

Detection of Metamorphic Computer Viruses Using Algebraic Specification

Matt Webster and Grant Malcolm*

Abstract

This paper describes a new approach towards the detection of metamorphic computer viruses through the algebraic specification of an assembly language. Metamorphic computer viruses are computer viruses that apply a variety of syntax-mutating, behaviour-preserving metamorphoses to their code in order to defend themselves against static analysis based detection methods. An overview of these metamorphoses is given. Then, in order to identify behaviourally-equivalent instruction sequences, the syntax and semantics of a subset of the IA-32 assembly language instruction set is specified formally using OBJ – an algebraic specification formalism and theorem prover based on order-sorted equational logic. The concepts of equivalence and semi-equivalence are given formally, and a means of proving equivalence from semi-equivalence is given. The OBJ specification is shown to be useful for proving the equivalence or semi-equivalence of IA-32 instruction sequences by applying reductions – sequences of equational rewrites in OBJ. These proof methods are then applied to fragments of two different metamorphic computer viruses, Win95/Bistro and Win9x.Zmorph.A, in order to prove their (semi-)equivalence. Finally, the application of these methods to the detection of metamorphic computer viruses in general is discussed.

Keywords: Computer virus - Metamorphic - Detection - Algebraic specification - Assembly language.

1 Introduction

Computer viruses are typically segments of a stored program that when run are able to create a copy of themselves in another stored program. During

*Department of Computer Science, University of Liverpool, Liverpool, L69 3BX, UK.

this process of reproduction, it is possible for the virus to modify itself in some way. Metamorphic computer viruses essentially replace sequences of instructions with syntactically different (yet semantically equivalent) sequences of instructions in successive generations [10]. In this way the behaviour of each generation is the same, but the actual code is different. Typically, this is done in order to avoid static analysis based detection methods such as signature scanning, heuristic analysis and spectral analysis.

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

1.1 Types of Code Metamorphosis

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

1.1.1 Junk Code Insertion

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

- Code that reverses the effects of a previous instruction or instructions, thus making the previous instruction(s) and the inverse code into junk. For example, the instruction sequence `xchg eax,ebx ; xchg eax,ebx`

– which swaps the values in registers `eax` and `ebx` twice – would fall under this category. (Note that, throughout this paper, we use a semi-colon (;) to indicate sequential composition of assembly language instructions.)

- Code that performs a computation that is not utilised in any of the outputs of the program. For example, the first instruction in the following instruction list does nothing as the result is overwritten by the next instruction: `mov eax,0 ; mov eax,ebx`.

1.1.2 Variable Renaming

Variables are renamed in successive generations of metamorphic computer viruses such as `Win9x.Regswap` [10]. For instance, `mov eax,0 ; push eax ; pop ebx` could be replaced by the equivalent instruction sequence `mov ecx,0 ; push ecx ; pop ebx`.

1.1.3 Unconditional Jump Insertion

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

```
pop edx
mov edi,0004h
mov esi,ebp
mov eax,000Ch

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

1.1.4 Instruction Reordering

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

1.1.5 Equivalent Sequence Replacement

A sequence of instructions is replaced by equivalent sequences of instructions in order to generate syntactic variants. A good example of this can be seen

in the Win9x/Bistro virus (see Figure 1, §3.1).

1.1.6 Pseudo-Conditional Jump Insertion

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

1.1.7 Arithmetical/Boolean Mutation

Arithmetical and Boolean operations can be easily mutated into other, equivalent forms. A good example of this can be found in the Win95.Zmorph.A virus (see Figure 2, §3.2).

1.1.8 Payload Mutation

Some viruses only reproduce on certain days of the week, or when the hour of the day is an even number, for example. These conditionalities can be mutated by a metamorphic computer virus. The payload of the virus could also be mutated.

1.1.9 Pseudo Branching

Here, the same code is executed whether the condition of a conditional jump is true or not. For example, the following two code fragments are equivalent with respect to the `eax` register:

```
je label1          mov eax, 435098
mov eax, 435098    sub eax, 340934
sub eax, 340934    ...
jmp label2
label1:
mov eax, 435098
sub eax, 340934
label2:
...
```

This form of metamorphism has not been seen in any metamorphic computer virus, to the authors’ knowledge. It is included as a likely future development

of metamorphic computer viruses. This is justified with the following quote from Filiol et al [2] on the ethics of the computer virology community: “We cannot rely on a ‘wait and see’ approach, but we must anticipate technological evolutions.”

1.2 Related Work

An overview of both static and dynamic analysis based methods for computer virus detection is given by Filiol [1]. A general overview of metamorphic computer viruses in the wild is given by Ször [10]. Lakhotia and Mohammed have studied an approach to the detection of metamorphic computer viruses based on imposing order on high-level language statements in order to reduce the number of syntactic variants of programs [7, 9]. Yoo has explored detection methods for metamorphic computer viruses using artificial neural networks [12].

The work in this paper is an extension of a previous project, in which the algebraic specification of computer viruses and their environments was explored using Abstract State Machines and OBJ [11].

2 Specifying IA-32 Assembly Language

A new means toward the detection of metamorphic computer viruses has been developed. It relies on a formal specification in OBJ of the IA-32 instruction set. The OBJ specification can be used to calculate the effects on any variable of any instruction sequence. Thus it is possible to prove equivalence or semi-equivalence of IA-32 instruction sequences by applying reductions – sequences of equational rewrites – using the OBJ term rewriting engine.

2.1 A Brief Introduction to OBJ

OBJ is a formal notation and theorem prover based on algebraic specification [4]. OBJ can be used for software specification [3], as data types can be defined in OBJ as sorts in an order-sorted algebra. Operators on these sorts can be defined and given meaning using equational rewrite rules.

For example, the syntax of the natural numbers in Peano notation could be laid out in OBJ as follows:

```
obj PEANO is
  sort Nat .
  op 0 : -> Nat .
```

```

op s_ : Nat -> Nat .
op _+_ : Nat Nat -> Nat .
endo

```

The first line simply introduces the name (`PEANO`) of the specification. This specification uses only one sort of data: natural numbers, whose name (`Nat`) is declared in the second line. The next three lines declare three operators. The first, `0` is a nullary operator (i.e., a constant) and can be used by other non-nullary operators (which have an operand of sort `Nat`) to generate more complex terms. `s_` is a unary operator that takes a `Nat` and returns a `Nat`, and `_+_` is an infix binary operator that takes two `Nats` and returns a `Nat`. `s_` is the successor function, and `_+_` is addition.

In OBJ the semantics of operators are given using a series of equations, which can also be used as term rewriting rules. So, if we wanted to give the semantics for the `_+_` operator above, we could do this as follows:

```

obj PEANO-SEMANTICS is
  protecting PEANO . *** Import the PEANO module
  vars M N : Nat .
  eq M + 0      = M .
  eq M + s(N)   = s(M + N) .
endo

```

Now we have specified the semantics of addition, or rather, we have made the `_+_` operator behave as a operation which returns the value of the sum of two natural number operands in Peano notation. The `=` is effectively a rewriting operator, showing that the term on the left is rewritten to the term on the right. Using OBJ, we can perform a reduction (a series of rewrites) in order to reduce a term to the most reduced form possible, i.e. the OBJ interpreter keeps applying rewrite rules as long as there is a rewrite rule that will apply. A typical reduction using the specification above would be as follows:

```

OBJ> reduce s(s(s(0))) + s(s(0)) .
result Nat: s (s (s (s (s 0))))

```

An important notion in OBJ is that of reduction as proof. Since each of the equations above holds for the natural numbers, the above reduction is a proof that “ $s(s(s(0))) + s(s(0)) = s(s(s(s(s(0))))$ ”, or “ $3 + 2 = 5$ ” in Arabic numerals. Whilst trivial, this example demonstrates the expressive power of the OBJ formalism.

2.2 Specifying the Semantics of IA-32

The Intel Architecture 32-bit (IA-32) instruction set architecture [5], also known as *x86-32*, is used by the vast majority of personal computers worldwide, and it follows that the majority of computer viruses will (at some point in their reproductive cycle) be manifest as a sequence of IA-32 instructions. This section describes how a subset of IA-32 has been formally specified using OBJ.

Together, the OBJ modules `IA-32-SYNTAX` and `IA-32-SEMANTICS` define an order-sorted initial algebra with sorts `Store`, `Instruction`, `Variable`, `Expression`, `Stack`, `Int` and `EInt`. A complete listing of these modules is given in Appendix A; for a more detailed account of OBJ and algebraic specification, see [3, 4].

In IA-32, as in any other imperative language, computation is achieved by updating the state of the machine that interprets the instructions of the language. We use the sort `Store` to represent this state, which comprises the values stored in the registers, stack, and various flags of the IA-32 architecture.

Instructions (when executed) can modify the store, which we express in OBJ using the following operator:

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

We say that `_;_` takes a `Store` and an `Instruction` and returns a `Store`. This operator lets us calculate the effects of any sequence of instructions on a store. This mirrors classical denotational semantics, where an instruction sequence denotes a function that takes a starting state as an argument, and returns the updated state that results from running the instruction sequence in the starting state.

It is very useful to be able to calculate the effects of a sequence of instructions on a store, and for this purpose another operator is defined:

```
op _[[_]] : Store Expression -> EInt .
op _[[stack]] : Store -> Stack .
```

This time the operator is overloaded, so that we can calculate the value of an expression or the stack relative to a store. (`Expression` is a supersort which encapsulates sorts `EInt` and `Variable`.) Therefore the `_[[_]]` operator will let us calculate the effects of any sequence of instructions on any integer, variable or the stack.

The previous operators are the foundation for the algebraic specification of the IA-32 instruction set. On top of this foundation we can start to specify

the syntax and semantics of IA-32 instructions. For example, the syntax of the MOV instruction can be defined as follows:

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

Now we can specify the semantics of MOV, using equational rewriting rules. First we need some variables for the rewrite rules. We define these as follows:

```
vars I I1 I2 : EInt .
vars V V1 V2 V3 : Variable .
```

These variables are used as “wildcards” in the equational rewrite rules, and are not to be confused with the sort `Variable` of IA-32 variables used in the instruction sequences (such as registers `eax`, `ebx` and so on).

Now we can define the semantics of MOV (as given in the IA-32 Architecture Software Developer’s Manual [5]) by the equations:

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

As we can see, a store `S` modified by `mov V,E` is denoted by `S ; mov V,E`. We use OBJ variables to generalise all instances of the `mov` instruction, i.e. `mov V,E` includes all `mov` instructions with an IA-32 variable as the destination operand and an expression (an IA-32 variable or an integer) as the source operand. The first equation states that the value of `V` in `S` after executing `mov V,E` is equal to `S[[E]]`, which is the value of the expression `E` in store `S`. The second equation is conditional, and says that the value of any variable other than the destination operand (`V1`) and the instruction pointer (`ip`) is unchanged by the execution of `mov V1,E`. The third equation states that the stack is left unchanged by `mov V,E`. The final equation states that `mov V,E` always increments the instruction pointer. (For the sake of simplicity in the OBJ specification, all instructions are defined as having length equal to 1.)

In a similar way the semantics of `ADD`, `SUB`, `XOR`, `AND`, `OR`, `PUSH`, `POP` and `NOP` have also been defined for use in the equivalence proofs in §3. In principle, there is no reason to stop here; it would be quite feasible to specify the semantics of IA-32 in its entirety using OBJ. Indeed, equational logic formalisms such as OBJ have been shown to be a useful tool for the specification of imperative languages [8, 4, 3]. The full specification of this subset of IA-32 can be found in Appendix A.

2.2.1 Using the OBJ Specification as an Interpreter

When the syntax and semantics of a programming language are defined in an automated theorem prover such as OBJ, an interpreter and program analysis tool for that programming language are obtained “essentially for free” [8]. In §3 this interpreter is used to prove the equivalence or semi-equivalence of instruction sequences towards metamorphic computer virus detection.

Here is an example of how we can use this implicit interpreter to prove properties of programs. The following IA-32 instruction sequence swaps the values in registers `eax` and `ebx`:

```
mov ecx, eax ; mov eax, ebx ; mov ebx, ecx (1)
```

We can test this by performing some reductions using OBJ. For example, to test the value of `eax` after executing the instruction sequence we can perform the following reduction, in which `s` denotes the state of an arbitrary store prior to execution of the instruction sequence:

```
OBJ> red s ; mov ecx, eax ; mov eax, ebx ; mov ebx, ecx [[eax]] .  
result EInt: s[[ebx]]
```

By applying the rewriting rules that specify the semantics of the `mov` instruction, the OBJ term-rewriting engine has reduced

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

to `s[[ebx]]`, thus proving that the effect of (1) on any store `s` is to assign the value of `ebx` in store `s` to `eax`. We can also check that `ebx` has been assigned the original value of `eax`:

```
OBJ> red s ; mov ecx, eax ; mov eax, ebx ; mov ebx, ecx [[ebx]] .  
result EInt: s[[eax]]
```

Therefore, using the OBJ specification of IA-32, we have proven that (1) swaps the values in the registers `eax` and `ebx`.

2.3 Equivalence of Instruction Sequences

Since metamorphic computer viruses change their code when they reproduce, we are particularly interested in applying our semantics for IA-32 to show that two segments of code have the same behaviour.

We begin by defining notions of equivalent and partially equivalent behaviour for code segments. In order to do so, we first need to extend the

semantics described above to sequences of instructions. This is done by declaring a sort `InstructionSequence` which includes all the single instructions of IA-32 as “singleton sequences”:

```
subsort Instruction < InstructionSequence .
```

We also declare an operation for sequential composition:

```
op _;_ : InstructionSequence InstructionSequence
      -> InstructionSequence .
```

In IA-32 syntax, this operator appears as juxtaposition; we adopt the semi-colon notation here for the sake of explicit clarity. The semantics of sequential composition is captured by an equation that states that executing a sequential composition amounts to executing the first sequence, then executing the second sequence in the resulting store:

```
var S : Store .
vars P1 P2 : InstructionSequence .
eq S ; (P1 ; P2) = (S ; P1) ; P2 .
```

In the remainder of this section, we adopt a more standard mathematical notation. Let S , V , and I denote the sets of all stores, variables, and instructions, respectively; the variables V include the registers, stack and flags of IA-32. Let $[_[_]] : S \times V \rightarrow \mathbb{Z}$ and $[_;_] : S \times I \rightarrow S$, so that $s;p$ denotes the state of an updated store after executing instruction sequence p in store s , and $s;p[[v]]$ denotes the value of the variable v in updated store $s;p$, as described in the OBJ specification outlined above.

We say that two sequences of instructions are *equivalent* if and only if they behave equivalently with respect to the set of variables in the store, and that they are *semi-equivalent* if and only if they behave equivalently with respect to a subset of the set of variables in the store.

Definition 1. For $W \subseteq V$, instruction sequences 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 $p_1 \equiv p_2$.

Note that these notions of equivalence are in fact equivalence relations.

2.4 Equivalence in Context

Our end goal is to be able to prove that two allomorphic sequences of code are equivalent. In this section, we give some results that allow us to use static analysis in such proofs. If $p_1 \equiv_W p_2$ then these instruction sequences may have different effects on variables that are not in W . However, if these instruction sequences are composed with another instruction sequence ψ whose behaviour does not depend on such variables, then we may have:

$$p_1; \psi \equiv p_2; \psi$$

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

For the purposes of static analysis, we identify the variables that are read or written to by instructions. In particular, we want $V_{in}(\theta)$ to be the set of variables that could affect the behaviour of some instruction θ in some way.

Definition 2. For instruction θ , define $V_{in}(\theta)$ by $v \in V_{in}(\theta)$ iff there exist $s, s' \in S$ and $v' \in V$ such that $s \equiv_{V-\{v\}} s'$ and $s; \theta[[v']] \neq s'; \theta[[v']]$.

For example, $V_{in}(\text{mov eax, ebx}) = \{\text{ebx, ip}\}$ because the values in `ebx` and `ip` are accessed as a means of determining the output of this instruction.

Similarly, we identify $V_{out}(\theta)$ as the set of variables that could be modified by some instruction θ .

Definition 3. For instruction θ , define $V_{out}(\theta)$ by $v \in V_{out}(\theta)$ iff there is an $s \in S$ such that $s; \theta[[v]] \neq s[[v]]$.

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

These functions extend naturally to sequences of instructions:

Definition 4. For instruction sequences ψ_1 and ψ_2 :

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

Lemma 1. For all instructions θ and for all p_1, p_2 :

$$p_1 \equiv_{V_{in}(\theta)} p_2 \quad \text{implies} \quad p_1; \theta \equiv_{V_{out}(\theta)} p_2; \theta .$$

The proof of this lemma is by case-analysis on each instruction; this can be done using an OBJ proof script: see Appendix B for details. For example, the following proof script proves the case for $\theta = \text{mov } v_1, v_2$, since $V_{in}(\text{mov } v_1, v_2) = \{v_2, \text{ip}\}$ and $V_{out}(\text{mov } v_1, v_2) = \{v_1, \text{ip}\}$. The result of each reduction should be, and is, `true`.

ops v1 v2 : -> Variable .
eq s1[[v2]] = s2[[v2]] .
eq s1[[ip]] = s2[[ip]] .
reduce s1 ; mov v1, v2 [[v1]] is s2 ; mov v1, v2 [[v1]] .
reduce s1 ; mov v1, v2 [[ip]] is s2 ; mov v1, v2 [[ip]] .

Lemma 2. *If $p_1 \equiv_W p_2$ and $V_{in}(\theta) \subseteq W$ then:*

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

Proof. Assume $p_1 \equiv_W p_2$. By the previous lemma, we know that $p_1; \theta \equiv_{V_{out}(\theta)} p_2; \theta$, so we need only consider variables in W and not in $V_{out}(\theta)$. For any $w \notin V_{out}(\theta)$, we have $s; p_1; \theta[[w]] = s; p_1[[w]]$ and $s; p_2; \theta[[w]] = s; p_2[[w]]$, by Definition 3. If $w \in W$, then $s; p_1[[w]] = s; p_2[[w]]$ by assumption that $p_1 \equiv_W p_2$, so $s; p_1; \theta[[w]] = s; p_2; \theta[[w]]$ as desired. \square

Now we can incrementally chain together sets of variables into equivalences for instruction sequences with our main

Theorem 1. *Let ψ be an instruction sequence such that $\psi = \theta_1; \theta_2; \dots; \theta_m$, where $\theta_{1 \leq i \leq m}$ are instructions. If $p_1 \equiv_W p_2$ and for all j with $1 \leq j \leq m$*

$$V_{in}(\theta_j) \subseteq W \cup \bigcup_{i=1}^{j-1} V_{out}(\theta_i) \quad (2)$$

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

Proof. By induction on m . The base case, where $m = 1$, is shown in Lemma 2. For the induction step, assume $p_1 \equiv_W p_2$ and for $i \leq j \leq m$

$$V_{in}(\theta_j) \subseteq W \cup \bigcup_{i=1}^{j-1} V_{out}(\theta_i)$$

so that, by the induction hypothesis, $p_1; \psi' \equiv_{W \cup V_{out}(\psi')} p_2; \psi'$, where $\psi' = \theta_1; \theta_2; \dots; \theta_{m-1}$. Now apply Lemma 1 (taking the p_1 of that lemma to be $p_1; \psi'$, p_2 to be $p_2; \psi'$ and $V_{in}(\theta)$ to be $W \cup V_{out}(\psi')$, noting that $V_{out}(\psi) = V_{out}(\psi') \cup V_{out}(\theta_m)$), which gives $p_1; \psi \equiv_{W \cup V_{out}(\psi)} p_2; \psi$ as desired. \square

It is possible to recover equivalence of instruction sequences from partial equivalence in some cases. If $p_1 \equiv_W p_2$, then p_1 and p_2 may have different effects on variables in $V - W$ (which we henceforth write as \overline{W}); but if all variables in \overline{W} are overwritten in the same way by some instruction sequence ψ , despite the differences in \overline{W} , then p_1 is equivalent to p_2 in the context of ψ , as stated in our final

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

Proof. If $\overline{W} \subseteq V_{out}(\psi)$ then $p_1; \psi \equiv_{W \cup \overline{W}} p_2; \psi$. Since $W \cup \overline{W} = V$ it follows that $p_1; \psi \equiv p_2; \psi$. \square

In the following section the OBJ specification of IA-32 is used for dynamic analysis in order to prove equivalence/semi-equivalence of metamorphic computer virus code fragments. The proofs in this section are useful for a static analysis based approach to equivalence proving. The application of both the static and dynamic analysis based approaches to metamorphic computer virus detection are discussed in §3.

3 Proving Equivalence of Viral Code

Any two generations of the same metamorphic computer virus that differ syntactically are called *allomorphs*. Using the formal specification of IA-32 described in §2.2 it is possible to prove the equivalence or semi-equivalence of various allomorphs of metamorphic computer viruses using reductions in OBJ, by using the OBJ specification as an interpreter. The technique is used on allomorphic code fragments of two metamorphic computer viruses: Win95/Bistro and Win9x.Zmorph.A. The application of this technique to the detection of computer viruses is discussed in §3.3.

3.1 Example 1: Win95/Bistro

Win95/Bistro applies equivalent statement replacement to generate syntactic variants. Figure 1 shows two allomorphic fragments from Win95/Bistro.

The fragments have been divided up into three blocks each. The first two blocks consist of instructions which alter the state of the stack, the `ebp` register and the instruction pointer (`ip`). We can analyse the effects on these variables using an OBJ reduction. First we define two store operators `a` and `b`, one for each block:

```
ops a b : Store -> Store .
```

Next we define the instruction sequences corresponding to `a` and `b` (this is a shorthand that allows more concise use of the instruction sequences):

```
eq a(S) = S ; push ebp ; mov ebp, esp .
eq b(S) = S ; push ebp ; push esp ; pop ebp .
```

push ebp mov ebp, esp	push ebp push esp pop ebp
mov esi, dword ptr [ebp + 08] test esi, esi je 401045	mov esi, dword ptr [ebp + 08] or esi, esi je 401045
mov edi, dword ptr [ebp + 0c] or edi, edi je 401045	mov edi, dword ptr [ebp + 0c] test edi, edi je 401045

Figure 1: Allomorphic fragments of Win95/Bistro. [10]

Now, using the semantics of IA-32 as specified in OBJ, we can use a reduction to calculate the effects of any instruction sequence on any variable. We can also use the `_is_` operation to prove that two instruction sequences have the same effect on the same variable, and are therefore equivalent with respect to that variable.

Proposition 1. *a* is equivalent to *b* with respect to every variable apart from the instruction pointer, i.e. $a \equiv_W b$ where $W = V - \{\text{ip}\}$.

Proof. Since $V_{out}(a) = V_{out}(b) = \{\text{stack}, \text{ebp}\}$ we need only prove equivalence with respect to $\{\text{stack}, \text{ebp}\}$ and non-equivalence with respect to `ip`, since all other variables (i.e. those outside $V_{out}(a)$) will be unchanged.

```
OBJ> reduce a(s)[[stack]] is b(s)[[stack]] .
result Bool: true
OBJ> reduce a(s)[[ebp]] is b(s)[[ebp]] .
result Bool: true
reduce in IA-32 : a(s)[[ip]] is b(s)[[ip]]
result Bool: false
```

Therefore, $a(S)$ and $b(S)$ are equivalent with respect to every variable except the instruction pointer. □

Next we can tackle the second pair of allomorphic fragments. This time we define a constant, `dword1`, to stand for the value of `dword ptr [ebp + 08]`, which is the same in both fragments.

```
op dword1 : -> EInt .
```

We define c and d in a similar way to last time:

```
eq c(S) = S ; mov esi, dword1 ; test esi, esi .
eq d(S) = S ; mov esi, dword1 ; or esi, esi .
```

`test` performs a Boolean-and operation on its operands, and sets the value of three flags (`zf`, `sf` and `pf`) in the EFLAGS register according to the result, and sets the value of two other flags (`cf` and `of`) in EFLAGS to zero (no other memory locations are updated) [5]. `or` performs a Boolean-or operation on its operands, and sets the value of three flags (`zf`, `sf` and `pf`) in the EFLAGS register according to the result, and sets the value of two other flags (`cf` and `of`) in EFLAGS to zero (no other memory locations are updated) [5]. Clearly, a Boolean-and is not equivalent to a Boolean-or, however these two instructions are equivalent if the source and destination operands in both instructions are the same variable. The Win95/Bistro virus uses this fact to generate allomorphs. We express this truth, a result of the idempotent law of Boolean-and and Boolean-or, using two equations. (The `_band_` and `_bor_` operators are overloaded so that the equations will apply to extended integers such as `dword1`.)

```
op _band_ : EInt EInt -> EInt .
op _bor_  : EInt EInt -> EInt .
eq I bor I = I .
eq I band I = I .
```

Proposition 2. c is equivalent to d , i.e. $c \equiv d$.

Proof. Proof is with a reduction. Since

$$V_{out}(c) = V_{out}(d) = \{esi, ip, zf, sf, pf, cf, of\}$$

we need only test the values of these variables in order to prove equivalence.

```
OBJ> reduce c(s)[[esi]] is d(s)[[esi]] .
result Bool: true
OBJ> reduce c(s)[[ip]] is d(s)[[ip]] .
result Bool: true
OBJ> reduce c(s)[[zf]] is d(s)[[zf]] .
result Bool: true
OBJ> reduce c(s)[[pf]] is d(s)[[pf]] .
result Bool: true
OBJ> reduce c(s)[[sf]] is d(s)[[sf]] .
result Bool: true
```

```

OBJ> reduce c(s)[[cf]] is d(s)[[cf]] .
result Bool: true
OBJ> reduce c(s)[[of]] is d(s)[[of]] .
result Bool: true

```

Therefore, c is equivalent to d . □

The third pair of code fragments can be dealt with in a similar way to the second, as the same instructions are used.

We define another constant, `dword2`, to stand for the value of `[ebp + 0c]`, which is the same in both fragments.

```
op dword2 : -> EInt .
```

We define e and f in a similar way to c and d :

```

eq e(S) = S ; mov edi, dword2 ; or edi, edi .
eq f(S) = S ; mov edi, dword2 ; test edi, edi .

```

Proposition 3. e is equivalent to f , i.e. $e \equiv f$.

Proof. Proof is with a reduction. Since

$$V_{out}(e) = V_{out}(f) = \{esi, ip, zf, sf, pf, cf, of\}$$

we need only test the values of these variables in order to prove equivalence.

```

OBJ> reduce e(s)[[esi]] is f(s)[[esi]] .
result Bool: true
OBJ> reduce e(s)[[ip]] is f(s)[[ip]] .
result Bool: true
OBJ> reduce e(s)[[zf]] is f(s)[[zf]] .
result Bool: true
OBJ> reduce e(s)[[pf]] is f(s)[[pf]] .
result Bool: true
OBJ> reduce e(s)[[sf]] is f(s)[[sf]] .
result Bool: true
OBJ> reduce e(s)[[cf]] is f(s)[[cf]] .
result Bool: true
OBJ> reduce e(s)[[of]] is f(s)[[of]] .
result Bool: true

```

Therefore, e is equivalent to f . □

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

Figure 2: Allomorphic fragments of Win9x.Zmorph.A.

3.2 Example 2: Win9x.Zmorph.A

IA-32 code that was found after the disassembly of two Win9x.Zmorph.A allomorphs can be seen in Figure 2. It is known that this virus decrypts itself onto the stack from hardcoded numbers [6]. Therefore we would expect the code fragments in Figure 2 to be equivalent with respect to the stack.

In a similar way to the previous section, we assign the two allomorphs to $g(S)$ and $h(S)$ respectively.

Proposition 4. g and h are equivalent with respect to the stack, i.e. $g \equiv_W h$ where $W = \{\text{stack}\}$.

Proof. We prove this by performing a reduction, and checking equality of the two resulting stacks using the `_is_` operator.

```
OBJ> reduce g(s)[[stack]] is h(s)[[stack]] .
result Bool: true
```

Therefore $g \equiv_W h$ where $W = \{\text{stack}\}$. □

We can check the resulting state of the stack by performing an additional reduction:

```
OBJ> reduce a(s)[[stack]] .
result Stack: 1894369473 next (2281701373 next (2430306267 next
(s[[stack]])))
```

The original state of the stack is denoted by $s[[\text{stack}]]$, and the `_next_` operator delimits individual values placed on the stack.

Therefore, the two allomorphic fragments are equivalent (with respect to the stack) to the following IA-32 instruction sequence:

```
PUSH 2430306267 ; PUSH 2281701373 ; PUSH 1894369473
```

3.3 Application to Anti-virus Scanning

Once a programming language such as IA-32 has been specified using a formal notation and theorem prover such as OBJ, we obtain an interpreter (and program analysis tool) for that language “essentially for free” [8]. Therefore, a suspect code segment could be interpreted using the OBJ specification of IA-32 in order to check the behaviour of that code. For example, in §3.2, 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. Checking this behaviour for a suspect code fragment would be straightforward using the methods shown. Therefore, the IA-32 specification in OBJ could be applied as a means of code emulation based dynamic analysis. This technique could also be applied to the analysis of suspected malware that functions by causing a stack overflow.

An application to aid signature scanning would be to check whether a suspect code fragment behaved (semi-)equivalently to a signature of a metamorphic computer virus. 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 [1]. 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 (see §2.4) could be proven then accuracy would again be to the same degree as the original signature.

4 Conclusion

A new means toward detection of metamorphic computer viruses has been developed. The method works by formally specifying the semantics of a programming language (in this case, IA-32) using OBJ – a formal notation for algebraic specification and theorem proving. This specification can be used to prove the equivalence or semi-equivalence of sequences of IA-32 instructions. In addition, there are methods for proving equivalence from semi-equivalence (equivalence in context). These techniques are readily applicable to real-life metamorphic computer viruses, as the examples of equivalence and semi-

equivalence proofs for allomorphs of Win95/Bistro and Win9x.Zmorph.A have shown.

It is reasonable to suggest that it is possible to detect metamorphic computer viruses, which use syntactically different yet semantically equivalent code in successive generations in order to avoid detection, by proving equivalence between syntactically different allomorphs using the methods described in this paper. Potential applications include (but are not limited to) those in §3.3.

4.1 Future Work

So far a subset of the IA-32 instruction set has been specified using OBJ, but there is no reason why the entire instruction set could not be implemented, as several imperative programming languages have been specified using similar approaches [8, 4, 3]. A full specification of IA-32 would enable application of the detection techniques described in this paper to computer viruses that use instructions beyond the subset of IA-32 specified here (see Appendix A for the OBJ specification). The technique of (semi-)equivalence proof has been applied to two of the eight computer virus code metamorphosis types given in §1.1: equivalent sequence replacement and arithmetical/Boolean metamorphism. A practical extension of this work would be to extend and test the techniques shown here for other types of metamorphism. In §2.4 a method for proving equivalence in context was given. An extension of this would be to find further means of proving equivalence in context, which would aid the detection of metamorphic computer viruses that employ semi-equivalence based code metamorphosis.

References

- [1] Eric Filiol. *Computer Viruses: from Theory to Applications*, chapter 5, pages 151–163. Springer, 2005. ISBN 2287239391.
- [2] Eric Filiol, Marko Helenius, and Stefano Zanero. Open problems in computer virology. *Journal in Computer Virology*, 1:55–66, 2006.
- [3] Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. Massachusetts Institute of Technology, 1996. ISBN 026207172X.
- [4] Joseph A. Goguen, Timothy Walker, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph A. Goguen

- and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer Academic Publishers, 2000. ISBN 0792377575.
- [5] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual*, March 2006. http://www.intel.com/design/pentium4/manuals/index_new.htm Accessed 21st June 2006.
 - [6] Kaspersky Lab. Win95.Zmorph. <http://www.avp.ch/avpve/newexe/win95/zmorphp.stm>. Accessed 22nd June 2006.
 - [7] Arun Lakhotia and Moinuddin Mohammed. Imposing order on program statements to assist anti-virus scanners. In *Proceedings of Eleventh Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2004.
 - [8] José Meseguer and Grigore Roşu. The rewriting logic semantics project. In *Proceedings of Structural Operational Semantics 2005*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005. To appear. <http://fm.cs.uiuc.edu/~grosu/download/sos05.pdf>.
 - [9] Moinuddin Mohammed. Zeroing in on metamorphic computer viruses. Master's thesis, University of Louisiana at Lafayette, 2003.
 - [10] Peter Ször and Peter Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference Proceedings*, 2001.
 - [11] Matt Webster. Algebraic specification of computer viruses and their environments. In Peter Mosses, John Power, and Monika Seisenberger, editors, *Selected Papers from the First Conference on Algebra and Coalgebra in Computer Science Young Researchers Workshop (CALCO-jnr 2005)*. University of Wales Swansea Computer Science Report Series CSR 18-2005, pages 99–113, 2005. <http://www.csc.liv.ac.uk/~matt/>.
 - [12] InSeon Yoo. Visualizing windows executable viruses using self-organizing maps. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, 2004.

A OBJ Specification

Below is a listing of the OBJ modules IA-32-SYNTAX, IA-32-SEMANTICS and BITWISE. The first two modules specify the syntax and semantics of a

subset of the IA-32 instruction set, and import sorts and operations from the BITWISE module, which defines the syntax and semantics of bitwise Boolean operations for integers in Arabic numeral notation using Lisp operations (OBJ3 is built on top of GNU Common Lisp).

The modules below are sufficient for reproducing all of the equivalence proofs in this paper. These modules, along with proof scripts for Lemma 1 and the (semi-)equivalence of the fragments of Win95/Bistro and Win9x.-Zmorph.A are available online from <http://www.csc.liv.ac.uk/~matt/pubs/obj/1/> .

Notes

- The sort `EInt` stands for Extended Integer. `EInt` is a superset of `Int`, and is used so that we can extend the sort of integers to include a value `undef` so that certain operations that return an integer can also return a value that signifies that the result of the operation is undefined. For example, any uninitialised variable has the value `undef`.
- The `prec` keyword sets the precedence of the operator, and is used so that the user can omit parentheses when constructing complex strings.

Code Listing

```
*** Specification of a subset of the IA-32 instruction set.
*** Subset = {MOV, ADD, SUB, XOR, AND, OR, PUSH, POP, NOP}.
*** By Matt Webster, June 2006. OBJ3 version 2.0 was used.

*** This module specifies the 16 bitwise Boolean operations
*** for integers in Arabic numeral notation using built-in Lisp
*** operations (OBJ is built on top of GNU Common Lisp).
obj BITWISE is
  protecting INT .

  *** syntax of the 16 Boolean binary operations
  op _bone_ : Int Int -> Int [prec 35] .
  op _btwo_ : Int Int -> Int [prec 35] .
  op _bandcone_ : Int Int -> Int [prec 35] .
  op _bandctwo_ : Int Int -> Int [prec 35] .
  op _band_ : Int Int -> Int [prec 35] .
  op _bcone_ : Int Int -> Int [prec 35] .
  op _bctwo_ : Int Int -> Int [prec 35] .
```

```

op _bclr_ : Int Int -> Int [prec 35] .
op _bxnor_ : Int Int -> Int [prec 35] .
op _bor_ : Int Int -> Int [prec 35] .
op _bnand_ : Int Int -> Int [prec 35] .
op _bnor_ : Int Int -> Int [prec 35] .
op _borccone_ : Int Int -> Int [prec 35] .
op _borctwo_ : Int Int -> Int [prec 35] .
op _bset_ : Int Int -> Int [prec 35] .
op _bxor_ : Int Int -> Int [prec 35] .

vars H1 H2 : Int .

*** semantics of the Boolean binary operations
bq H1 bone H2 = (boole boole-1 H1 H2) .
bq H1 btwo H2 = (boole boole-2 H1 H2) .
bq H1 bandccone H2 = (boole boole-andc1 H1 H2) .
bq H1 bandctwo H2 = (boole boole-andc2 H1 H2) .
bq H1 band H2 = (boole boole-and H1 H2) .
bq H1 bccone H2 = (boole boole-c1 H1 H2) .
bq H1 bctwo H2 = (boole boole-c2 H1 H2) .
bq H1 bclr H2 = (boole boole-clr H1 H2) .
bq H1 bxnor H2 = (boole boole-eqv H1 H2) .
bq H1 bor H2 = (boole boole-ior H1 H2) .
bq H1 bnand H2 = (boole boole-nand H1 H2) .
bq H1 bnor H2 = (boole boole-nor H1 H2) .
bq H1 borccone H2 = (boole boole-orc1 H1 H2) .
bq H1 borctwo H2 = (boole boole-orctwo H1 H2) .
bq H1 bset H2 = (boole boole-set H1 H2) .
bq H1 bxor H2 = (boole boole-xor H1 H2) .
endo

*** This module defines the syntax of a subset of IA-32.
obj IA-32-SYNTAX is
  protecting BITWISE .
  protecting INT .
  sorts Store Instruction Variable Expression Stack EInt .
  subsorts Variable EInt < Expression .
  subsort Int < EInt .

op _[[[]]] : Store Expression -> EInt [prec 30] .
op _[[stack]] : Store -> Stack [prec 30] .

```

```

op _;_ : Store Instruction -> Store [prec 25] .
*** IA-32 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] .
*** 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 : EInt -> EInt .
op parity : EInt -> EInt .
*** error messages
op emptyStackError1 : -> Stack .
op emptyStackError2 : -> EInt .
op initial : -> Store .
*** IA-32 registers
ops eax ebx ecx edx ebp esp esi edi ip : -> Variable .
*** IA-32 EFLAGS register
ops cf of sf af zf pf : -> Variable .
*** stores
ops s : -> Store .
*** equality operation
op _is_ : EInt EInt -> Bool .
op _is_ : Stack Stack -> Bool .
*** extending the Int sort to include "undef"
op undef : -> EInt .
endo

*** This module defines the semantics of the IA-32 instructions
*** whose syntax is defined in IA-32-SYNTAX.
obj IA-32-SEMANTICS is

```

```

protecting IA-32-SYNTAX .

*** variables for rewriting rules
var S : Store .
vars I I1 I2 : EInt .
vars INT INT1 INT2 : Int .
vars V V1 V2 V3 : Variable .
vars E E1 E2 E3 : Expression .
vars ST ST1 ST2 : Stack .

*** _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 .
cq initial[[V]] = undef
  if V /= ip .
eq initial[[ip]] = 0 .

*** IA-32 instruction semantics
eq S ; and V,E [[V]] = S[[V]] band S[[E]] .
cq 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]] band S[[E]] ) .
eq S ; and V,E [[zf]] = isZero( S[[V]] band S[[E]] ) .
eq S ; and V,E [[pf]] = parity( S[[V]] band S[[E]] ) .
eq S ; and V,E [[cf]] = 0 .
eq S ; and V,E [[of]] = 0 .

eq S ; or V,E [[V]] = S[[V]] bor S[[E]] .
cq 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]] bor S[[E]] ) .

```

```

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

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

eq S ; test V,E [[V]] = S[[V]] .
cq 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]] band S[[E]] ) .
eq S ; test V,E [[zf]] = isZero( S[[V]] band S[[E]] ) .
eq S ; test V,E [[pf]] = parity( S[[V]] band S[[E]] ) .
eq S ; test V,E [[cf]] = 0 .
eq S ; test V,E [[of]] = 0 .

eq S ; mov V,E [[V]] = S[[E]] .
cq 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]]) band 4294967295 .
cq 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 .

```

```

eq S ; sub V,E [[V]] = (S[[V]] - S[[E]]) band 4294967295 .
cq 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 .

eq S ; push E [[stack]] = stackPush(S[[E]],S[[stack]]) .
cq 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]]) .
cq S ; pop V1 [[V2]] = S[[V2]]
  if V1 /= V2 and V2 /= ip .
eq S ; pop V [[ip]] = S[[ip]] + 1 .

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

*** 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 .

```

endo

B Proof of Lemma 1 for IA-32

Lemma 1. *For all instructions θ and for all p_1, p_2 :*

$$p_1 \equiv_{V_{in}(\theta)} p_2 \quad \text{implies} \quad p_1; \theta \equiv_{V_{out}(\theta)} p_2; \theta .$$

Proof. For any IA-32 assembly language instruction θ , the following generalised proof script (based on the OBJ specification of IA-32) can be used to prove Lemma 1. First, two store constants are defined:

```
ops s1 s2 : -> Store .
```

Next, we specify that $p_1 \equiv_{V_{in}(\theta)} p_2$ is true for arbitrary stores $s1$ and $s2$, by adding, for each $v \in V_{in}(\theta)$, the equation:

```
eq s1[[v]] = s2[[v]] .
```

Then we check that $p_1; \theta \equiv_{V_{out}(\theta)} p_2; \theta$ is implied. This is done by checking for each $v \in V_{out}(\theta)$ that the following reduction evaluates to `true` when interpreted by the OBJ term-rewriting engine:

```
reduce s1 ;  $\theta$  [[v]] is s2 ;  $\theta$  [[v]] .
```

□

For example, the following proof script proves Lemma 1 for $\theta = \text{mov } v_1, v_2$. From the IA-32 Manual specification [5] we define $V_{in}(\text{mov } v_1, v_2) = \{v_2, \text{ip}\}$ and $V_{out}(\text{mov } v_1, v_2) = \{v_1, \text{ip}\}$.

```
ops v1 v2 : -> Variable .
eq s1[[v2]] = s2[[v2]] .
eq s1[[ip]] = s2[[ip]] .
reduce s1 ; mov v1, v2 [[v1]] is s2 ; mov v1, v2 [[v1]] .
reduce s1 ; mov v1, v2 [[ip]] is s2 ; mov v1, v2 [[ip]] .
```

When the above proof script is run, the output after the reductions should be, and is, `true`:

```
OBJ> reduce s1 ; mov v1, v2 [[v1]] is s2 ; mov v1, v2 [[v1]] .
result Bool: true
OBJ> reduce s1 ; mov v1, v2 [[ip]] is s2 ; mov v1, v2 [[ip]] .
result Bool: true
```