J Comput Virol (2009) 5:221–245
DOI 10.1007/s11416-008-0094-0

EICAR 2008 EXTENDED VERSION

Detection of metamorphic and virtualization-based malware using algebraic specification

Matt Webster · Grant Malcolm

Received: 20 January 2008 / Revised: 14 June 2008 / Accepted: 27 June 2008 / Published online: 22 July 2008

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 [14,21].

There are currently many avenues of research into the detection of metamorphic computer viruses, both academic and industrial. Lakhotia and Mohammed describe an algorithm for imposing order on high-level language programs based on control- and data-flow analysis [15,18]. Bruschi et al. [1] describe a similar method for malware detection to the one described by Lakhotia and Mohammed which uses code normalisation. Christodorescu et al. describe a formal approach to metamorphic computer 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. In a later paper, Preda et al. [19] are able to prove the correctness of this approach with respect to instruction reordering, variable renaming and junk code insertion. Bruschi et al. [2,3] and Walenstein et al. [25] describe approaches to code normalisation based on program rewriting. Chouchane and 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 and Ultes-Nitsche [29,30] 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 [23,24] describes some of the industrial techniques for the detection of metamorphic computer virus detection.

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

G. Malcolm
e-mail: Grant@liverpool.ac.uk

As virtualization-based malware is a relatively recent phenomenon [14,21], there is less in the literature on the problem of its detection. King et al. [14] 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. 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 [20] 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 (VMM) through an altered interrupt descriptor table.

Algebraic specification has been applied to the problem of metamorphic malware detection previously [26]. 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 equational term rewriting. 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'. The present paper builds upon this approach.

In Sect. 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. We also extend the earlier specification by giving a semantics for conditional and unconditional jumps. 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 [26]. In Sect. 3 we improve this result by showing that equivalence-in-context is applicable to all instructions in imperative programming languages; i.e., the earlier restriction is not necessary. We then give concrete examples of how equivalence-in-context can be used in practice to detect metamorphic malware, using allomorphs taken from the Win9x.Zmorph.A, Win95/Bistro and Win95/Zperm viruses. In Sect. 4, we discuss the applicability of the algebraic approaches given in Sect. 3 and [26] to the practical problem of detection of metamorphic malware based on formal static and dynamic analysis, and in Sect. 5 we give a proof-of-concept system for generating metamorphic variants of virtualization-detection programs (such as Red Pill [20]), based on the additional proof tools available in Maude.

2 Specifying Intel 64 assembly language

The Intel 64 instruction set architecture [13] (an extension of the Intel 32-bit architecture, IA-32) is used by the majority of personal computers worldwide., and it follows that many computer viruses will be manifest at some point in their reproductive cycle by a program in Intel 64. We have specified the syntax and semantics of a fragment of Intel 64 using Maude [7], a formal specification and verification framework. Maude is a language in the OBJ family of languages, which have been used for software specification and verification for over 30 years [12]. The full Maude specification of our fragment of Intel 64, which is described below, can be found in the appendix.

In this section, we describe our approach 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. In Sects. 2.1 and 2.2 we summarise the specification of the syntax and semantics of non-looping instructions (i.e., instructions that do not change the value of the instruction pointer), which has also been described in [26,28]. In Sect. 2.3, we describe how we have extended this approach to include looping instructions (i.e., jumps and conditional jumps). This extension of the specification describes a fragment of Intel 64 which is computationally more powerful, as we have conditional execution and iteration as well as variable assignment.

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 specific knowledge of Maude). The specification of the language declares sorts for the relevant syntactic categories, such as instructions, expressions, variables, etc., and declares 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 '`moves`' 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. All of these variable names can be declared as constants of sort `Variable`; for example

```
eax : -> Variable
```

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. Our 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 in starting state `s`. 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.

In practice, we can reduce a term like the one above to a simpler term denoting the value of `ebx` after executing the program. Maude applies the three equations above as rewrite rules, rewriting the term to a simpler term. In this case, applying the above equations to the term results in the following three-step simplification,

```
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 ]]
```

indicating that the value of `ebx` in the final store is equal to the value of `eax` in the initial store.

The semantics of non-looping instructions, such as `mov`, `or`, `xor`, `test`, `push`, `pop` and `nop` can be captured in this way.

2.3 Specifying the semantics of (conditional) loops

We described in the previous section how the semantics of non-looping instructions can be captured. To capture the semantics of looping instructions requires an extension of the specification which allows an arbitrary nesting and ordering of looping instructions. This extension uses an `exec_of_in_` operator, where `exec` p_1 `of` p_2 `in` s denotes that we are executing program p_1 in the store s, and that p_2 is the listing of the program from which we have derived p_1 (i.e., p_1 is the fragment of p_2 that follows the instruction pointer). We can make this clearer with an example. Suppose that we wish to execute the code

```
mov eax, 0 ; jmp label1 ; label1: mov ebx, 1
```

in an arbitrary store `s`. This would be captured by the following term:

```
exec mov eax, 0 ; jmp label1 ; label1: mov ebx, 1
of   mov eax, 0 ; jmp label1 ; label1: mov ebx, 1
in   s
```

Execution of the first instruction, `mov eax, 0`, results in an updated store `s'`. Then execution proceeds to the second instruction. Notionally, we have rewritten the term above to a second term:

```
exec jmp label1 ; label1: mov ebx, 1
of   mov eax, 0 ; jmp label1 ; label1:
mov ebx, 1 in   s'
```

which says that the next instruction to be executed is the jump to `label1`. (Notice that the store has been updated to `s'`, and that we have removed the first instruction from the list appearing after `exec`, but that the second list after `of` has stayed the same. The utility of this constant second list will soon become clear.) Next, execution proceeds to the third instruction, which is a `jmp` instruction, and can redirect the flow of control to anywhere else in the program. Our knowledge of the behaviour of `jmp` l is that it will execute the instruction that follows the label l. To capture the semantics of `jmp` l, we wish to update the first instruction list p_1 so that it starts at the point following the label l. This is where the second, constant list p_2 becomes useful. We specify a function that searches p_2 for an occurrence of the label l and updates the value of p_1 to the program that appears after the label l. In our running example, the term above rewrites as follows:

```
exec jmp label1 ; label1: mov ebx, 1
of   mov eax, 0 ; jmp label1 ; label1:
    mov ebx, 1
in   s'
   ===>
exec mov ebx, 1
of   mov eax, 0 ; jmp label1 ; label1:
    mov ebx, 1
in   s''
```

where `s''` is the state `s'` but with the instruction pointer updated to point to the instruction following `label1`.

Now suppose that we wish to capture the semantics of the `je` l instruction. Usually, `je` will appear after a calculation and will jump to label l if and only if the result of the last calculation is zero. In practice, the Intel 64 processor checks the results of all calculations and sets the zero flag (`zf`) to 1 if the calculation is equal to zero. The `je` l instruction is designed to jump if and only if the zero flag is equal to 1 (as a shorthand, we say that the zero flag is 'set'). Therefore the behaviour of `je` l is conditional on the value of the zero flag.

As described in the previous section, we associate variables with their values using the notion of a store, which is a function mapping variable names to values. Therefore, to capture the semantics of `je` l, we must query the store to check the value of the zero flag, and if it is set, then `je` l behaves exactly as `jmp` l. If the zero flag is not set, then `je` l behaves exactly as the `nop` ('no operation') instruction.

Therefore, to execute `je` l we must know the value of the zero flag. Within our specification, we know the value of any variable if (i) we know the initial value of the store, and (ii) we have a list of every instruction that has been executed thus far. Therefore, each time we evaluate `exec` p_1 `of` p_2 `in s` we do one of three things:

1. If the first instruction in p_1 is a non-looping instruction, then we remove it from the list p_1 and append it to s;
2. If the first instruction in p_1 is an unconditional jump (e.g., `jmp`), then we search for the place in the instruction following the target jump location, and update p_1 so that it contains the everything after this point;
3. If the first instruction in p_1 is a conditional jump (e.g., `je`), then we test the value of the variable(s) upon which the jump is conditional (e.g., the zero flag) relative to the current store s using the semantics of non-looping instructions given in the previous section.

Once again, we will make this clear with an example. Suppose we wish to execute the following:

```
sub eax, eax ; je label1 ; mov eax, 0 ;
    label1 ; mov ebx, 1
```

Now, we start to execute the program as before, but following the three rules above. The first instruction is non-looping, and therefore we invoke condition 1:

```
exec sub eax, eax ; je label1 ;
    mov eax, 0 ; label1 ; mov ebx, 1
of   sub eax, eax ; je label1 ;
    mov eax, 0 ; label1 ; mov ebx, 1
in   s
   ===>
exec je label1 ; mov eax, 0 ; label1 ;
    mov ebx, 1
    of   sub eax, eax ;
je label1 ; mov eax, 0 ; label1 ;
    mov ebx, 1
in   s ; sub eax, eax
```

Now, we want to execute the conditional jump `je label1`. To test whether `je label1` jumps to `label1` or not, we evaluate the value of the zero flag relative to the current store. To do this, we evaluate the value of the zero flag relative to the current store:

```
s ; sub eax, eax [[ zf ]]
```

We do this using our semantics of non-looping instructions. In this case, `sub eax, eax` assigns zero to the `eax` register, and therefore the zero flag is set, so the jump is performed:

```
exec je label1 ; mov eax, 0 ; label1 ;
    mov ebx, 1
of   sub eax, eax ;
je label1 ; mov eax, 0 ; label1 ;
    mov ebx, 1
```

```
in   s ; sub eax, eax
   ===>
exec mov ebx, 1
of   sub eax, eax ; je label1 ;
    mov eax, 0 ; label1 ; mov ebx, 1
in   s ; sub eax, eax
```

Execution of the next `mov` instruction proceeds as above, and we are left with an empty list in p_1:

```
exec
of   sub eax, eax ; je label1 ;
    mov eax, 0 ; label1 ; mov ebx, 1
in   s ; sub eax, eax ; mov ebx, 1
```

Therefore we have obtained a list of instructions which specifies the resulting state of the machine executing the instructions,

```
s ; sub eax, eax ; mov ebx, 1
```

where `s` is the initial state of the machine.

2.4 Specifications as interpreters, and virtualization

Meseguer and Roşu [16,17] give an overview of the many languages whose semantics have been specified in Maude, and reiterate the point made by Goguen and Malcolm [11] 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. In our earlier work [26] we explored 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 use 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 Equivalence of programs

Our earlier work [26] showed that a Maude specification of the Intel 64 assembly programming language can be used for detection by dynamic analysis. In this section, we demonstrate how equivalence of behaviour can be used for detection by static analysis. We present an improved form of a theorem from [26] and show how this can be used to reason about allomorphs of metamorphic computer viruses, using the Win9x.Zmorph.A, Win95/Zperm and Win95/Bistro viruses as examples.

3.1 Equivalence of states and programs

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. The notions of equivalence and behaviour are semantic notions, so our goal can be rephrased as being able to prove that two allomorphic sequences of viral code have the same semantics. In classical denotational semantics, programs denote functions from states to states, and a state is itself a function from variables to values. Our algebraic approach is less concrete in allowing states to be implemented in any way that satisfies the Maude specification of the semantics of the language, but the notion of equivalence is still the same: two programs are equivalent if they have the same effect on all variables. The technical machinery that we develop in this section applies to any imperative language with variables, though of course we are primarily interested in Intel 64, and this is the language used in the examples of Sects. 3.2 and 3.3. For the remainder of this section we assume there is a countable set V of variables in the language (for Intel 64, this would include the registers, flags, stack and memory addresses). We also write S for the set of states, which we refer to throughout as 'stores', following the terminology of the previous section. We write $s; p$ for the state that results from running program p in store s, and we write $s[\![v]\!]$ for the value that the store s assigns to the variable v. Thus, for example, $s; p[\![v]\!]$ represents the value of the variable v after p has been run in starting state s. We also assume that the language has sequential composition, which we also denote with a semicolon, e.g., $p_1; p_2$.

Any program affects only a finite set of variables, and two programs may be 'partially equivalent' in that they have the same effect on some variables, but not necessarily all variables. We begin by defining partial equivalence of states.

Definition 1 For $W \subseteq V$, stores s_1 and s_2 are *W-equivalent*, written $s_1 \equiv_W s_2$, iff for all variables $v \in W$:

$$s_1[\![v]\!] = s_2[\![v]\!].$$

In the case that $W = V$, we say that s_1 *is equivalent to* s_2, and write $s_1 \equiv s_2$.

Similarly, programs p_1 and p_2 are *W-equivalent*, written $p_1 \equiv_W p_2$, iff for all stores s, and all variables $v \in W$:

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

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

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

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

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

A straightforward consequence of this definition is

Proposition 3 *Let $p = p_1; \ldots; p_n$. If $v \notin \bigcup_{i=1}^{n} V_{\text{out}}(p_i)$, then $v \notin V_{\text{out}}(p)$, and so for all stores s we have $s;\, p[\![v]\!] = s[\![v]\!]$.*

Similarly, we want $V_{\text{in}}(p)$ to be the set of variables that could affect the behaviour of some program 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 4 For program p, define $V_{\text{in}}(p)$ by $v \notin V_{\text{in}}(p)$ iff for all $s, s' \in S$, $s \equiv_{V\setminus\{v\}} s'$ implies $s;\, p \equiv_{V_{\text{out}}(p)} s';\, p$.

That is, v has no effect on p if running p in two states that differ only in the value of v has no effect on the variables that p affects (attention is restricted to $V_{\text{out}}(p)$ because the stores $s;\, p$ and $s';\, p$ may of course differ at v itself).

In our earlier work [26] we presented some basic results that allow the notion of equivalence to be applied to metamorphic viruses, principally Corollary 9 below. Our proof, however, uses a lemma that is proved by case-analysis on Intel 64 programs, and therefore only holds for those specific programs: the proof we give below removes this dependency on a particular language, since it uses only the abstract properties of V_{in} and V_{out}. First, we introduce a slight generalisation of the notion of equivalence, that allows us to ignore certain variables (for example, in Intel 64, we may wish to prove the equivalence of two instruction sequences of different lengths, which means we need to disregard the value of the instruction pointer after execution of the programs). For a subset $W \subseteq V$ we write $\overline{W}$ for the complement $V\setminus W$.

Definition 5 Let p be a program and $W \subseteq V$ a set of variables; we say that *p has local effect at W* iff for all stores s_1 and s_2, if $s_1 \equiv_{\overline{W}} s_2$ then $s_1;\, p \equiv_{\overline{W}} s_2;\, p$.

Note that $s_1 \equiv_{\overline{W}} s_2$ says that stores s_1 and s_2 differ only on the values of variables in W, so p has local effect at W means that any differences that can be observed after running p in the two stores are kept within W. For example, most Intel 64 programs have local effect on the instruction pointer: execution of each instruction will increase the instruction pointer, and two programs of different length will increase the instruction pointer by different amounts, but that might well be the only difference between the programs. The notion of local effect allows us to disregard such differences if we so desire. Note also that in the special case $W = \emptyset$, local effect simply states that a program produces the same results when run in equivalent stores.

Lemma 6 *For all programs p with local effect at $W \subseteq V$ and for all states s_1, s_2:*

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

Proof Let $x_1, \ldots, x_n$ be an enumeration of $(V\setminus V_{\text{in}}(p))\setminus W$, 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

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

Then $s_{1,n} \equiv_{\overline{W}} s_2$, so by local effect at W we have $s_{1,n};\, p \equiv_{\overline{W}} s_2;\, p$. Moreover, by Definition 4, $s_1;\, p \equiv_{V_{\text{out}}(p)} s_{1,1};\, p \equiv_{V_{\text{out}}(p)} s_{1,2};\, p \equiv_{V_{\text{out}}(p)} \cdots \equiv_{V_{\text{out}}(p)} s_{1,n};\, p$. It follows that for any $v \in V_{\text{out}}(p)\setminus W$, $s_1;\, p[\![v]\!] = s_{1,n};\, p[\![v]\!] = s_2;\, p[\![v]\!]$, as desired.

As a technical remark, the above proof assumes that $(V\setminus V_{\text{in}}(p))\setminus W$ is finite. For classical denotational semantics, where a store is a function from variables to values, it is straightforward to allow the set to be countably infinite: the required store $s_{1,n}$ is just the uniquely determined function that agrees with s_2 on $(V\setminus V_{\text{in}}(p))\setminus W$, and with s_1 everywhere else. In our algebraic setting, we need to take more care that the required store exists. The simplest way of doing so is to have a default value, such as `0`, for all variables, and impose a reachability constraint on stores, so that we consider only those stores that assign the default value to all but a finite number of variables. In the proof, we need then consider only the finite subset of $(V\setminus V_{\text{in}}(p))\setminus W$ on which s_1 and s_2 differ.

The use of local effect at W gives a greater generality than our earlier results [26], and allows us to ignore differences at W. For the sake of clarity, however, it is worthwhile stating the special case where $W = \emptyset$:

Corollary 7 *For all programs p and for all states s_1, s_2:*

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

This states that running p in stores that agree on all the variables that can affect the behaviour of p gives results that agree on all the variables that can be affected by p.

As in [26], the above lemma allows us to incrementally chain together sets of variables into equivalences for programs. The key result, in a general statement with local effects is

Theorem 8 *Let q be a program with local effect at W, and let p_1 and p_2 be programs with $p_1 \equiv_{U\setminus W} p_2$. If $V_{\text{in}}(q) \subseteq U$, then $p_1; q \equiv_{(U\cup V_{\text{out}}(q))\setminus W} p_2; q$.*

Proof For any store s, we have $s; p_1 \equiv_{U\setminus W} s; p_2$, so $s; p_1 \equiv_{V_{\text{in}}(q)\setminus W} s; p_2$ because $V_{\text{in}}(q) \subseteq U$. It follows from Lemma 6 that $s; p_1; q \equiv_{V_{\text{out}}(q)\setminus W} s; p_2; q$. Now for $v \in U\setminus W$ and $v \notin V_{\text{out}}(q)$, we have $s; p_1; q[\![v]\!] = s; p_1[\![v]\!] = s; p_2[\![v]\!] = s; p_2; q[\![v]\!]$, so we conclude that $p_1; q \equiv_{(U\cup V_{\text{out}}(q))\setminus W} p_2; q$ as desired.

Taking $W = \emptyset$, and allowing for general sequential compositions, we get

Corollary 9 *Let q be a program such that $q = q_1; q_2; \ldots; q_m$. If $p_1 \equiv_U p_2$ and for all j with $1 \le j \le m$*

$$V_{in}(q_j) \subseteq U \cup \bigcup_{i=1}^{j-1} V_{\text{out}}(q_i)$$

then $p_1; q \equiv_{U\cup V_{\text{out}}(q)} p_2; q$.

It is possible to recover equivalence of programs from U-equivalence in some cases. If $p_1 \equiv_U p_2$, then p_1 and p_2 may have different effects on variables in $\overline{U}$; but if all variables in $\overline{U}$ are overwritten in the same way by some program q, then this theorem allows us to 'add' those variables until we cover all of V, in which case we say that p_1 and p_2 are *equivalent-in-context of q*.

Corollary 10 (Equivalence-in-context) If $p_1 \equiv_U p_2$ and $V_{in}(q) \subseteq U$ and $U \cup V_{\text{out}}(q) = V$, then $p_1; q \equiv p_2; q$.

All of the above provides some technical support for static analysis of programs written in any imperative language. Of course, the determination of the sets $V_{\text{in}}(p)$ and $V_{\text{out}}(p)$ will depend upon the particular program p. We conclude this section by showing that our Maude specification of Intel 64 allows us to determine these sets for instructions of the assembly language.

Example 11 Suppose we wish to know the value of $V_{\text{out}}(\texttt{mov v1, v2})$. By Definition 2, we must show that for every program variable $v \in V_{\text{out}}(\texttt{mov v1, v2})$ that v is different after executing `mov v1,v2` in some store s. The semantics of `mov` is given by the following equations:

```
eq  S ; mov V,E [[V]]  =  S[[E]].
ceq S ; mov V1,E [[V2]]  =  S[[V2]]
     if V1 =/= V2 and V2 =/= ip.
```

From these, we may suspect that `v1` is in $V_{\text{out}}(\texttt{mov v1, v2})$. We can prove this by assuming that the values of program variables `v1` and `v2` in some store `s` are different. We can express this in Maude notation as

```
eq s[[v1]] = value1.
eq s[[v2]] = value2.
```

where `value1` and `value2` are the (numeric) values of `v1` and `v2`, respectively. Then, by performing reductions in Maude we can calculate the value of `v1` before and after executing `mov v1,v2`:

```
reduce s[[v1]].
result Int: value1
reduce s ; mov v1, v2[[v1]].
result Int: value2
```

These reductions tell us that the value of v1 has changed from `value1` to `value2` by executing `mov v1,v2`. Therefore, we know that `v1` $\in V_{\text{out}}(\texttt{mov v1, v2})$.

Example 12 We can determine $V_{\text{in}}(\theta)$ for an instruction θ based on the Maude specification of Intel 64. By the definition of V_{in}, we know that if there exist stores $s, s' \in S$ such that $s \equiv_{V\setminus\{v\}} s'$ and $s; \theta \not\equiv_{V_{\text{out}}(\theta)} s'; \theta$ then $v \in V_{\text{out}}(\theta)$. Inspection of the Maude specification might result in the suspicion that `v2` $\in V_{\text{in}}(\texttt{mov v1, v2})$. We can prove this by assuming that $s \equiv_{V\setminus\{\texttt{v2}\}} s'$, which we can specify in Maude as follows:

```
eq s[[v2]] = value1.
eq s'[[v2]] = value2.
ceq s[[V]] = s'[[V]]
   if V =/= v2.
```

The first two equations say that `v2` is different in stores `s` and `s'`, and the last equation says that every variable apart from `v2` has the same value in stores `s` and `s'`. Now, we can test using reductions in Maude whether the variables in $V_{\text{out}}(\texttt{mov v1, v2})$ are equal after executing `mov v1,v2`. Since $V_{\text{out}}(\texttt{mov v1, v2}) = \{\texttt{v1}, \texttt{ip}\}$, we can test these values using reductions:

```
reduce s ; mov v1, v2 [[v1]].
result: value1
```

g	h
`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`

Fig. 1 Allomorphic fragments of Win9x.Zmorph.A

```
reduce s' ; mov v1, v2 [[v1]].
result: value2

reduce s ; mov v1, v2 [[ip]].
result: 1 + s'[[ip]]

reduce s' ; mov v1, v2 [[ip]].
result: 1 + s'[[ip]]
```

We can see that the value of `ip` after executing `mov v1,v2` is the same in both stores, but the value of `v1` is different. Therefore, we know that `v2` $\in V_{\text{in}}$(`mov v1, v2`).

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 [22]. Therefore the code samples g and h in Fig. 1 exhibit a part of Zmorph's decryption algorithm.

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 10.

Example 13 By inspection of the Maude specification of Intel 64, we know that

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

By Proposition 3, we know that $s; g[\![v]\!] = s[\![v]\!]$ for all $v \notin V_{\text{out}}(g)$, and $s; h[\![v']\!] = s[\![v']\!]$ for all $v' \notin V_{\text{out}}(h)$. Therefore, $s; g[\![v]\!] = s; h[\![v]\!]$ for all $v \notin V_{\text{out}}(g) \cup V_{\text{out}}(h)$. Using the dynamic analysis approach of our earlier work [26] (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 Corollary 9, we must first check the values of $V_{\text{in}}(p_i)$ and $V_{\text{out}}(p_i)$ for all instructions p_i in p (these can be inferred easily by inspection of the Maude specification of Intel 64):

$$\begin{aligned}
V_{\text{in}}(\texttt{mov edi, 0}) &= \{\texttt{ip}\} \\
V_{\text{in}}(\texttt{mov ebx, 0}) &= \{\texttt{ip}\} \\
V_{\text{in}}(\texttt{mov ecx, 0}) &= \{\texttt{ip}\} \\
V_{\text{in}}(\texttt{mov edx, 0}) &= \{\texttt{ip}\} \\
V_{\text{out}}(\texttt{mov edi, 0}) &= \{\texttt{edi}, \texttt{ip}\} \\
V_{\text{out}}(\texttt{mov ebx, 0}) &= \{\texttt{ebx}, \texttt{ip}\} \\
V_{\text{out}}(\texttt{mov ecx, 0}) &= \{\texttt{ecx}, \texttt{ip}\} \\
V_{\text{out}}(\texttt{mov edx, 0}) &= \{\texttt{edx}, \texttt{ip}\}
\end{aligned}$$

The following therefore hold:

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

Therefore by Corollary 9, $g; p \equiv_{W \cup V_{\text{out}}(p)} h; p$, and since $\overline{W} \subseteq V_{\text{out}}(p)$, we know by Corollary 10 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 $g; p \equiv h; p$ [27].

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 14 Let p' = `pop edi ; pop ebx ; pop ecx ; mov ecx, edx`. Once again we must check the values of $V_{\text{in}}(p'_i)$ and $V_{\text{out}}(p'_i)$ for all instructions p'_i in p' before we

```
push ebp
mov ebp, esp
mov esi, dword ptr [ebp + 08]
test esi, esi
je 401045
```

```
push ebp
push esp
pop ebp
mov esi, dword ptr [ebp + 08]
or esi, esi
je 401045
```

Fig. 2 Allomorphic fragments of Win95/Bistro [24]

can apply Corollary 10:

$V_{\text{in}}(p'_1) = \{\texttt{ip}, \texttt{stack}\}$
$V_{\text{in}}(p'_2) = \{\texttt{ip}, \texttt{stack}\}$
$V_{\text{in}}(p'_3) = \{\texttt{ip}, \texttt{stack}\}$
$V_{\text{in}}(p'_4) = \{\texttt{ip}, \texttt{ecx}\}$

$V_{\text{out}}(p'_1) = \{\texttt{edi}, \texttt{ip}\}$
$V_{\text{out}}(p'_2) = \{\texttt{ebx}, \texttt{ip}\}$
$V_{\text{out}}(p'_3) = \{\texttt{ecx}, \texttt{ip}\}$
$V_{\text{out}}(p'_4) = \{\texttt{edx}, \texttt{ip}\}$

The following therefore hold:

$V_{\text{in}}(p'_1) \subseteq W$
$V_{\text{in}}(p'_2) \subseteq W \cup V_{\text{out}}(p'_1)$
$V_{\text{in}}(p'_3) \subseteq W \cup V_{\text{out}}(p'_1) \cup V_{\text{out}}(p'_2)$
$V_{\text{in}}(p'_4) \subseteq W \cup V_{\text{out}}(p'_1) \cup V_{\text{out}}(p'_2) \cup V_{\text{out}}(p'_3)$

Therefore by Corollary 9, $g; p' \equiv_{W \cup V_{\text{out}}(p')} h; p'$, and since $\overline{W} \subseteq V_{\text{out}}(p')$, we know by Corollary 10 that $g; p' \equiv h; p'$.

As with the previous example, it is also possible to verify this directly using a reduction in Maude [27].

3.3 Example using Win95/Bistro

Win95/Bistro applies equivalent sequence replacement to generate syntactic variants. Figure 2 shows two allomorphic fragments from Win95/Bistro. Previously, we proved equivalence of the two Bistro fragments by dividing them into sub-fragments [26]. However, using our semantics of looping instructions (see Sect. 2.3) we can now prove equivalence in a simpler, more natural way.

In both fragments, the instruction `je 401045` will jump if and only if the value of `esi` (which itself depends on the value of `ebp`) is equal to zero. Since there is no way to determine the value of `ebp`, we replace the source operand `dword ptr [ebp + 08]` with a constant, `dword1`. Then, if `dword1` is equal to zero, the instruction `je 401045` will cause a jump; if not, no jump will occur.

In order to prove equivalence of the two fragments, we assign the two fragments of Bistro to two different constants as follows:

```
eq prog1 = push ebp ; mov ebp, esp ;
           mov esi, dword1 ;
           test esi, esi ; je 401045 ;
           jmp l1 ; label 401045: mov flag, 1 ;
           label l1: end .

eq prog2 = push ebp ; push esp ;
           pop ebp ; mov esi, dword1 ;
           or esi, esi ; je 401045 ;
           jmp l1 ; label 401045: mov flag, 1 ;
           label l1: end .
```

(In each case we append a sequence of instructions that will test whether the jump has occurred. If the jump is successful, then the variable `flag` is set.)

To determine equivalence of the two fragments, we need only test that the values of the variables in $V_{\text{out}}(\texttt{prog1})$ and $V_{\text{out}}(\texttt{prog2})$ are the same. Using a similar method to Example 11, we know that

$$V_{\text{out}}(\texttt{prog1}) = V_{\text{out}}(\texttt{prog2}) = \{\texttt{ebp}, \texttt{esp}, \texttt{esi}, \texttt{stack}, \texttt{zf}, \texttt{sf}, \texttt{pf}, \texttt{cf}, \texttt{of}\},$$

(neglecting the instruction pointer). In the first case, we assume that `dword1` is equal to zero.

```
eq dword1 = 0.
```

Then, we perform reductions to test whether the variables in $V_{\text{out}}(\texttt{prog1})$ are equal. In addition, we test that the value of flag is the same after executing both fragments, showing that the `je 401045` instruction had the same behaviour in both fragments:

```
reduce exec prog1 of prog1 in s[[ebp]] is
       exec prog2 of prog2 in s[[ebp]].
result: true
reduce exec prog1 of prog1 in s[[esp]] is
       exec prog2 of prog2 in s[[esp]].
result: true

reduce exec prog1 of prog1 in s[[flag]] is
       exec prog2 of prog2 in s[[flag]].
result: true

    ...
```

We therefore know that all variables are treated in the same way by the two fragments, and therefore the two fragments are equivalent when `dword1` is equal to zero. Similarly, when we assume that `dword1` is not equal to zero, we find that the values of the variables are the same after executing both fragments. Therefore, the two fragments of Bistro are equivalent. (The complete proof script can be found in the appendix.) It is also possible to use Theorem 8 to show that the two program fragments of Fig. 2 are $\overline{\{\texttt{ip}\}}$-equivalent; i.e., they have the same effect except on the instruction pointer, since the two fragments are of different length.

3.4 Example using Win95/Zperm

The Win95/Zperm metamorphic computer virus generates syntactic variants by separating its code into fragments, which are joined using unconditional jump statements. In addition, junk code is inserted within the fragments. Szor and Ferrie [24] describe this process, and give three different examples of the Zperm obfuscation process. In their example, a five-instruction program is obfuscated by jumps. The original code is not given, so we simulate Zperm's obfuscation using a five-instruction program called `zperm1`. We then generate three different `zperm1` variants (see Fig. 3) using the schemas set out graphically by Szor and Ferrie. Wherever the authors indicate garbage code, we have inserted a `nop` instruction.

We can demonstrate the equivalence of these programs in a similar way to Win95/Bistro. First, we calculate

$$V_{\text{out}}(\texttt{zperm1}) = V_{\text{out}}(\texttt{zperm2}) = V_{\text{out}}(\texttt{zperm3}) = V_{\text{out}}(\texttt{zperm4}) = \{\texttt{eax}, \texttt{ebx}, \texttt{ecx}\}$$

(neglecting the instruction pointer). Then, we prove equivalence of the four programs using reductions in Maude.

Each of the programs has a different start (entry) point, which we are able to specify using the `exec` p_1 `of` p_2 `in` s described in Sect. 2.3. We assign the code appearing after `start:` to p_1, whilst p_2 and s contain the usual full program and store (respectively). For example, we can simulate execution of `zperm2` with the following:

```
exec mov eax, 0 ; mov ebx, 1 ; jmp l2 ;
     nop ; label l2:
     mov ecx, ebx ; jmp l1 ; nop ;
     label out ; end
of   label l1: mov ebx, eax ; mov eax,
     ecx ; jmp out ; nop ; mov eax, 0 ;
     mov ebx, 1 ; jmp l2 ; nop ;
     label l2: mov ecx, ebx ; jmp l1 ;
     nop ; label out: end
in   s
```

In general, we can describe the above as `exec start`n `of prog`n `in` s, where n corresponds to the number of the fragment from Fig. 3. To prove equivalence of these fragments we perform reductions with respect to the different variables in $V_{\text{out}}(\texttt{zperm1})$. We start with variable `eax`:

```
reduce exec start1 of prog1 in s[[eax]].
result: 1
```

```
 <<zperm1>>

start:
 mov eax, 0
 mov ebx, 1
 mov ecx, ebx
 mov ebx, eax
 mov eax, ecx
```

```
 <<zperm2>>

 label l1:
 mov ebx, eax
 mov eax, ecx
 jmp out
 nop
start:
 mov eax, 0
 mov ebx, 1
 jmp l2
 nop
 label l2:
 mov ecx, ebx
 jmp l1
 nop
 label out:
```

```
 <<zperm3>>

 label l1:
 mov ebx, 1
 jmp l2
 nop
 label l2:
 mov ecx, ebx
 jmp l3
 nop
 label l4:
 mov eax, ecx
 jmp out
start:
 mov eax, 0
 jmp l1
 label l3:
 mov ebx, eax
 jmp l4
 label out:
```

```
 <<zperm4>>

label l1:
 mov ecx, ebx
 mov ebx, eax
 jmp l2
 nop
 label l2:
 mov eax, ecx
 jmp out
start:
 mov eax, 0
 jmp l3
 nop
 label l3:
 mov ebx, 1
 jmp l1
 nop
 label out:
```

Fig. 3 Allomorphic fragments of Win95/Zperm

```
reduce exec start2 of prog2 in s[[eax]].
result: 1
reduce exec start3 of prog3 in s[[eax]].
result: 1
reduce exec start4 of prog4 in s[[eax]].
result: 1
```

Therefore, the four different variants of Zperm are equivalent with respect to `eax`. We can prove equivalence with respect to `ebx` and `ecx` in a similar way, thus proving that all four variants are equivalent. The full Maude proof script is available in the appendix.

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 described in our earlier work [26], and in Sect. 3, 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 our previous work [26], two variants of the Win9x.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 13 and 14 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 Sects. 3.2–3.4 still hold if we swap the program variable names for other program variable names of the same sort (e.g., we do not interchange stack variables and "ordinary" variables such as the `eax` register). For example, if

```
push ebp ; mov ebp,esp ≡ W push ebp ;
     push esp ; pop ebp                    (1)
```

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,

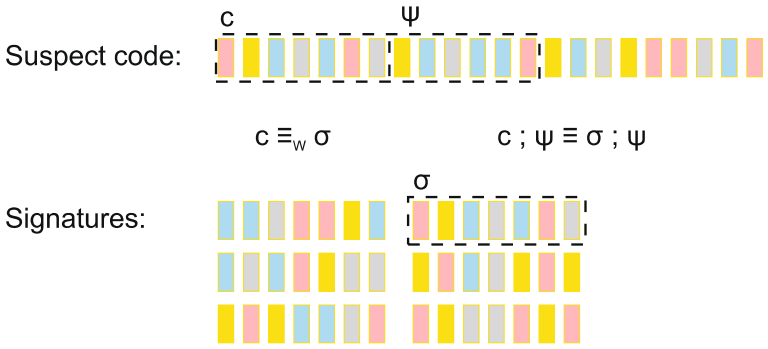

Fig. 4 Signature-based detection of a metamorphic computer virus, by application of equivalence-in-context. Instruction sequences c and σ are semi-equivalent with respect to W. Applying the result in Corollary 10 to c, σ and ψ reveals that in fact $c; \psi \equiv \sigma; \psi$ and therefore c has been 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

we might rephrase the above with a more standard mathematical notation, e.g.,

$$\texttt{push } x\texttt{; mov } x\texttt{, } y \equiv_W \texttt{push } x\texttt{; push } y\texttt{; pop } x \quad (2)$$

with the additional requirement that $x \neq y$ (which was implicit in Eq. 1).

Therefore, if we know that metamorphic computer viruses might use a set of equations similar to Eq. 2, 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 Eq. 2 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 Sect. 3 and in earlier work by ourselves [26], 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 Fig. 4 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 used our formal algebraic specification of the Intel 64 assembly programming language 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 [10,14,20,21]. 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 [20]. 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 [14]. The 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 Sect. 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 5 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.

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 for certain term rewriting systems, such as the one in Fig. 5, there are an infinite number of terms satisfying the condition we have specified. Since each of these is generated 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 Fig. 5 in Maude version 2.3, and produced 1,000 programs satisfying the condition of assigning `"sidt"` to variable `eax` in approximately 0.36 seconds [27]. (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.

```
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]]`:

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

Fig. 5 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 sequences of rewrite rule applications that result in programs that assign `"sidt"` to `eax`. The Maude specification for this proof-of-concept engine can be found in the appendix

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 Corollary 6. To show the applicability to metamorphic computer virus detection, we gave worked examples of equivalence of allomorphs of the Win9x.Zmorph.A, Win95/Bistro and Win95/Zperm viruses, 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. [14].

6.1 Formal and informal approaches

Most of the approaches to metamorphic computer virus detection described in Sect. 1 are based on some description of the syntax and semantics of a programming language. (The only exception is the approach of Yoo and Ultes-Nitsche [29,30] 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 less formal. For example, in control-flow analysis (e.g., [18,15]), 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 Sect. 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 reductions 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 ten-instruction subset of the Intel 64 assembly language instruction set for the proofs in Sect. 3, a Maude specification of around 180 lines had to be produced [27]. 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 [13]).

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 [18,15]. 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 Sect. 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 Sects. 3–5, and in [26], is to combine them with other means of metamorphic computer virus detection. For instance, the formally-verified equivalent code library described in Sect. 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 and Ultes Nitsche [29,30] 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 Sect. 2, a subset of the Intel 64 instruction set has been specified using algebraic specification in Maude. Expanding the current specification of ten 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" [17]. The development of such a specification is well within the reach of specification languages like Maude [11,17], and therefore we propose the use of Maude for the formal proofs on assembly language programs, e.g., [26].

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 [14]).

Acknowledgments We would like to thank the reviewers and participants of the 17th Annual European Institute for Computer Antivirus Research Conference (EICAR 2008) for their comments, which we have found indispensable in improving our paper.

Appendix A: Maude specification

The specification described in this paper was used with Maude 2.3 (built: Feb 14 2007 17:53:50). Maude is available online from http://maude.cs.uiuc.edu/. To input the Maude specification, it must be saved to some file f. Then, execute Maude and type `in` f. The same applies for the examples for Win95/Bistro, Win95/Zperm and the metamorphic engine for virtualization detection.

A.1 Intel 64 specification

```
*** This module defines the syntax of a subset of I-64.
fmod I-64-SYNTAX is
  protecting INT .
  sorts Variable Expression Stack EInt Label .
  sorts Instruction InstructionSequence .
  subsort Instruction < InstructionSequence .
  subsorts Variable EInt < Expression .
  subsort Int < EInt .

  op dadd_,_ : Variable Expression -> Instruction [prec 20] .
  op dsub_,_ : Variable Expression -> Instruction [prec 20] .

  *** I-64 instructions
  op mov_,_ : Variable Expression -> Instruction [prec 20].
  op add_,_ : Variable Expression -> Instruction [prec 20].
  op sub_,_ : Variable Expression -> Instruction [prec 20].
  op nop : -> Instruction.
  op push_ : Expression -> Instruction [prec 20].
  op pop_ : Variable -> Instruction [prec 20].
  op and_,_ : Variable Expression -> Instruction [prec 20].
  op or_,_ : Variable Expression -> Instruction [prec 20].
  op xor_,_ : Variable Expression -> Instruction [prec 20].
  op test_,_ : Variable Expression -> Instruction [prec 20].
  op label_ : Label -> Instruction [prec 20].
  op jmp_ : Label -> Instruction [prec 20].
  op je_ : Label -> Instruction [prec 20].

  *** helper operations
  op stackPush : Expression Stack -> Stack.
  op stackPop : Stack -> Stack.
  op stackTop : Stack -> EInt.
  op _next_ : EInt Stack -> Stack [prec 15].
  op stackBase : -> Stack.
  op msb : EInt -> EInt.
  op isZero : Expression -> EInt.
  op isZero : EInt -> EInt.
  op parity : EInt -> EInt.
  *** error messages
  op emptyStackError1 : -> Stack.
  op emptyStackError2 : -> EInt.
  *** I-64 registers
```

```
  ops eax ebx ecx edx ebp esp esi edi ip : -> Variable.
  *** I-64 EFLAGS register
  ops cf of sf af zf pf : -> Variable.
  *** equality operation
  op _is_ : EInt EInt -> Bool.
  op _is_ : Stack Stack -> Bool.
  *** extending the Int sort to include "undef"
  op undef : -> EInt.

  *** overloaded Boolean operations
  op _band_ : EInt EInt -> EInt [prec 35].
  op _bor_ : EInt EInt -> EInt [prec 35].
endfm

*** This module defines the semantics of the I-64 instructions
*** whose syntax is defined in I-64-SYNTAX.
fmod I-64-SEMANTICS is
  protecting I-64-SYNTAX.
  sort Store.

  *** stores
  ops s : -> Store.
  op initial : -> Store.

  *** operators for defining the semantics of I-64
  op _[[_]] : Store Expression -> EInt [prec 30].
  op _[[stack]] : Store -> Stack [prec 30].
  op _;_ : Store Instruction -> Store [prec 25].
  op _;_ : InstructionSequence InstructionSequence ->
        InstructionSequence [gather (e E) prec 26].
  *** variables for equations
  vars S S1 S2 S3 : Store.
  vars I I1 I2 I3 : EInt.
  vars INT INT1 INT2 : Int.
  vars V V1 V2 V3 : Variable.
  vars E E1 E2 E3 : Expression.
  vars ST ST1 ST2 : Stack.
  vars P P1 P2 : InstructionSequence.
  vars L L1 L2 L3 : Label.

  *** evaluation of instruction sequences
  eq S ; (P1 ; P2 ) = (S ; P1) ; P2.
  *** _is_ semantics
  eq I1 is I2 = (I1 == I2).
  eq ST1 is ST2 = (ST1 == ST2).
  *** the value of any integer in a store is the integer itself
  eq S[[I]] = I.
  *** initial values of variables and the stack
  eq initial[[stack]] = stackBase.
  ceq initial[[V]] = undef
    if V =/= ip.
  eq initial[[ip]] = 0.
```

```
*** Axioms to deal with static analysis of primitive
*** operators such as +, -, |, &, xor.
eq isZero(0) = 1.
eq I | I = I.
eq I & I = I.
eq (I1 + I2) is (I3 + I2) = I1 is I3.
eq (I1 + I2) is (I1 + I2) = true.
eq (I1 - I2) is (I1 - I2) = true.
eq (I1 | I2) is (I1 | I2) = true.
eq (I & S1[[V1]]) is (I & S2[[V2]]) = S1[[V1]] is S2[[V2]].
eq isZero(I1 & I2) is isZero(I1 & I2) = true.
eq parity(I1 & I2) is parity(I1 & I2) = true.
eq msb(I1 & I2) is msb(I1 & I2) = true.
eq isZero(I1 | I2) is isZero(I1 | I2) = true.
eq parity(I1 | I2) is parity(I1 | I2) = true.
eq msb(I1 xor I2) is msb(I1 xor I2) = true.
eq isZero(I1 xor I2) is isZero(I1 xor I2) = true.
eq parity(I1 xor I2) is parity(I1 xor I2) = true.
eq msb(I1 | I2) is msb(I1 | I2) = true.
eq (I1 xor I2) is (I1 xor I2) = true.

*** I-64 instruction semantics
eq S ; and V,E [[V]] = S[[V]] & S[[E]].
ceq S ; and V1,E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip and V2 =/= sf and V2 =/= zf
  and V2 =/= pf and V2 =/= cf and V2 =/= of.
eq S ; and V,E [[stack]] = S[[stack]].
eq S ; and V,E [[ip]] = S[[ip]] + 1.
eq S ; and V,E [[sf]] = msb( S[[V]] & S[[E]] ).
eq S ; and V,E [[zf]] = isZero( S[[V]] & S[[E]] ).
eq S ; and V,E [[pf]] = parity( S[[V]] & S[[E]] ).
eq S ; and V,E [[cf]] = 0.
eq S ; and V,E [[of]] = 0.

eq S ; or V,E [[V]] = S[[V]] | S[[E]].
ceq S ; or V1,E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip and V2 =/= sf and V2 =/= zf
  and V2 =/= pf and V2 =/= cf and V2 =/= of.
eq S ; or V,E [[stack]] = S[[stack]].
eq S ; or V,E [[ip]] = S[[ip]] + 1.
eq S ; or V,E [[sf]] = msb( S[[V]] | S[[E]] ).
eq S ; or V,E [[zf]] = isZero( S[[V]] | S[[E]] ).
eq S ; or V,E [[pf]] = parity( S[[V]] | S[[E]] ).
eq S ; or V,E [[cf]] = 0.
eq S ; or V,E [[of]] = 0.
eq S ; xor V,E [[V]] = S[[V]] xor S[[E]].
ceq S ; xor V1,E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip and V2 =/= sf and V2 =/= zf
  and V2 =/= pf and V2 =/= cf and V2 =/= of.
eq S ; xor V,E [[stack]] = S[[stack]].
eq S ; xor V,E [[ip]] = S[[ip]] + 1.
eq S ; xor V,E [[sf]] = msb( S[[V]] xor S[[E]] ).
```

```
eq S ; xor V,E [[zf]] = isZero( S[[V]] xor S[[E]] ).
eq S ; xor V,E [[pf]] = parity( S[[V]] xor S[[E]] ).
eq S ; xor V,E [[cf]] = 0.
eq S ; xor V,E [[of]] = 0.

eq S ; test V,E [[V]] = S[[V]].
ceq S ; test V1,E [[V2]] = S[[V2]]
  if V2 =/= ip and V2 =/= sf and V2 =/= zf
  and V2 =/= pf and V2 =/= cf and V2 =/= of.
eq S ; test V,E [[stack]] = S[[stack]].
eq S ; test V,E [[ip]] = S[[ip]] + 1.
eq S ; test V,E [[sf]] = msb( S[[V]] & S[[E]] ).
eq S ; test V,E [[zf]] = isZero( S[[V]] & S[[E]] ).
eq S ; test V,E [[pf]] = parity( S[[V]] & S[[E]] ).
eq S ; test V,E [[cf]] = 0.
eq S ; test V,E [[of]] = 0.

eq S ; mov V,E [[V]] = S[[E]].
ceq S ; mov V1,E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip.
eq S ; mov V,E [[stack]] = S[[stack]].
eq S ; mov V,E [[ip]] = S[[ip]] + 1.

eq S ; add V,E [[V]] = (S[[V]] + S[[E]]).
ceq S ; add V1, E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip.
eq S ; add V,E [[stack]] = S[[stack]].
eq S ; add V,E [[ip]] = S[[ip]] + 1.

*** special version of add ("dynamic add") that keeps
*** results of additions within I-64 limits (2^32-1).
eq S ; dadd V,E [[V]] = (S[[V]] + S[[E]]) & 4294967295.
ceq S ; dadd V1, E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip.
eq S ; dadd V,E [[stack]] = S[[stack]].
eq S ; dadd V,E [[ip]] = S[[ip]] + 1.

eq S ; sub V,E [[V]] = (S[[V]] - S[[E]]).
ceq S ; sub V1, E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip.
eq S ; sub V,E [[stack]] = S[[stack]].
eq S ; sub V,E [[ip]] = S[[ip]] + 1.
*** special version of add ("dynamic sub") that keeps
*** results of additions within I-64 limits (2^32-1).
eq S ; dsub V,E [[V]] = (S[[V]] - S[[E]]) & 4294967295.
ceq S ; dsub V1, E [[V2]] = S[[V2]]
  if V1 =/= V2 and V2 =/= ip.
eq S ; dsub V,E [[stack]] = S[[stack]].
eq S ; dsub V,E [[ip]] = S[[ip]] + 1.

eq S ; push E [[stack]] = stackPush(S[[E]],S[[stack]]).
ceq S ; push E [[V]] = S[[V]]
  if V =/= ip.
```

```
  eq S ; push E [[ip]] = S[[ip]] + 1.

  eq S ; pop V [[stack]] = stackPop(S[[stack]]).
  eq S ; pop V [[V]] = stackTop(S[[stack]]).
  ceq S ; pop V1 [[V2]] = S[[V2]]
    if V1 =/= V2 and V2 =/= ip.
  eq S ; pop V [[ip]] = S[[ip]] + 1.

  ceq S ; nop [[V]] = S[[V]]
    if V =/= ip.
  eq S ; nop [[stack]] = S[[stack]].
  eq S ; nop [[ip]] = S[[ip]] + 1.

  eq S ; label L [[V]] = S[[V]].
  eq S ; label L [[stack]] = S[[stack]].

  *** Stack helper operations semantics
  eq stackPush(I,ST) = I next ST.
  eq stackPop(I next ST) = ST.
  eq stackPop(stackBase) = emptyStackError1.
  eq stackTop(I next ST) = I.
  eq stackTop(stackBase) = emptyStackError2.
endfm

*** This module gives some helper operations for looping
*** instructions.
fmod HELPER-OPERATIONS is
  pr I-64-SEMANTICS.

  *** variables for equations
  vars S S1 S2 S3 : Store.
  vars I I1 I2 I3 : EInt.
  vars IN IN1 IN2 IN3 : Instruction.
  vars INT INT1 INT2 : Int.
  vars V V1 V2 V3 : Variable.
  vars E E1 E2 E3 : Expression.
  vars ST ST1 ST2 : Stack.
  vars P P1 P2 : InstructionSequence.
  vars L L1 L2 L3 : Label.

  op findSubProgram : Label InstructionSequence ->
                      InstructionSequence.
  op labelNotFoundError : -> InstructionSequence.
  op end : -> Instruction.
  eq findSubProgram(L, label L ; P) = P.
  eq findSubProgram(L, IN ; P) = findSubProgram(L, P).
  eq findSubProgram(L, end) = labelNotFoundError.
endfm

*** This module gives the semantics of looping instructions.
fmod I-64-JUMPS is
  pr I-64-SEMANTICS.
  pr HELPER-OPERATIONS.
```

```
  ops l1 l2 l3 : -> Label.
  op exec_of_in_ : InstructionSequence InstructionSequence Store
                   -> Store [prec 30].

  *** variables for equations
  vars S S1 S2 S3 : Store.
  vars I I1 I2 I3 : EInt.
  vars INT INT1 INT2 : Int.
  vars V V1 V2 V3 : Variable.
  vars E E1 E2 E3 : Expression.
  vars ST ST1 ST2 : Stack.
  vars P P1 P2 : InstructionSequence.
  vars L L1 L2 L3 : Label.

  *** NON-LOOP INSTRUCTIONS
  eq exec mov V,E ; P1 of P2 in S = exec P1 of P2 in S ; mov V,E.
  eq exec or V,E ; P1 of P2 in S = exec P1 of P2 in S ; or V,E.
  eq exec xor V,E ; P1 of P2 in S = exec P1 of P2 in S ; xor V,E.
  eq exec and V,E ; P1 of P2 in S = exec P1 of P2 in S ; and V,E.
  eq exec test V,E ; P1 of P2 in S = exec P1 of P2 in S ; test V,E.
  eq exec add V,E ; P1 of P2 in S = exec P1 of P2 in S ; add V,E.
  eq exec dadd V,E ; P1 of P2 in S = exec P1 of P2 in S ; dadd V,E.
  eq exec sub V,E ; P1 of P2 in S = exec P1 of P2 in S ; sub V,E.
  eq exec push E ; P1 of P2 in S = exec P1 of P2 in S ; push E.
  eq exec pop V ; P1 of P2 in S = exec P1 of P2 in S ; pop V.
  eq exec nop ; P1 of P2 in S = exec P1 of P2 in S ; nop.
  eq exec label L ; P1 of P2 in S = exec P1 of P2 in S ; label L.

  *** LOOP INSTRUCTIONS
  eq exec jmp L ; P1 of P2 in S =
     exec findSubProgram(L, P2) of P2 in S.
  ceq exec je L ; P1 of P2 in S =
      exec findSubProgram(L, P2) of P2 in S
      if S[[zf]] == 1.
  ceq exec je L ; P1 of P2 in S = exec P1 of P2 in S
    if S[[zf]] =/= 1.

  eq exec end of P2 in S = S.
endfm
```

A.2 Win95/Bistro example

```
fmod BISTRO1 is
  pr I-64-JUMPS.

  ops prog prog1 prog2 : -> InstructionSequence.
  ops dword1 dword2 : -> EInt.
  ops flag : -> Variable.
  ops 401045 : -> Label.

  *** dword1 is equal to zero
  eq dword1 = 0.
  *** BISTRO FRAGMENT 1
```

```
  eq prog1 = push ebp ; mov ebp, esp ; mov esi, dword1 ;
             test esi, esi ; je 401045 ; jmp l1 ;
             label 401045 ; mov flag, 1 ; label l1 ; end.
  *** BISTRO FRAGMENT 2
  eq prog2 = push ebp ; push esp ; pop ebp ; mov esi, dword1 ;
             or esi, esi ; je 401045 ; jmp l1 ; label 401045 ;
             mov flag, 1 ; label l1 ; end.
endfm

*** should all be true
red exec prog1 of prog1 in s[[ebp]] is
    exec prog2 of prog2 in s[[ebp]].
red exec prog1 of prog1 in s[[esp]] is
    exec prog2 of prog2 in s[[esp]].
red exec prog1 of prog1 in s[[stack]] is
    exec prog2 of prog2 in s[[stack]].
red exec prog1 of prog1 in s[[flag]] is
    exec prog2 of prog2 in s[[flag]].
red exec prog1 of prog1 in s[[zf]] is
    exec prog2 of prog2 in s[[zf]].
red exec prog1 of prog1 in s[[sf]] is
    exec prog2 of prog2 in s[[sf]].
red exec prog1 of prog1 in s[[pf]] is
    exec prog2 of prog2 in s[[pf]].
red exec prog1 of prog1 in s[[cf]] is
    exec prog2 of prog2 in s[[cf]].
red exec prog1 of prog1 in s[[of]] is
    exec prog2 of prog2 in s[[of]].

fmod BISTRO2 is
  pr I-64-JUMPS.

  ops prog prog1 prog2 : -> InstructionSequence.
  ops dword1 dword2 : -> EInt.
  ops flag : -> Variable.
  ops 401045 : -> Label.

  *** dword1 is not equal to zero
  *** BISTRO FRAGMENT 1
  eq prog1 = push ebp ; mov ebp, esp ; mov esi, dword1 ;
             test esi, esi ; je 401045 ; jmp l1 ;
             label 401045 ; mov flag, 1 ; label l1 ; end.
  *** BISTRO FRAGMENT 2
  eq prog2 = push ebp ; push esp ; pop ebp ; mov esi, dword1 ;
             or esi, esi ; je 401045 ; jmp l1 ; label 401045 ;
             mov flag, 1 ; label l1 ; end.
endfm

*** should all be true
red exec prog1 of prog1 in s[[ebp]] is
    exec prog2 of prog2 in s[[ebp]].
red exec prog1 of prog1 in s[[esp]] is
    exec prog2 of prog2 in s[[esp]].
```

```
red exec prog1 of prog1 in s[[stack]] is
    exec prog2 of prog2 in s[[stack]].
red exec prog1 of prog1 in s[[flag]] is
    exec prog2 of prog2 in s[[flag]].
red exec prog1 of prog1 in s[[zf]] is
    exec prog2 of prog2 in s[[zf]].
red exec prog1 of prog1 in s[[sf]] is
    exec prog2 of prog2 in s[[sf]].
red exec prog1 of prog1 in s[[pf]] is
    exec prog2 of prog2 in s[[pf]].
red exec prog1 of prog1 in s[[cf]] is
    exec prog2 of prog2 in s[[cf]].
red exec prog1 of prog1 in s[[of]] is
    exec prog2 of prog2 in s[[of]].
```

A.3 Win95/Zperm example

```
fmod ZPERM is
  pr I-64-JUMPS.

  ops prog1 prog2 prog3 prog4 start2 start3 start4 :
      -> InstructionSequence.
  ops out l4 : -> Label.

  *** ZPERM GENERATION 0
  eq prog1 = mov eax, 0 ; mov ebx, 1 ; mov ecx, ebx ;
             mov ebx, eax ; mov eax, ecx ; end.
  *** ZPERM GENERATION 1
  eq prog2 = label l1 ; mov ebx, eax ; mov eax, ecx ;
             jmp out ; nop ; mov eax, 0 ; mov ebx, 1 ;
             jmp l2 ; nop ; label l2 ; mov ecx, ebx ;
             jmp l1 ; nop ; label out ; end.
  *** ZPERM GENERATION 1 start point
  eq start2 = mov eax, 0 ; mov ebx, 1 ; jmp l2 ; nop ;
              label l2 ; mov ecx, ebx ; jmp l1 ; nop ;
              label out ; end.
  *** ZPERM GENERATION 2
  eq prog3 = label l1 ; mov ebx, 1 ; jmp l2 ; nop ; label l2 ;
             mov ecx, ebx ; jmp l3 ; nop ; label l4 ;
             mov eax, ecx ; jmp out ; mov eax, 0 ; jmp l1 ;
             label l3 ; mov ebx, eax ; jmp l4 ; label out ; end.
  *** ZPERM GENERATION 2 start point
  eq start3 = mov eax, 0 ; jmp l1 ; label l3 ; mov ebx, eax ;
              jmp l4 ; label out ; end.
  *** ZPERM GENERATION 3
  eq prog4 = label l1 ; mov ecx, ebx ; mov ebx, eax ; jmp l2 ;
             nop ; label l2 ; mov eax, ecx ; jmp out ; mov eax, 0 ;
             jmp l3 ; nop ; label l3 ; mov ebx, 1 ; jmp l1 ; nop ;
             label out ; end.
  *** ZPERM GENERATION 3 start point
  eq start4 = mov eax, 0 ; jmp l3 ; nop ; label l3 ; mov ebx, 1 ;
              jmp l1 ; nop ; label out ; end.
endfm
```

```
*** should be equal to 1
red exec prog1 of prog1 in s[[eax]].
red exec start2 of prog2 in s[[eax]].
red exec start3 of prog3 in s[[eax]].
red exec start4 of prog4 in s[[eax]].
*** should be equal to 0
red exec prog1 of prog1 in s[[ebx]].
red exec start2 of prog2 in s[[ebx]].
red exec start3 of prog3 in s[[ebx]].
red exec start4 of prog4 in s[[ebx]].
*** should be equal to 1
red exec prog1 of prog1 in s[[ecx]].
red exec start2 of prog2 in s[[ecx]].
red exec start3 of prog3 in s[[ecx]].
red exec start4 of prog4 in s[[ecx]].
```

A.4 Virtualization detection example

```
mod METAMORPHIC is
  pr I-64-SEMANTICS.

  op "sidt" : -> EInt. *** "sidt" is a special integer
  op s : -> Store.
  var S : Store.

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

search [1000] in METAMORPHIC :
  S =>+ S such that S[[eax]] is "sidt".
```

References

1. Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In: Büschkes, R., Laskov, P. (eds) Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), vol. 4064 of Lecture Notes in Computer Science, pp. 129–143. Springer, Heidelberg (2006)
2. Bruschi, D., Martignoni, L., Monga, M.: Using code normalization for fighting self-mutating malware. In: Proceedings of the International Symposium on Secure Software Engineering (2006)
3. Bruschi, D., Martignoni, L., Monga, M: Code normalization for self-mutating malware. IEEE Secur. Priv. **5**(2), 46–54 (2007)
4. Chess, D.M., White, S.R.: An undetectable computer virus. In: Virus Bulletin Conference, September (2000)
5. Chouchane, M.R., Lakhotia, A.: Using engine signature to detect metamorphic malware. In: Proceedings of the Fourth ACM Workshop on Recurring Malcode (WORM), pp. 73–78 (2006)
6. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy, pp. 32–46. ACM Press, New York (2005)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The Maude 2.0 system. In: Nieuwenhuis, R. (ed.) Rewriting Techniques and Applications (RTA 2003), number 2706 in Lecture Notes in Computer Science, pp. 76–87. Springer, Heidelberg (2003)
8. Filiol, E.: Computer Viruses: from Theory to Applications, chap. 5, pp. 151–163. Springer, Heidelberg (2005) ISBN 2287239391
9. Filiol, E., Josse, S.: A statistical model for undecidable viral detection. J. Comput. Virol. **3**, 65–74 (2007)
10. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is not transparency: VMM detection myths and realities. In: 11th Workshop on Hot Topics in Operating Systems (HOTOS-X) (2007)
11. Goguen, J.A., Malcolm, G.: Algebraic Semantics of Imperative Programs. Massachusetts Institute of Technology (1996) ISBN 026207172X
12. Goguen, J.A., Malcolm, G. (eds.) Software Engineering with OBJ: Algebraic Specification in Action. Kluwer, Boston (2000) ISBN 0792377575
13. Intel Corporation. Intel®64 and IA-32 Architectures Software Developer's Manual, November 2007. http://www.intel.com/products/processor/manuals/index.htm Accessed 14 June (2008)
14. King, S.T., Chen, P.M., Wang, Y.-M., Verbowski, C., Wang, H.J., Lorch, J.R.: SubVirt: Implementing malware with virtual machines. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy (2006)
15. Lakhotia, A., Mohammed, M.: Imposing order on program statements to assist anti-virus scanners. In: Proceedings of Eleventh Working Conference on Reverse Engineering. IEEE Computer Society Press, New York (2004)
16. Meseguer, J., Roşu, G.: The rewriting logic semantics project. In: Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005), vol. 156 of Electronic Notes in Theoretical Computer Science, pp. 27–56. Elsevier, Amsterdam (2005)
17. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theor. Comput. Sci. **373**(3), 213–237 (2007)
18. Moinuddin Mohammed.: Zeroing in on metamorphic computer viruses. Master's thesis, University of Louisiana at Lafayette (2003)
19. Preda, M.D., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. In: Proceedings of the 34th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL 2007) (2007)
20. Rutkowska, J.: Red Pill…or how to detect VMM using (almost) one CPU instruction. http://www.invisiblethings.org/papers/redpill.html, November 2004. Accessed 14 June 2008
21. Rutkowska, J.: Subverting Vista™ 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 14 June 2008
22. Ször, P.: The new 32-bit Medusa. Virus Bulletin, December 2000
23. Ször, P.: The Art of Computer Virus Research and Defense. Addison-Wesley, Resading (2005) ISBN 0321304543
24. Ször, P., Ferrie, P.: Hunting for metamorphic. In: Virus Bulletin Conference Proceedings, 2001
25. Walenstein, A., Mathur, R., Chouchane, M.R., Lakhotia, A.: Normalizing metamorphic malware using term rewriting. In: IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), 2006
26. Webster, M., Malcolm, G.: Detection of metamorphic computer viruses using algebraic specification. J. Comput. Virol. **2**(3), 149–161, (2006). doi:10.1007/s11416-006-0023-z
27. Webster, M., Malcolm, G.: 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 14 June 2008
28. Webster M., Malcolm, G.: Detection of metamorphic and virtualization-based malware using algebraic specification. In: Broucek, V., Filiol, E. (eds.) 17th European Institute for Computer Antivirus Research Annual Conference Proceedings (EICAR 2008), pp. 99–119, 2008
29. Yoo, I., Ultes-Nitsche, U.: Non-signature based virus detection: towards establishing a unknown virus detection technique using SOM. J. Comput. Virol. **2**(3), (2006)
30. Yoo, I.: Visualizing Windows executable viruses using self-organizing maps. In: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security, 2004