   S ; mov V,E [[V]]      =   S[[E]]
   S ; mov V,E [[ip]]     =   S[[ip]] + 1
   S ; mov V1,E [[V2]]    =   S[[V2]]
      if V1 =/= V2 and V2 =/= ip
```

state that a `mov` instruction assigns the given value to the given variable, increments the instruction pointer by 1, and does not affect the value of any other variables.

The full Maude specification in Webster and Malcolm [24] gives a formal semantics for a subset of Intel 64.

2.3 Specifications as Interpreters, and Virtualization

Meseguer and Roşu [15, 16] give an overview of the many languages whose semantics have been specified in Maude, and point out that term rewriting provides interpreters for these languages: using equations to simplify terms effectively simulates the execution of programs. For example, the equations above give us

```
    s ; mov ecx, eax ; mov eax, ebx ; mov ebx, ecx [[ ebx ]]
  = s ; mov ecx, eax ; mov eax, ebx [[ ecx ]]
  = s ; mov ecx, eax [[ ecx ]]
  = s [[ eax ]]
```

which calculates that the program sets `ebx` to the value initially stored in `eax`; similarly, we could calculate that the program increments the instruction pointer by 3. Maude has a rewriting engine that automates this process of simplification using equations, and which can therefore be viewed as interpreting Intel 64 programs. In a very precise sense, this specification virtualizes Intel 64 programs: it provides a virtual machine on which these programs can be run. Webster and Malcolm [24] explore the ramifications of this for static and dynamic analysis of metamorphic viruses, and we further develop these ideas in the following sections. We will also argue that virtualization, to some extent, turns the tables in the battle between malware and anti-malware: on gaining control of a host machine, virtualizing malware becomes a defender of the resources that the virtualized anti-malware may used to detect its virtualized status, while the anti-malware may use stealth, obfuscation, or any of the techniques more usually associated with malware, to circumvent these countermeasures. The formal basis provided by a Maude specification of Intel 64 semantics allows us to reason rigorously about both malware and anti-malware.

3 Static and Dynamic Analysis

Webster and Malcolm [24] have shown that a Maude specification of the Intel 64 assembly programming language can be used for detection by dynamic analysis. In this section we will demonstrate how "equivalence in context" can be used for detection by static analysis.

In this section we describe how the Maude specification of Intel 64 can be used to reason about allomorphs of metamorphic computer viruses, using the Win9x.Zmorph.A virus as an example. We also present an improved form of a theorem proved in Webster and Malcolm [24].

3.1 Equivalence of Instructions

Our end goal is to be able to prove that two allomorphic sequences of viral code are equivalent, in that they behave in the same way. This means they have the same effect on all variables; for the remainder of this section we write V for the set of all variables available in Intel 64, including the flags and stack. This notion of equivalence is captured in

Definition 1. *For $W \subseteq V$, instructions p_1 and p_2 are W-*equivalent*, written $p_1 \equiv_W p_2$, iff for all stores s, and all variables $v \in W$:*

$$s; p_1[\![v]\!] = s; p_2[\![v]\!] \quad .$$

In the case that $W = V$, we say that p_1 is equivalent to *p_2, and write $p_1 \equiv p_2$.*

If $p_1 \equiv_W p_2$ then these instruction sequences may have different effects on variables that are not in W. However, if these instruction sequences are composed with another instruction sequence ψ whose behaviour does not depend on such variables, then we may have:

$$p_1; \psi \equiv p_2; \psi \quad .$$

If these conditions are met by some p_1, p_2 and ψ then we say that p_1 and p_2 are *equivalent-in-context of ψ*.

For the purposes of static analysis, we identify the variables that are read or written to by instructions. We identify $V_{out}(p)$ as the set of variables that could be modified by some instruction p.

Definition 2. *For instruction p, define $V_{out}(p)$ by $v \in V_{out}(p)$ iff there is an $s \in S$ such that $s; p[\![v]\!] \neq s[\![v]\!]$.*

For example, $V_{out}(\texttt{mov eax, ebx}) = \{\texttt{eax}, \texttt{ip}\}$ because the values in `eax` and `ip` are modified by this instruction.

Similarly, we want $V_{in}(p)$ to be the set of variables that could affect the behaviour of some instruction p in some way. We find it more convenient to express this by saying when a variable has no effect on the behaviour of p:

Definition 3. *For instruction p, define $V_{in}(p)$ by $v \notin V_{in}(p)$ iff for all $s, s' \in S$, $s \equiv_{V-\{v\}} s'$ implies $s; p \equiv_{V_{out}(p)} s'; p$.*

Additionally, these functions extend naturally to sequences of instructions:

Definition 4. *For instruction sequences ψ_1 and ψ_2:*

$$\begin{aligned} V_{in}(\psi_1; \psi_2) &= V_{in}(\psi_1) \cup V_{in}(\psi_2) \\ V_{out}(\psi_1; \psi_2) &= V_{out}(\psi_1) \cup V_{out}(\psi_2) \quad . \end{aligned}$$

Webster and Malcolm [24] present some basic results that allow the notion of equivalence to be applied to metamorphic viruses, principally Theorem 1 below. Their proof, however, uses a lemma that is proved by case-analysis on Intel 64 instructions, and therefore only holds for those specific instructions: the following proof removes this dependency on a particular language, using only the abstract properties of V_{in} and V_{out}.

Lemma 1. *For all instructions p and for all states s_1, s_2:*

$$s_1 \equiv_{V_{in}(p)} s_2 \quad \textit{implies} \quad s_1;p \equiv_{V_{out}(p)} s_2;p \ .$$

Proof. Let $x_1, \ldots, x_n$ be an enumeration of $V \setminus V_{in}(p)$, and let $s_{1,1}$ be some state identical to s_1, except

$$s_{1,1}[\![x_1]\!] = s_2[\![x_1]\!] \ .$$

Inductively, let $s_{1,i+1}$ be some state identical to s_i except

$$s_{1,i+1}[\![x_{i+1}]\!] = s_2[\![x_{i+1}]\!] \ .$$

By Definition 3, $s_1;p \equiv_{V_{out}(p)} s_{1,1};p \equiv_{V_{out}(p)} s_{1,2};p \equiv_{V_{out}(p)} \ldots \equiv_{V_{out}(p)} s_{1,n};p = s_2;p$, and therefore $s_1;p \equiv_{V_{out}(p)} s_2;p$, as desired.

As in Webster and Malcolm [24], this lemma allows us to incrementally chain together sets of variables into equivalences for instruction sequences with

Theorem 1. *Let ψ be an instruction sequence such that $\psi = p_1;p_2;\ldots;p_m$, where $p_{1\leq i\leq m}$ are instructions. If $p_1 \equiv_W p_2$ and for all j with $1 \leq j \leq m$*

$$V_{in}(p_j) \subseteq W \cup \bigcup_{i=1}^{j-1} V_{out}(p_i) \qquad (1)$$

then $p_1;\psi \equiv_{W\cup V_{out}(\psi)} p_2;\psi$.

It is possible to recover equivalence of instruction sequences from semi-equivalence in some cases. If $p_1 \equiv_W p_2$, then p_1 and p_2 may have different effects on variables in $V \backslash W$ (which we henceforth write as $\overline{W}$); but if all variables in $\overline{W}$ are overwritten in the same way by some instruction p, then this theorem allows us to "add" those variables until we cover all of V:

Corollary 1 (Equivalence-in-Context). *If $p_1 \equiv_W p_2$ and $p_1;\psi \equiv_{W\cup V_{out}(\psi)} p_2;\psi$ for instruction sequences p_1, p_2, ψ and $W \cup V_{out}(\psi) = V$, then $p_1;\psi \equiv p_2;\psi$.*

3.2 Examples Using Win9x.Zmorph.A

The following code excerpts were taken from the entry point of two different executables infected with Zmorph. This virus reconstructs its code instruction-by-instruction, pushing each one onto the stack [21]. Therefore the following code samples exhibit a part of Zmorph's decryption algorithm:

```
mov edi, 2580774443          mov ebx, 535699961
mov ebx, 467750807           mov edx, 1490897411
sub ebx, 1745609157          xor ebx, 2402657826
sub edi, 150468176           mov ecx, 3802877865
xor ebx, 875205167           xor edx, 3743593982
```

```
push edi                    add ecx, 2386458904
xor edi, 3761393434         push ebx
push ebx                    push edx
push edi                    push ecx
```

We shall refer to these two allomorphs as g and h respectively. In the following examples we will show that g and h are equivalent-in-context of two different instruction sequences, p and p', by applying the result from Corollary 1.

Before we begin, it is necessary to establish that if there is some sequence of instructions ψ for which $v \notin V_{out}(\psi)$, then the value of v is unchanged after executing ψ. We formalise this in

Proposition 1. *Let $\psi = \theta_1, \ldots, \theta_n$ be some sequence of instructions. Then for all stores $s \in S$, $s;\psi[\![v]\!] = s[\![v]\!]$ if $v \notin V_{out}(\psi)$.*

Proof is by induction. By Definition 4, we know that $v \notin V_{out}(\theta_i)$ for $0 \leq i \leq n$. By Definition 2, $s;\theta_1[\![v]\!] = s[\![v]\!]$ for all stores s. Let ψ_m be the subsequence of ψ consisting of the first m instructions in ψ, i.e., $\psi_m = \theta_1;\ldots;\theta_m$. Now, assume that $s;\psi_m[\![v]\!] = s[\![v]\!]$. Then by Definition 2, taking $s = \psi_m$ and $\theta = \theta_{m+1}$, we know that $s;\psi_{m+1}[\![v]\!] = s;\psi'[\![v]\!] = s[\![v]\!]$. Therefore $s;\psi[\![v]\!] = s[\![v]\!]$, as desired.

Example 1. *By inspection of the Maude specification of Intel 64, we know that*

$$V_{out}(g) \cup V_{out}(h) = \{\texttt{stack}, \texttt{ip}, \texttt{edi}, \texttt{ebx}, \texttt{ecx}, \texttt{edx}\}$$

By Proposition 1, we know that $s;g[\![v]\!] = s[\![v]\!]$ for all $v \notin V_{out}(g)$, and $s;h[\![v']\!] = s[\![v']\!]$ for all $v' \notin V_{out}(h)$. Therefore, $s;g[\![v]\!] = s;h[\![v]\!]$ for all $v \notin V_{out}(g) \cup V_{out}(h)$. Using the dynamic analysis approach of Webster & Malcolm [24] (i.e., using reductions in Maude), we can show that $s;g[\![\texttt{stack}]\!] = s;h[\![\texttt{stack}]\!]$ and $s;g[\![\texttt{ip}]\!] = s;h[\![\texttt{ip}]\!]$. Therefore we know that $s;g \equiv_W s;h$ where $\overline{W} = \{\texttt{edi}, \texttt{ebx}, \texttt{ecx}, \texttt{edx}\}$. (Note that for the sake of brevity, we have omitted the EFLAGS register in this example.)

We will show how an instruction sequence p executed immediately after g and h results in an equivalent store, which allows the metamorphic computer virus to freely swap g and h as long as p executes next.

Let $p =$ `mov edi, 0 ; mov ebx, 0; mov ecx, 0 ; mov edx, 0`. In order to apply Theorem 1, we must first check the values of $V_{in}(p_i)$ and $V_{out}(p_i)$ for all instructions p_i in p (these can be inferred easily by inspection of the Maude specification of Intel 64):

$$\begin{array}{lcl@{\qquad}lcl}
V_{in}(\texttt{mov edi, 0}) & = & \{\texttt{ip}\} & V_{out}(\texttt{mov edi, 0}) & = & \{\texttt{edi}, \texttt{ip}\} \\
V_{in}(\texttt{mov ebx, 0}) & = & \{\texttt{ip}\} & V_{out}(\texttt{mov ebx, 0}) & = & \{\texttt{ebx}, \texttt{ip}\} \\
V_{in}(\texttt{mov ecx, 0}) & = & \{\texttt{ip}\} & V_{out}(\texttt{mov ecx, 0}) & = & \{\texttt{ecx}, \texttt{ip}\} \\
V_{in}(\texttt{mov edx, 0}) & = & \{\texttt{ip}\} & V_{out}(\texttt{mov edx, 0}) & = & \{\texttt{edx}, \texttt{ip}\}
\end{array}$$

The following therefore hold:

$$\begin{array}{lcl}
V_{in}(\texttt{mov edi, 0}) & \subseteq & W \\
V_{in}(\texttt{mov ebx, 0}) & \subseteq & W \cup V_{out}(\texttt{mov edi, 0}) \\
V_{in}(\texttt{mov ecx, 0}) & \subseteq & W \cup V_{out}(\texttt{mov edi, 0}) \cup V_{out}(\texttt{mov ebx, 0}) \\
V_{in}(\texttt{mov edx, 0}) & \subseteq & W \cup V_{out}(\texttt{mov edi, 0}) \cup V_{out}(\texttt{mov ebx, 0}) \cup V_{out}(\texttt{mov ecx, 0})
\end{array}$$

Therefore by Theorem 1, $g;p \equiv_{W \cup V_{out}(p)} h;p$, and since $\overline{W} \subseteq V_{out}(p)$, we know by Corollary 1 that $g;p \equiv h;p$.

Alternatively, we can check directly using the Maude specification of Intel 64 that this is the case, using the above definitions of g, h and p. We can use Maude's term rewriting to simplify terms such as the following:

```
s ; g ; p[[stack]] == s ; h ; p[[stack]]
s ; g ; p[[ip]]    == s ; h ; p[[ip]]
s ; g ; p[[edi]]   == s ; h ; p[[edi]]
```

Each of these terms tests the equality of the two programs on the variables `stack`, `ip`, `edi`, etc. By testing for all the variables in Intel 64, we can take these Maude reductions as a second proof that $g;p \equiv h;p$ [25].

In the example above we showed that by overwriting the non-equivalent variables from the semi-equivalent programs g and h in the instruction sequence p, that we can show that g and h are equivalent-in-context of p. In the following example we will show that equivalence can also be demonstrated where an instruction sequence p' contains instructions which overwrite the non-equivalent variables, as long as the instructions in p' are not dependent on the non-equivalent variables.

Example 2. *Let $p' =$ `pop edi ; pop ebx ; pop ecx ; mov ecx, edx`. Once again we must check the values of $V_{in}(p'_i)$ and $V_{out}(p'_i)$ for all instructions p'_i in p' before we can apply Theorem 1:*

$$\begin{aligned} V_{in}(p'_1) &= \{\texttt{ip}, \texttt{stack}\} & V_{out}(p'_1) &= \{\texttt{edi}, \texttt{ip}\} \\ V_{in}(p'_2) &= \{\texttt{ip}, \texttt{stack}\} & V_{out}(p'_2) &= \{\texttt{ebx}, \texttt{ip}\} \\ V_{in}(p'_3) &= \{\texttt{ip}, \texttt{stack}\} & V_{out}(p'_3) &= \{\texttt{ecx}, \texttt{ip}\} \\ V_{in}(p'_4) &= \{\texttt{ip}, \texttt{ecx}\} & V_{out}(p'_4) &= \{\texttt{edx}, \texttt{ip}\} \end{aligned}$$

The following therefore hold:

$$\begin{aligned} V_{in}(p'_1) &\subseteq W \\ V_{in}(p'_2) &\subseteq W \cup V_{out}(p'_1) \\ V_{in}(p'_3) &\subseteq W \cup V_{out}(p'_1) \cup V_{out}(p'_2) \\ V_{in}(p'_4) &\subseteq W \cup V_{out}(p'_1) \cup V_{out}(p'_2) \cup V_{out}(p'_3) \end{aligned}$$

Therefore by Theorem 1, $g;p' \equiv_{W \cup V_{out}(p')} h;p'$, and since $\overline{W} \subseteq V_{out}(p')$, we know by Corollary 1 that $g;p' \equiv h;p'$.

As with the previous example, it is also possible to verify this directly using a reduction in Maude [25].

4 Detecting Metamorphism

In the previous sections we have shown how the formal specification in Maude of the Intel 64 assembly programming language enables static and dynamic analysis to prove equivalence and semi-equivalence of code. We have shown how metamorphic computer viruses use equivalent and semi-equivalent code in order to avoid detection by signature scanning. Therefore, given the techniques for code analysis described above, it seems reasonable that static and dynamic analysis based on the formal specification of Intel 64 should give ways to detect metamorphic computer viruses by proving the equivalence of different generations of the same virus to some virus signature, thus enabling detection of metamorphic computer viruses by a signature-based approach.

Implementation of a industrial tool for metamorphic computer virus detection is beyond the scope of this work, but a discussion of the application of the technique presented earlier to the problem of detecting metamorphic and virtualized malware is given below.

4.1 Dynamic Analysis for Detection of Metamorphic Code

4.1.1 Signature Equivalence

The most obvious application for detection is based on the techniques used by Webster and Malcolm [24], and in Section 2, to prove by dynamic analysis the equivalence of code fragments. Suppose that a signature σ is stored in a disassembled form, and that there is a fragment of suspect code c within a disassembled executable file. Then, the effects of c and σ on a generalised store could be discovered by performing Maude reductions. The resulting stores could be compared, and if equal, would prove that $c \equiv \sigma$. Computer virus signatures must be *sufficiently discriminating* and *non-incriminating*, meaning that they must identify a particular virus reliably without falsely incriminating code from a different virus or non-virus [8]. If a suspect code block was proven to have equivalent behaviour to a signature, this would result in identification to the same degree of accuracy as the original signature. (Since a signature uses a syntactic representation of the semantics of a code fragment to identify a viral behavioural trait, any equivalent signature must therefore identify the same trait.) If the code block is only semi-equivalent, then the accuracy of detection could be reduced. However if equivalence-in-context could be proven then accuracy would again be to the same degree as the original signature.

4.1.2 Signature Semi-Equivalence

It might be the case that a given metamorphic computer virus is known to write certain values onto the stack, and therefore the state of the stack at a certain point in the execution of the metamorphic virus could be a possible means of detection. In the work by Webster and Malcolm [24], two variants of the Win$9x$.Zmorph.A metamorphic computer virus were shown to be equivalent with respect to the stack, meaning that the state of the stack was affected in the same way by both generations of the virus. Therefore, the same technique could be used for detection. In this case, equivalence need not be proven, as the detection method relies on equivalence with respect to a subset of variables, i.e., semi-equivalence.

4.2 Static Analysis for Detection of Metamorphic Code

4.2.1 Formally-Verified Equivalent Code Libraries

One important result in the field of algebraic specification is the Theorem of Constants (p.38, [11]). Informally, the theorem states that any nullary operator (i.e., constant) used in a reduction within an algebraic specification system such as Maude, can be used as a variable in that reduction. This holds because the definition of variables within Maude is that they are actually constants within a supersignature, i.e., a variable in a Maude module is a constant within another module that encompasses it.

This lets us use constants in place of variables, e.g., for the reductions used in Examples 1 and 2 we use a constant `s` to denote any store s.

This means that the proofs of equivalence and semi-equivalence of the code fragments in propositions 2–4 still hold if we swap the program variable names for other program variable names of the same sort (e.g., we don't interchange stack variables and "ordinary" variables such as the `eax` register). For example, if

$$\texttt{push ebp ; mov ebp, esp} \equiv_W \texttt{push ebp; push esp; pop ebp} \tag{2}$$

where $W = V - \{ip\}$, then by the Theorem of Constants we can replace `ebp` with `eax`, and `esp` with `edx`, for example, and the statement of semi-equivalence still holds. Therefore, we might rephrase the above with a more standard mathematical notation, e.g.,

$$\texttt{push } x\texttt{; mov } x\texttt{, } y \equiv_W \texttt{push } x\texttt{; push } y\texttt{; pop } x \tag{3}$$

with the additional requirement that $x \neq y$ (which was implicit in Equation 2).

Therefore, if we know that metamorphic computer viruses might use a set of equations similar to Equation 3, then we may wish to build up a library of equivalent instruction lists based on those equations. In doing so we could decide, for instance, that all instances of the left-hand side of Equation 3 should be "replaced by" the right-hand side. If there was a metamorphic computer virus that exhibited only this kind of metamorphism, then we would have effectively created a normal form of the virus that would enable detection by straightforward signature scanning. Of course, this example is kept simple intentionally, and many metamorphic computer viruses will employ code mutation techniques which are far more complex, but the general idea of code libraries which are formally verified using a formal specification language, such as Maude, may be useful.

4.2.2 Equivalence in Context

As shown in Section 3 and in earlier work by Webster and Malcolm [24], metamorphic computer viruses can use semi-equivalent code replacement in order to produce syntactic variants in order to evade signature-based detection. The obvious advantage of this stratagem is that restricting metamorphism to code sequences that are equivalent limits the number of syntactic variants. An obvious example is that metamorphic computer viruses may wish to use code that treats all variables equivalently except the instruction pointer, i.e., equivalent code of differing length that is semi-equivalent with respect to every variable except the instruction pointer. Clearly, this will not pose a problem for the metamorphic computer virus as long as there is no part of its program that is dependent on the value of the instruction pointer at a given point after the mutated code.

It is likely, therefore, that a code segment c of a suspect executable will be semi-equivalent to some signature σ of a metamorphic computer virus. If it were possible to prove equivalence-in-context, i.e., that $c; \psi \equiv \sigma; \psi$, where ψ is some code appearing immediately after c in the suspect executable, then it would be known that σ was a successful match to c and detection of the virus would be achieved. (See Figure 1 for an illustrated example.) Another possible application of equivalence-in-context would be in the scenario where dynamic analysis was computationally-expensive. Equivalence-in-context can be proven using only static analysis, and therefore could limit the use of dynamic analysis.

5 Detection of Virtualization by Metamorphic Code Generation

In the previous sections we have described a methodology for detecting metamorphic malware using a formal algebraic specification of the Intel 64 assembly programming language. In this section we will

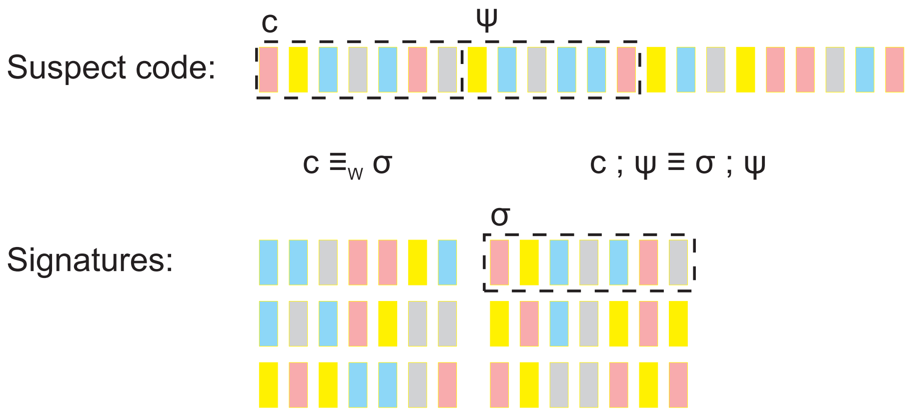

Figure 1: Signature-based detection of a metamorphic computer virus, by application of equivalence-in-context. Instruction sequences c and σ are semi-equivalent with respect to W. Applying the result in Corollary 1 to c, σ and ψ reveals that in fact $c; \psi \equiv \sigma; \psi$ and therefore c has been identified as equivalent to signature σ, resulting in detection of the virus. This method could result in a false positive as there may be a non-malware instruction sequence which is equivalent-in-context of some signature.

show how the same specification could be used to detect virtualization-based malware. Previously, we used the specification to prove that different generations of a metamorphic code were equivalent, i.e., we used reductions in Maude to simplify an Intel 64 instruction sequence to a term denoting the state of the computer after executing that instruction sequence. Here, we will show how we can essentially do the opposite: we can specify some end-condition for the state after executing some sequence of instructions, and using Maude's built-in search function, find sequences of instructions which satisfy that end-condition.

This is applicable to virtualization-based detection as follows. Suppose we have some Intel 64 instruction sequence which, when executed, can highlight the presence of virtualization-based malware. Naturally, virtualization-based malware will try to detect this instruction by signature matching, as part of a detection counter-measure. Therefore, it would be useful to be able to generate automatically sequences of instructions which we know are equivalent, and therefore would be difficult for the malware detect. In other words, we can use metamorphism to improve the performance of the detection method.

We can specify an end-condition in which the detection instruction sequence is stored in memory. Then, by applying the Maude search functionality, we can find sequences of instructions which generate this instruction sequence. The advantage of using the Intel 64 specification in Maude is that it is formal, and so any instruction sequence generated is automatically proven to work.

We will now describe the more technical details of this application of the Intel 64 specification.

5.1 Virtual Machine Rootkits

Virtual machine rootkits can be used to force the user to use an operating system that executes within a virtual machine [19, 13, 20, 10]. The advantages to the potential attacker are obvious; the user would be oblivious to any malicious programs executing outside the virtual machine. Rutkowska describes an approach to detection of virtualized malware from within the virtualized operating system, based on the execution of an Intel 64 assembly language instruction called SIDT x [19]. When executed, this instruction stores the contents of the interrupt descriptor table register into the destination operand x.

The value of x varies depending on whether the SIDT instruction has been executed inside or outside a virtual machine, and therefore detection is possible. This method is called *Red Pill*.

However, this detection method is not always guaranteed to work, as the user's interaction with the operating system can be controlled and manipulated in order to avoid detection using methods akin to Red Pill. King et al describe a counter-measure to Red Pill based on emulation [13]. The virtual machine monitor (VMM), which controls execution of the virtual machine, detects when the Red Pill executable is being loaded into memory, and sets a breakpoint to trap the execution of SIDT. When the breakpoint is reached, the VMM will emulate the instruction, setting the value of the destination operand of SIDT to a value not indicating detection. The authors note that this detection counter-measure could be defeated by a program R that generates the SIDT instruction dynamically.

At this point the writers of the malware have two options: they can re-write the virtualization-based malware so that it can detect R, as well as Red Pill, by static analysis. Alternatively, they can trace the execution of programs in order to detect by dynamic analysis any occurrence of Red Pill. King et al note that the latter could be computationally expensive, adding overhead which might result in detection by timing methods (see, e.g., [10]).

Suppose that the former option were chosen. Then, all the malware writers need do in order to avoid detection of their malware is to adjust their program to detect R' as well as R and Red Pill. Therefore, from the perspective of the writers of the Red Pill program, a means of automatic generation of programs that have the same behaviour as Red Pill would be desirable. In other words, we would like to use a metamorphic version of Red Pill, that changes its syntax at run-time in order to evade detection. Clearly, metamorphic engines as seen in metamorphic computer viruses could be used, but they are not reliable, in that the syntactic variants generated are not guaranteed to preserve the semantics of the original program. Therefore, we propose a solution to this problem based on our formal description of Intel 64 assembly language, which could be employed as a means of generating Red Pill variants before or during run-time.

5.2 Detecting Virtualization using the Intel 64 Specification

As was discussed in Section 2, the Maude specification of Intel 64 denotes a term rewriting system. The usual application of such a system is to apply equations and rewrite rules in order to reduce terms to some terminal form, i.e., to rewrite terms until they can no longer be rewritten. However, it is also possible to perform a search of the rewriting space of a term rewriting system in order to determine whether it is possible to reduce one term to another, and if there are non-deterministic aspects to the term rewriting system, whether there are multiple ways of performing such a reduction. It is also possible to test for some conditional value, and find all rewriting routes that lead to a term satisfying that condition.

Using the Maude specification of Intel 64, it is possible to rewrite a term such as `S[[eax]]`, which denotes the value of `eax` in some store `S`, using a variety of rewrite rules, and check using a breadth-first search of the term rewriting system whether a condition such as `S[[eax]] = "sidt"` is true, which says that the value of `eax` in some store `S` is equal to `"sidt"`. In other words, it is possible to create a term rewriting system in Maude that constructs programs based on rewrite rules, and search the rewriting space for constructed programs that are satisfy the requirement that `"sidt"` is stored in some variable. Figure 2 shows such a term rewriting system that generates different ways of constructing a program that satisfies the condition that `S[[eax]] = "sidt"`. Therefore, it is possible to create a metamorphic code engine based on our formal specification of Intel 64 in Maude.

```
rl [1] : S[[eax]]  =>  S ; mov ebx, "sidt" [[eax]] .
rl [2] : S[[eax]]  =>  S ; mov eax, ebx [[eax]] .
rl [3] : S[[eax]]  =>  S ; mov ecx, ebx [[eax]] .
rl [4] : S[[eax]]  =>  S ; mov eax, ecx [[eax]] .
```

Let the end condition be `s[[eax]] = "sidt"`.
Then, apply any of the following to reach the end condition from `s[[eax]]`:

$$(1,2),(1,2,3),(1,2,3,4),(1,3,4),(1,3,3,4),(1,3,\ldots,3,4).$$

Figure 2: A metamorphic engine based on the Maude specification of Intel 64. The four lines beginning with `rl` are rewrite rules that construct programs by appending an instruction to an instruction sequence. The search of the rewriting space then reveals the sequence of rewrite rule applications which culminated in an equivalent program. This sequence denotes the program, and therefore the syntactic variant can be inferred.

The previous example also shows how we can automatically generate programs that assign the number corresponding to the opcode of SIDT x to some variable, e.g., register `eax`. Therefore this technique could be used to generate automatically syntactically-mutated forms of a Red Pill program in order to evade detection of the Red Pill program by the VMM. This approach is advantageous to applying a metamorphic engine from a computer virus, which tend to be buggy, because the formality of the Intel 64 specification assures that any metamorphic code generated satisfies a given condition. If that condition is equivalence with respect to some variables, then we can generate syntactic variants of code which preserve semantics with respect to those variables.

5.3 A Note on Tractability

We described above how term rewriting systems can be specified in Maude, and used to generate metamorphic code. It is interesting to note that certain term rewriting systems, such as the one in Figure 2, there are an infinite number of terms satisfying the condition we have specified. Since each of these is generate by applying the rewriting rules in different sequences, we know that the set of terms satisfying the condition is infinite and recursively enumerable. Therefore, if we directed the Maude term rewriting engine to enumerate all the different terms satisfying a condition, the engine would never halt.

Therefore, it may appear that tractability is an issue in this regard. However, our aim is not to enumerate all of the different metamorphic programs that have the desired property, but to generate as many as we require in order to evade the detection counter-measures of the virtualization-based malware. For example, in Maude we can specify that we want only the first n programs that have the desired property. For example, we specified the rewriting system in Figure 2 in Maude version 2.3, and produced 1,000 programs satisfying the condition of assigning `"sidt"` to variable `eax` in approximately 0.36 seconds [25]. (The computer used was a Linux PC with a 3.2 GHz Intel Pentium 4 CPU and 1 GB of RAM.)

Therefore, it is practical to use Maude to generate programs with different syntax in order to evade the detection counter-measures employed by virtualization-based malware. In addition, this method is based on a formal specification of Intel 64, and therefore each of the generated programs is formally verified by Maude as it is generated.

6 Conclusion

In this paper we have demonstrated the applicability of formal algebraic specification to detection of metamorphic and virtualization-based malware. In order to improve the detection of metamorphic code, we have extended the applicability of equivalence-in-context to all programs in imperative programming languages through a redefinition of V_{out} and a new proof of Lemma 1. To show the applicability to metamorphic computer virus detection, we gave two worked examples of equivalence-in-context in action, and discussed the role of a formal model of the Intel 64 assembly language within the practical setting of anti-virus software. Finally, we gave a proof-of-concept system for generating metamorphic code in order to assist detection of virtualization-based malware by disabling detection counter-measures such as those used in the SubVirt system described by King et al [13].

6.1 Formal and Informal Approaches

Most of the approaches to metamorphic computer virus detection described above are based on some description of the syntax and semantics of a programming language. (The only exception is the approach of Yoo & Ultes-Nitsche [26, 27] to the detection of metamorphic computer viruses using neural networks, in which the semantics of the program being analysed are completed ignored, as the program code is treated only as data.) Perhaps then, the most distinctive feature of our approach to metamorphic computer virus detection is that the description of the programming language is both explicit *and* formal, i.e., it is based on a formal specification of the syntax and semantics of an assembly programming language written in a formal specification language. In contrast, many of the other approaches to detection, perhaps with the exception of the work by Christodorescu et al [6], are informal. For example, in control-flow analysis (e.g., [17, 14]), the flow of control is extracted from a program based on an implicit assumption about the way that looping instructions work, i.e., they update the value of the instruction pointer. Based on this assumption, the control-flow graph is constructed. Another example is Bruschi et al's approach to program rewriting and normalisation, in which a program is translated into a meta-representation based on an implicit knowledge of the behaviour of the program's instructions [1].

The advantage of a formal specification of the virus's programming language is that it is possible to prove properties of a section of code, which in turn allows for the development of methods of analysis which themselves are formally verifiable. A good example is the proofs of the equivalence of viral code in Section 3. Assuming that we know that the implicit formal specification in Maude is accurate, then given the existence of reduction as proof, then by performing reductions within Maude we can prove a property of a program (in this example, its equivalence to another program) using a number of reduction in Maude. Checking the accuracy of the formal specification is equivalent to checking the accuracy of the axioms within a logical system, that is, we formulate the formal specification of the Intel 64 assembly language with truths (i.e., axioms) that we hold to be self-evident. For example, in the specification of the MOV a, b instruction which assigns the value of variable b to variable a, then we specify that this the value of variable a after executing MOV a, b as equal to the value of b before we executed the instruction using the following equational rewrite rule, which expresses this truth formally:

```
eq S ; mov V,E [[V]] = S[[E]] .
```

The danger in using an implicit and/or informal description of the programming language is that our assumptions are not made clear, and therefore any detection method or program analysis based on the description may not do the job it is designed to do.

However, there is an obvious disadvantage to using a formal approach to program specification, verification and analysis. In order to reap the rewards of a formal specification of a programming language, first we must create it, which itself can be a time-consuming, but nevertheless straightforward, process. For example, in order to define the syntax and semantics of a 10-instruction subset of the Intel 64 assembly language instruction set for the proofs in Section 3, a Maude specification of around 180 lines had to be produced [25]. The main difficulty was not in the writing or debugging of the Maude specification, but rather in the translation from the informal and implicit definitions of the instructions given in the official Intel literature (see [12]).

Once created, though, a formal specification of an assembly programming language could be applied to a number of different problems in the field of computer virology. For example, the approach of Lakhotia and Mohammed to control- and data-flow analysis resulted in a rewritten version of a program called a zero form [17, 14]. The specification of Intel 64 could be used to prove the equivalence of the original program and its zero form through dynamic analysis in manner of Section 3. Another example would be in the code normalisation procedure described by Bruschi et al, in which the code is transformed into a meta-representation [1]. A formal specification of the syntax and semantics of the meta-representation could be written in Maude in a similar manner to the Maude specification of Intel 64, and the translation of the Intel 64 into the meta-representation could be then formally verified through proofs that an instruction and the translated form have the same effect on a generalised store.

6.2 Future Work

6.2.1 Combination With Other Approaches

An obvious further application of the methods for computer virus detection described in Sections 3–5, and in [24], is to combine them with other means of metamorphic computer virus detection. For instance, the formally-verified equivalent code library described in Section 4.2.1 may not always result in reduction of every generation of a metamorphic computer virus to a normal form. However, the overall syntactic variance of the set of all generations may be significantly reduced, so that another technique may be used to enable detection. For instance, the neural network-based approach of Yoo & Ultes Nitsche [26, 27] relies on the identification of similar code structures, and therefore may be assisted by an equivalent code library.

6.2.2 Analysis of Virtualization-based Malware

As described in Section 2, a subset of the Intel 64 instruction set has been specified using algebraic specification in Maude. Expanding the current specification of 10 instructions to the full instruction set would provide a way of formally proving properties of programs written in the Intel 64 assembly language. In addition to this, the formal specification is executable, and therefore once we have fully described the syntax and semantics of the language, we obtain an interpreter “for free” [16]. The development of such a specification is well within the reach of specification languages like Maude [16, 11], and therefore we propose the use of Maude for the formal proofs on assembly language programs, e.g., [24].

In addition, a specification in Maude of the full Intel 64 instruction set would be a virtual machine (in a very precise sense), because it would simulate an Intel 64 processor. Whilst the advanced features of virtual machine software (e.g., full operating system simulation), such as would be more difficult to specify, the Maude specification of the whole instruction set would enable the simulation of virtualization-based malware at a low-level of abstraction without major modification. For example, we could simulate the modification of the boot sector, a critical phase of the infection process of some virtualization-based malware (e.g., SubVirt [13]).

Acknowledgements

We would like to thank the reviewers for their comments, which we have found indispensable in improving our paper.

References

[1] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In R. Büschkes and P. Laskov, editors, *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, volume 4064 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2006.

[2] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of the International Symposium on Secure Software Engineering*, 2006.

[3] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Code normalization for self-mutating malware. *IEEE Security & Privacy*, 5(2):46–54, 2007.

[4] David M. Chess and Steve R. White. An undetectable computer virus. In *Virus Bulletin Conference*, September 2000.

[5] Mohamed R. Chouchane and Arun Lakhotia. Using engine signature to detect metamorphic malware. In *Proceedings of the Fourth ACM Workshop on Recurring Malcode (WORM)*, pages 73–78, 2006.

[6] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46. ACM Press, 2005.

[7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[8] Eric Filiol. *Computer Viruses: from Theory to Applications*, chapter 5, pages 151–163. Springer, 2005. ISBN 2287239391.

[9] Eric Filiol and Sébastien Josse. A statistical model for undecidable viral detection. *Journal in Computer Virology*, 3:65–74, 2007.

[10] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *11th Workshop on Hot Topics in Operating Systems (HOTOS-X)*, 2007.

[11] Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. Massachusetts Institute of Technology, 1996. ISBN 026207172X.

[12] Intel Corporation. *Intel®64 and IA-32 Architectures Software Developer's Manual*, November 2007. `http://www.intel.com/products/processor/manuals/index.htm` Accessed 19th March 2008.

[13] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[14] Arun Lakhotia and Moinuddin Mohammed. Imposing order on program statements to assist anti-virus scanners. In *Proceedings of Eleventh Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2004.

[15] José Meseguer and Grigore Roşu. The rewriting logic semantics project. In *Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005)*, volume 156 of *Electronic Notes in Theoretical Computer Science*, pages 27–56. Elsevier, 2005.

[16] José Meseguer and Grigore Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[17] Moinuddin Mohammed. Zeroing in on metamorphic computer viruses. Master's thesis, University of Louisiana at Lafayette, 2003.

[18] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. In *Proceedings of the 34th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, 2007.

[19] Joanna Rutkowska. Red Pill...or how to detect VMM using (almost) one CPU instruction. `http://www.invisiblethings.org/papers/redpill.html`, November 2004. Accessed 19th March 2008.

[20] Joanna Rutkowska. Subverting Vista^{TM} kernel for fun and profit. Black Hat Briefings 2006, Las Vegas, USA, August 2006. `http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf` Accessed 19th March 2008.

[21] Péter Ször. The new 32-bit Medusa. Virus Bulletin, December 2000.

[22] Peter Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005. ISBN 0321304543.

[23] Peter Ször and Peter Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference Proceedings*, 2001.

[24] Matt Webster and Grant Malcolm. Detection of metamorphic computer viruses using algebraic specification. *Journal in Computer Virology*, 2(3):149–161, December 2006. DOI: 10.1007/s11416-006-0023-z.

[25] Matt Webster and Grant Malcolm. Detection of metamorphic and virtualization-based malware using algebraic specification — Maude specification, January 2008. `http://www.csc.liv.ac.uk/~matt/pubs/maude/2/` Accessed 19th March 2008.

[26] In Seon Yoo and Ulrich Ultes-Nitsche. Non-signature based virus detection: Towards establishing a unknown virus detection technique using SOM. *Journal in Computer Virology*, 2(3), 2006.

[27] InSeon Yoo. Visualizing Windows executable viruses using self-organizing maps. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, 2004.