Cautious Virus Detection in the Extreme

John Case and Samuel E. Moelius III

University of Delaware

{case,moelius}@cis.udel.edu

Abstract

It is well known that there exist viruses whose set of infected programs is *un*decidable. If a virus detector is to err on the side of caution with respect to such a virus, then it must label some perfectly innocent programs as being infected by the virus. Can there exist a virus whose set of infected programs is *so unwieldy* that any cautious virus detector must label *all but finitely many* programs as being infected by the virus — even when *infinitely many* programs are *not* infected by the virus? Although such viruses can exist, strong theoretical evidence is presented that such a virus is *un*likely to be encountered in the *real world*. Several of our proofs employ infinitary self-reference arguments.

Categories and Subject Descriptors F.1.1 [*Computation by Abstract Devices*]: Models of Computation—Computability Theory; K.6.5 [*Management of Computing and Information Systems*]: Security and Protection—Invasive software (e.g., viruses, worms, Trojan horses); F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Computability Theory, Recursive Function Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Security, Theory

Keywords Virus Detection, Co-isolated Sets, Simple Sets, Program Self-Reference, Recursion Theorems

1. Introduction

For several models of computer viruses, it has been shown that there exist viruses whose set of infected programs is undecidable [Coh89] and even Σ_1-complete [Adl88, ZZ04].[1] We call a virus detector (e.g., McAfee VirusScan, Norton AntiVirus, Sophos Anti-Virus) *cautious with respect to virus* v iff the detector *never* labels a program *infected* by v as being *un*infected by v, but possibly labels some programs *un*infected by v as being infected by v.[2] (We omit *with respect to* v when it is clear from the context.) For a virus whose set of infected programs is undecidable, any cautious virus detector *must* label some perfectly innocent programs as being infected by the virus. The question is: *how many* innocent programs must suffer such a fate? In particular, can there exist a virus whose set of infected programs is *so unwieldy* that any cautious virus detector must label *all but finitely many* programs as being infected by the virus — even when *infinitely many* programs are *not* infected by the virus?

Herein, we show that such a virus can exist (Proposition 1, Section 1.3). However, we also show (Theorem 2, Section 3) that all such viruses have a property that we call $\neq$*-programmability* (Section 1.4). We argue that it is *un*likely that a *real world* virus would have this property (Thesis 1, Section 1.4). As such, our results are strong theoretical evidence that, for any real world virus, a cautious virus detector will (correctly) label infinitely many programs as *un*infected by the virus (Thesis 2, Section 1.4).

We model a virus as a computable function, as is similarly done in [Adl88, ZZ04, ZZZ05]. However, our treatment will differ in: (1) which functions we consider to be viruses, and (2) what we consider to be the set of programs infected by any given virus. We say more about each below.

1.1 Viruses as Computable Functions

In Adleman's model [Adl88], a virus is a computable function mapping programs to programs, subject to certain conditions. The programs in Adleman's model are considered to compute partial functions mapping environments to environments. If $v(p)$ is program p infected by virus v, then, when given an environment as input, $v(p)$ must either: (1) *injure* the environment, or (2) *infect* zero or more programs in the environment, and then *imitate* p by running p on the (possibly) modified environment.

[1] A set A is Σ_1 iff A is of the form $\{x : (\exists y)[P(x,y)]\}$ for some computable predicate P. A set A is Σ_1-*complete* iff A is Σ_1 *and* being able to decide membership in A would allow one to decide membership in *any* Σ_1 set [Rog67, Ch. 14]. For example, K, the diagonal halting program, is a well-known Σ_1-complete set [Rog67, Theorem 7-IV].

[2] We assume a virus detector to be a *total* decision procedure, as presumably any bug-free commercial virus detector would be.

Several variants of Adleman's model are proposed in [ZZ04, ZZZ05]. The model used primarily in both (called "nonresident virus") is, essentially, a refinement of Adleman's model, in that it requires, for infinitely many environments, $v(p)$ does nothing *but* imitate p. The purpose of this restriction is to eliminate certain functions considered *not* to be viruses [ZZ04, page 632].

Adleman's model and the variants in [ZZ04, ZZZ05] seem to be limited in the types of viruses that they can easily represent. For example, none would seem to readily model spyware (e.g, certain versions of the Sality virus [CA06]).[3]

Rather than limit our attention to some proper subset of the computable functions, we will instead allow *any* computable function to represent a virus. By taking such an approach, our results will be most general.

1.2 A Virus's Infected Programs

In [Adl88, ZZ04, ZZZ05], the set of programs infected by any given virus is considered to be the range of the function representing that virus. Under this interpretation, the set of programs infected by virus v is

$$I_v \overset{\text{def}}{=} \{q : (\exists p)[v(p) = q]\}, \tag{1}$$

where I is mnemonic for *infected*.

However, there is a problem with this approach. In order to avoid *re*-infecting already infected programs, many real world viruses incorporate some means of detecting the set (or, possibly, some superset) of the programs that they have already infected. For example, the following is taken from [Sop06] in regard to the Mac OS X Leap-A virus.

> The worm uses the text "oompa" as an infection marker in the resource forks of infected programs to prevent it from reinfecting the same files.

Some such viruses, however, can be *fooled*. That is, an *un*infected program can be made to *look like* an infected one so that it does not *become* infected. This process is sometimes called *immunization*. (See, for example, [Sec01].)

In mathematical language, the situation is the following. When virus v is presented with immunized program q, v does *not* alter q, i.e.,

$$v(q) = q. \tag{2}$$

So, is q *infected* by v? According to (1), we should say *yes*. But as long as q is *not* the result of infection by v of any *other* program p, clearly, the answer should be *no*.

Corresponding more closely to our intuitions, we take the set of programs infected by v to be the following.

$$I_v^{\neq} \overset{\text{def}}{=} \{q : (\exists p \neq q)[v(p) = q]\}. \tag{3}$$

[3] This is *by no means* a criticism of any of the models in [Adl88, ZZ04, ZZZ05]. Viruses propagate by exploiting system-level details that most models of computation try to avoid. As such, modeling a computer virus is difficult problem. For other recent work in this area, see [BKM05, BKM07].

By (3), the programs infected by v are those programs in the range of v that are *not* just fixed-points of v. Thus, if $q \in I_v^{\neq}$, then there is some program p that v *actively* changes into q, i.e.,

$$p \neq v(p) = q. \tag{4}$$

Henceforth, *the programs infected by virus* v refers to $I_v^{\neq}$, unless stated otherwise. In Section 3, we show that the set of programs infected by a virus in this sense can be Σ_1-complete (Theorem 1).

1.3 Co-isolated Sets

In computability theory, a set A is *co-isolated* $\overset{\text{def}}{\Leftrightarrow}$ every decidable superset of A is cofinite (i.e., has finite complement) [Rog67, page 107].[4] If A is itself cofinite, then A is trivially co-isolated. However, Post showed that there exist *recursively enumerable* (re)[5] co-isolated sets that are *not* cofinite [Pos44]. A set that is re, co-isolated, and *not* cofinite is called *simple* [Pos44] (see also [Rog67, Ch. 8]).

Clearly, if a virus v is such that any cautious virus detector labels all but finitely many programs as being infect by v, then $I_v^{\neq}$ is co-isolated. Does there exist a virus v such that $I_v^{\neq}$ is co-isolated?

PROPOSITION 1. For any re co-isolated set A, there exists a computable function v such $I_v^{\neq} = A$.

Proof. Let A be any re co-isolated set. Let f be a computable function such that the range of f is A. Let v be such that, for all p,

$$v(p) = f(\lfloor p/2 \rfloor). \tag{5}$$

Clearly,

$$(\forall q \in I_v)(\exists p \neq q)[v(p) = q]. \tag{6}$$

Thus, $I_v^{\neq} = I_v$, which is the range of f. $\Box$ (*Proposition 1*)

So, given that such computable functions exist, do any of them correspond to *real world* viruses? To answer this question, we ask another: if v is such that $I_v^{\neq}$ is co-isolated, what must *necessarily* be true about v?

The answer to the last question is surprising. As it turns out, if v is such that $I_v^{\neq}$ is co-isolated, then v has a property that we call *$\neq$-programmability* (Theorem 2, Section 3). In the next subsection, we introduce $\neq$-programmability and argue that it is *un*likely that a real world virus would have this property (Thesis 1, Section 1.4).

1.4 Programmability and $\neq$-Programmability

For all programs p and q, let $p \equiv q$ (p is *semantically equivalent* to q) $\overset{\text{def}}{\Leftrightarrow}$ p and q exhibit the same I/O behavior. (This notion is made precise in Section 2.) We say that

[4] An equivalent definition is: a set A is co-isolated iff there is *no* algorithm for listing infinitely many elements *not* in A.

[5] A set A is *recursively enumerable* iff there exists an algorithm for listing *all* and *only* the elements of A. Equivalently, as set A is re iff A is empty *or* A is the range of a computable function.

a virus v is *programmable* $\overset{\text{def}}{\Leftrightarrow}$ there exists a computable function f such that, for all p,

$$(v \circ f)(p) \equiv p. \tag{7}$$

Thus, a virus v is programmable iff there is an algorithm such that, given any program p, the algorithm finds $f(p)$ such that $(v \circ f)(p)$ exhibits the same I/O behavior as p.

We say that a virus v is *$\neq$-programmable* $\overset{\text{def}}{\Leftrightarrow}$ there exists a computable function f such that, for all p,

$$(v \circ f)(p) \equiv p \ \wedge \ (v \circ f)(p) \in I_v^{\neq}. \tag{8}$$

Thus, a virus v is $\neq$-programmable iff there is an algorithm witnessing the programmability of v with the additional property that: for all p, there is some program p' that v actively changes into $(v \circ f)(p)$.

For the sake of illustration, suppose the following.

- v is a $\neq$-programmable virus.
- f is as in (8) for v.
- p is an out-of-the-box copy of Microsoft Excel spreadsheet software.
- p' is a program that v actively changes into $(v \circ f)(p)$.

To summarize:

$$p' \neq v(p') = (v \circ f)(p) \equiv p. \tag{9}$$

Thus, since $p' \neq v(p')$, *v alters p' by infecting it*. Moreover, for the program that results, the I/O behavior *is exactly that of p*, i.e., *not* the I/O behavior of p with some malevolent behavior superimposed, but, rather, *exactly the I/O behavior of Microsoft Excel spreadsheet software*.[6]

Given the implausibility of such a v representing a real world virus, we theorize:

THESIS 1. It is *un*likely that a real world virus would be $\neq$-programmable.

It can be shown that there exists a virus v such that $I_v^{\neq}$ contains a program for every partial computable function, yet v fails to be $\neq$-programmable for lack of a *computable f* as in (8). Thus, Thesis 1 is a *strictly weaker* statement than: it is unlikely that there exists a real world virus v such that $I_v^{\neq}$ contains a program for every partial computable function.

It is tempting to extend Thesis 1 to: it is *un*likely that there exists a real world virus that is *programmable*. But such a statement would almost certainly be false. As mentioned in Section 1.2, for many real world viruses, a uninfected program can be immunized, i.e., made to *look like* an infected program so that it does not *become* infected. Moreover, this process is often algorithmic. That is, for such a virus v, there exists a computable function f such that, for all p,

$$(v \circ f)(p) = f(p) \equiv p. \tag{10}$$

(This is the case, for example, in [Sec01].) Clearly, then, such a v is programmable.

Thesis 1 has significant implications. Theorem 2 (Section 3) says that, for any virus v, if $I_v^{\neq}$ is co-isolated, then v is $\neq$-programmable. Thesis 1 and Theorem 2, together, say that:

THESIS 2. It is *un*likely that there would exist a real world virus whose set of infected programs is co-isolated.

Thesis 2 suggests that, for any real world virus, the set of infected programs will be *neither* simple *nor* cofinite. Thus, a cautious virus detector will (correctly) label infinitely many programs as *un*infected by the virus.

For all viruses v and v', let $v \equiv v'$ (v is *semantically equivalent* to v') $\overset{\text{def}}{\Leftrightarrow}$ v and v' induce the same I/O behavior in their infected programs, pointwise. (This notion is made precise in Section 2.)

Section 3 contains another rather surprising result. Theorem 3 says that a virus is $\neq$-programmable iff it is semantically equivalent to a virus whose set of infected programs is co-isolated.

1.5 Summary of Results

Our main results are summarized below. Several of our proofs employ infinitary self-reference arguments.

- Theorem 1 shows that, for any virus v, there exists $v' \equiv v$ such that both $I_{v'}$ and $I_{v'}^{\neq}$ are Σ_1-complete.
- Theorem 2 shows that, for any virus v, if $I_v^{\neq}$ is co-isolated, then v is $\neq$-programmable. Thesis 1 and Theorem 2, together, say that it is *un*likely that there would exist a real world virus whose set of infected programs is co-isolated (Thesis 2).
- Theorem 3 shows that a virus is $\neq$-programmable iff it is semantically equivalent to a virus whose set of infected programs is co-isolated.

The remainder of this paper is organized as follows. Section 2 just below covers notation and preliminaries. Section 3 presents our main results. Section 4 concludes and discusses future work.

2. Notation

Computability-theoretic concepts not explained below are treated in [Rog67].

$\mathbb{N}$ denotes the set of natural numbers, $\{0, 1, 2, ...\}$. A ranges over subsets of $\mathbb{N}$. Lowercase Roman letters *other than e, f, g, h and v*, with or without decorations, range over elements of $\mathbb{N}$. Lowercase letters e, f, g, h, and v, with or without decorations, range over total functions of

[6] Note that it may be the case that p' is semantically equivalent to p, and, thus, v makes a *syntactic* change to p', but not a *semantic* change. However, this observation does not seem to add significantly to the plausibility of v representing a real world virus.

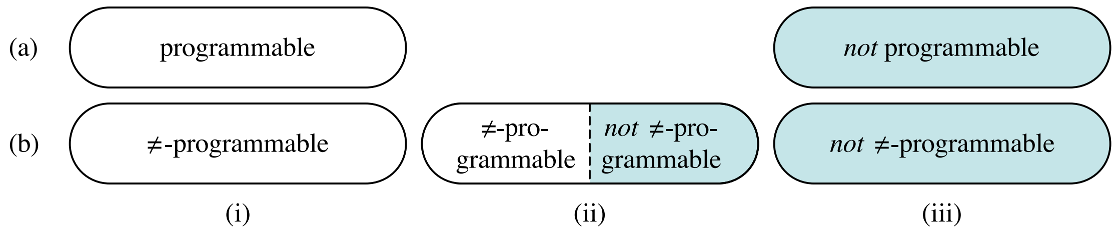

Figure 1. A graphical depiction of Proposition 2. In regard to programmability, a semantic class can be: (a)(i) entirely programmable, or (a)(iii) entirely *not* programmable, but *never* (a)(ii) mixed. In regard to $\neq$-programmability, a semantic class can be: (b)(i) entirely $\neq$-programmable, (b)(ii) mixed, or (b)(iii) entirely *not* $\neq$-programmable.

type $\mathbb{N} \to \mathbb{N}$. We will typically use v for a computable function representing a virus, and, e, f, g, and h, for arbitrary functions. id denotes the identity function of type $\mathbb{N} \to \mathbb{N}$. For all $f : \mathbb{N} \to \mathbb{N}$, $\mathrm{rng}(f)$ denotes the range of f, i.e., $\{y : (\exists x)[f(x) = y]\}$.

$\mathcal{PC}$ denotes the set of all partial computable functions mapping $\mathbb{N}$ to $\mathbb{N}$; $\mathcal{C}$ denotes the set of all (total) computable functions mapping $\mathbb{N}$ to $\mathbb{N}$. $\varphi_0, \varphi_1, ...$ denotes a fixed, *standard*, *algorithmic* numbering of $\mathcal{PC}$ [Rog58, Rog67]. One can think of φ as a programming language (e.g., C++, Java, Haskell), and of φ_p as the partial function (mapping $\mathbb{N}$ to $\mathbb{N}$) computed by the pth φ-program in some fixed indexing of all φ-programs. As a *standard* numbering of $\mathcal{PC}$, **composition** holds for φ [Rog67, Theorem 1-VI]. This asserts the existence of a computable function $\mathrm{comp} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that, for all p and q, $\varphi_{\mathrm{comp}(p,q)} = \varphi_p \circ \varphi_q$.

For all p and x, $\varphi_p(x)\downarrow$ ($\varphi_p(x)$ *converges*) $\overset{\text{def}}{\Leftrightarrow}$ there exists y such that $\varphi_p(x) = y$; $\varphi_p(x)\uparrow$ ($\varphi_p(x)$ *diverges*) $\overset{\text{def}}{\Leftrightarrow}$ there is *no* y such that $\varphi_p(x) = y$. In the latter case, one can imagine that the program p *goes into an infinite loop* on input x. $K \overset{\text{def}}{=} \{p : \varphi_p(p)\downarrow\}$, a well-known Σ_1-complete set [Rog67, Theorem 7-IV]. We use $\uparrow$ to denote the value of a divergent computation. $\lambda x\,.\uparrow$ denotes the everywhere divergent partial function.

For all p and q, $p \equiv q \overset{\text{def}}{\Leftrightarrow} \varphi_p = \varphi_q$. Thus, $p \equiv q$ whenever p and q exhibit the same I/O behavior. For all $f, g : \mathbb{N} \to \mathbb{N}$, $f \equiv g \overset{\text{def}}{\Leftrightarrow} (\forall x)[f(x) \equiv g(x)]$. Clearly, $\equiv$ is an equivalence relation with respect to $\mathcal{C}$. We call the equivalence classes so formed, *semantic classes*. $\mathcal{V}$ ranges over semantic classes.

Several of our proofs make use of the **Operator Recursion Theorem (ORT)** [Cas74], a result due to the first author. **ORT** represents a form of infinitary self-reference, similar to the way in which Kleene's Recursion Theorem [Rog67, page 214, problem 11-4] represents a form of individual self-reference. That is, **ORT** provides a means of forming an infinite computable sequence of programs $e(0), e(1), ...$ such that each program $e(i)$ *knows all* programs in the sequence *and* its own index i. The function e can also be assumed monotone increasing. The first author gives a thorough explanation of **ORT** in [Cas94].

ORT generalizes Kleene's Parametric Recursion Theorem (**PKRT**) [Rog67, Ch. 11]. **PKRT** provides a means of forming an infinite computable sequence of programs $e(0), e(1), ...$ such that each program $e(i)$ knows *its own program* and its own index i, but does *not* necessarily know the other programs in the sequence.

One of our proofs (Theorem 3) makes use of the following result due to Rogers.

COROLLARY 1 (of Rogers [Rog58, page 336]). For all programmable $f \in \mathcal{C}$, there exists 1-1, onto $g \in \mathcal{C}$ such that $f \circ g \equiv \mathrm{id}$.

The following is a recap of some of the key definitions introduced in Section 1. For all $v \in \mathcal{C}$, (a)-(d) below.

(a) $I_v \overset{\text{def}}{=} \{q : (\exists p)[v(p) = q]\}$.
(b) $I_v^{\neq} \overset{\text{def}}{=} \{q : (\exists p \neq q)[v(p) = q]\}$.
(c) v is *programmable* $\overset{\text{def}}{\Leftrightarrow} (\exists f \in \mathcal{C})[v \circ f \equiv \mathrm{id}]$.
(d) v is *$\neq$-programmable* $\overset{\text{def}}{\Leftrightarrow}$

$$(\exists f \in \mathcal{C})[v \circ f \equiv \mathrm{id} \ \wedge\ \mathrm{rng}(v \circ f) \subseteq I_v^{\neq}]. \tag{11}$$

Proposition 2 just below shows how programmability and $\neq$-programmability behave with respect to semantic classes. The proposition is depicted graphically in Figure 1.

PROPOSITION 2.

(a) (i) $(\exists \mathcal{V})(\forall v \in \mathcal{V})[v \text{ is programmable}]$.
(ii) $(\forall \mathcal{V})\big[(\exists v \in \mathcal{V})[v \text{ is programmable}] \Rightarrow (\forall v \in \mathcal{V})[v \text{ is programmable}]\big]$.
(iii) $(\exists \mathcal{V})(\forall v \in \mathcal{V})[v \text{ is } \textit{not} \text{ programmable}]$.
(b) (i) $(\exists \mathcal{V})(\forall v \in \mathcal{V})[v \text{ is } \neq\text{-programmable}]$.
(ii) $(\exists \mathcal{V})\big[(\exists v \in \mathcal{V})[v \text{ is } \neq\text{-programmable}] \wedge (\exists v \in \mathcal{V})[v \text{ is } \textit{not} \neq\text{-programmable}]\big]$.
(iii) $(\exists \mathcal{V})(\forall v \in \mathcal{V})[v \text{ is } \textit{not} \neq\text{-programmable}]$.

Proof. We show only (b)(i) here, as the others are straightforward. Let f be any computable permutation of $\mathbb{N}$ such that $f \neq \mathrm{id}$ and $f \circ f = \mathrm{id}$. Let $\mathcal{V}$ be such that, for all $v \in \mathcal{V}$,

$$\varphi_{v(p)} = f \circ \varphi_p. \tag{12}$$

By **composition**, $\mathcal{V}$ is *non*empty. Let $v \in \mathcal{V}$ be fixed. By **PKRT**, there exists $e \in \mathcal{C}$ such that, for all p,

$$\varphi_{e(p)} = \begin{cases} \text{id}, & \text{if } (v \circ e)(p) = e(p); \\ f \circ \varphi_p, & \text{otherwise.} \end{cases} \tag{13}$$

First, it will be shown that, for all p,

$$(v \circ e)(p) \neq e(p). \tag{14}$$

By way of contradiction, let p be such that $(v \circ e)(p) = e(p)$. Then,

$$\begin{array}{rcll} \varphi_{(v\circ e)(p)} & = & \varphi_{e(p)} & \{\text{assumed}\} \\ & = & \text{id} & \{(13)\} \\ & \neq & f \circ \text{id} & \{\text{choice of } f\} \\ & = & f \circ \varphi_{e(p)} & \{(13)\} \\ & = & \varphi_{(v\circ e)(p)} & \{(12)\} \end{array}$$

— a contradiction. Thus, (14) holds. Furthermore, for all p,

$$\begin{array}{rcll} \varphi_{(v\circ e)(p)} & = & f \circ \varphi_{e(p)} & \{(12)\} \\ & = & f \circ f \circ \varphi_p & \{(13) \text{ and } (14)\} \\ & = & \varphi_p & \{\text{choice of } f\}. \end{array}$$

Thus, by (14) and the immediately above equation, e witnesses the $\neq$-programmability of v. □ (*Proposition 2*)

3. Main Results

This section presents our main results. Theorem 1 below shows that, for any semantic class $\mathcal{V}$, there exists $v' \in \mathcal{V}$ such that both $I_{v'}$ and $I^{\neq}_{v'}$ are Σ_1-complete. Theorem 2 below shows that, for any $v \in \mathcal{C}$, if $I^{\neq}_v$ is co-isolated, then v is $\neq$-programmable. Theorem 3 below shows that a virus is $\neq$-programmable iff it is semantically equivalent to a virus whose set of infected programs is co-isolated. The proofs of Theorems 1-3 employ infinitary self-reference arguments.

THEOREM 1. $(\forall \mathcal{V})(\exists v' \in \mathcal{V})[I_{v'}$ and $I^{\neq}_{v'}$ are Σ_1-complete$]$.

Proof. Let $\mathcal{V}$ be fixed. To establish the theorem, it suffices to show that there exists $v' \in \mathcal{V}$ such that $(\forall p)[v'(p) \neq p]$ and $I_{v'}$ is Σ_1-complete. Let $v \in \mathcal{V}$ be fixed. Let f be a 1-1, computable function such that rng$(f) = K$. By **ORT**, there exists monotone increasing $e \in \mathcal{C}$ such that the behavior of $e(0), e(1), ...$ is determined by the following procedure, *where*, if i is such that $\varphi_{e(i)}$ is *never* set, then $\varphi_{e(i)} = \lambda x \centerdot \uparrow$.

> STAGE $s \geq 0$. Let p be *least* such that $(e \circ f)(s) \neq p$ *and* p was *not* chosen in any earlier stage. Then, set $\varphi_{(e\circ f)(s)} = \varphi_{v(p)}$.

Let v' be such that, for all p,

$$v'(p) = \begin{cases} (e \circ f)(s), & \text{where } s \text{ is the stage} \\ & \quad \text{of the above procedure} \\ & \quad \text{in which } p \text{ is chosen,} \\ & \quad \text{if such an } s \text{ exists;} \\ \uparrow, & \text{otherwise.} \end{cases} \tag{15}$$

Since e and f are 1-1, $e \circ f$ is 1-1. It follows that, for all p, $(e \circ f)(s) = p$ for *at most one* s. Thus, *every* p is chosen in *some* stage, and, so, v' is computable.

Clearly, $v' \equiv v$ and $(\forall p)[v'(p) \neq p]$. Furthermore, for all i, $e(i) \in I_{v'} \Leftrightarrow i \in \text{rng}(f) \Leftrightarrow i \in K$. Thus, e witnesses that $K \leq_1 I_{v'}$. □ (*Theorem 1*)

Clearly, for all $v \in \mathcal{C}$, I_v and $I^{\neq}_v$ are Σ_1 sets. Thus, Theorem 1 is, in a sense, as strong as possible.

THEOREM 2. For all $v \in \mathcal{C}$, if $I^{\neq}_v$ is co-isolated, then v is $\neq$-programmable.

Proof. Let v be such that $I^{\neq}_v$ is co-isolated. By **ORT**, there exists monotone increasing $e \in \mathcal{C}$ such that, for all j,

$$\varphi_{e(j)} = \begin{cases} \varphi_p, & \text{where } z \text{ is \textit{least} such that } P(z,j) \\ & \quad \text{and } p = |\{y < z : (\exists i)P(y,i)\}|, \\ & \quad \text{if such a } z \text{ exists;} \\ \lambda x \centerdot \uparrow, & \text{otherwise;} \end{cases} \tag{16}$$

where, for all z and j,

$$P(z,j) \Leftrightarrow [v(z) = e(j) \wedge e(j) \neq z]. \tag{17}$$

Note that since e is monotone increasing, P and $(\exists i)[P(\cdot, i)]$ are computable predicates. Let g be such that, for all p,

$$g(p) = \begin{cases} z, & \text{where } z \text{ is the \textit{unique} number} \\ & \quad \text{such that } (\exists j)[P(z,j)] \text{ and} \\ & \quad p = |\{y < z : (\exists i)[P(y,i)]\}|, \\ & \quad \text{if such a } z \text{ exists;} \\ \uparrow, & \text{otherwise;} \end{cases} \tag{18}$$

where P is as in (17). Since $I^{\neq}_v$ is co-isolated and rng(e) is infinite, $I^{\neq}_v \cap \text{rng}(e)$ must be infinite. It follows that, for infinitely many j, there exists z such that $P(z,j)$. Thus, g is computable. Clearly, $v \circ g \equiv \text{id}$. □ (*Theorem 2*)

The following lemma is used in the proof of Theorem 3.

LEMMA 1. For all $f \in \mathcal{C}$, if rng(f) is co-isolated, then f is programmable.

Proof. A straightforward modification of the proof of Theorem 2. We show only how to modify (16). By **ORT**, there exists monotone increasing $e \in \mathcal{C}$ such that, for all i,

$$\varphi_{e(i)} = \begin{cases} \varphi_p, & \text{where } z \text{ is \textit{least} such that } f(z) = e(i) \\ & \quad \text{and } p = |\{y < z : f(y) \in \text{rng}(e)\}|, \\ & \quad \text{if such a } z \text{ exists;} \\ \lambda x \centerdot \uparrow, & \text{otherwise.} \end{cases}$$

□ (*Lemma 1*)

THEOREM 3. For all $\mathcal{V}$, (a)-(g) below are *equivalent*.

(a) $(\forall v \in \mathcal{V})[v$ is programmable$]$.
(b) $(\exists v \in \mathcal{V})[v$ is programmable$]$.
(c) $(\forall$ re co-isolated $A)(\exists v \in \mathcal{V})[I_v = A]$.
(d) $(\exists$ re co-isolated $A)(\exists v \in \mathcal{V})[I_v = A]$.
(e) $(\exists v \in \mathcal{V})[v$ is $\neq$-programmable$]$.
(f) $(\forall$ re co-isolated $A)(\exists v \in \mathcal{V})[I^{\neq}_v = A]$.
(g) $(\exists$ re co-isolated $A)(\exists v \in \mathcal{V})[I^{\neq}_v = A]$.

Proof. By Proposition 2(a)(ii), (a) and (b) are equivalent. We show (b) $\Rightarrow$ (f) $\Rightarrow$ (g) $\Rightarrow$ (e) $\Rightarrow$ (b) below. It can similarly be shown that (b) $\Rightarrow$ (c) $\Rightarrow$ (d) $\Rightarrow$ (b).
(b) $\Rightarrow$ (f): Let $v \in \mathcal{V}$ be such that v is programmable. Let re co-isolated A be fixed. Let f be such that $\mathrm{rng}(f) = A$. Let f' be such that, for all x,

$$f'(x) = f(\lfloor x/2 \rfloor). \tag{19}$$

Clearly, $\mathrm{rng}(f') = \mathrm{rng}(f)$. By Lemma 1, f' is programmable. By Corollary 1, there exists 1-1, onto $g, h \in \mathcal{C}$ such that (20) and (21) below.

$$f' \circ g \quad \equiv \quad \mathrm{id}. \tag{20}$$

$$v \circ h \quad \equiv \quad \mathrm{id}. \tag{21}$$

Let v' be such that

$$v' = f' \circ g \circ h^{-1}. \tag{22}$$

Clearly, $v' \equiv v$ and $I_{v'} = \mathrm{rng}(f') = \mathrm{rng}(f) = A$. Furthermore, by (19),

$$(\forall q \in I_{v'})(\exists p \neq q)[v'(p) = q]. \tag{23}$$

Thus, $I_{v'}^{\neq} = I_{v'}$.
(f) $\Rightarrow$ (g): Immediate.
(g) $\Rightarrow$ (e): By Theorem 2.
(e) $\Rightarrow$ (b): Immediate. □ (*Theorem 3*)

N.B. Theorem 3 can*not* be extended to include

$$(\forall v \in \mathcal{V})[v \text{ is } \neq\text{-programmable}], \tag{24}$$

as this would contradict Proposition 2(b)(ii).

4. Conclusion

In this paper, we considered the question: can there exist a virus whose set of infected programs is co-isolated? We argued that there was a problem with the existing notion of *set of infected programs* (I_v), and presented an alternative notion that corresponds more closely to our intuitions ($I_v^{\neq}$). We showed that the set of programs infected by a virus (in the $I_v^{\neq}$ sense) can be Σ_1-complete (Theorem 1). We then showed that any virus whose set of infected programs is co-isolated is $\neq$-programmable (Theorem 2). We further argued that it is *un*likely that a real world virus would be $\neq$-programmable (Thesis 1). Thus, it is *un*likely that there would exist a real world virus whose set of infected programs is co-isolated (Thesis 2). Finally, we gave a surprising characterization of those semantic classes containing viruses whose set of infected programs is co-isolated (Theorem 3).

Our results suggest strongly that, for any real world virus, a cautious virus detector will (correctly) label infinitely many programs as *un*infected by the virus. A reasonable question to ask is: given a virus v and a program p *not* infected by v, is it possible to produce a program q such that $p \equiv q$ and q is *guaranteed not* to be incorrectly labeled as being infected by v? This question seems to be related to a remark of Adleman [Adl88, page 371]:

> Can some programs be given a 'clean bill of health'? For example, if it is known that a certain virus is about, would it be possible for a vendor to 'prove' that his program was not [semantically equivalent to a program infected by the virus]?

These issues certainly deserve further consideration.

References

[Adl88] Leonard M. Adleman. An abstract theory of computer viruses. In *Advances in Cryptology - CRYPTO '88*, volume 403 of *Lecture Notes in Computer Science*, pages 354–374, 1988.

[BKM05] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. On abstract computer virology from a recursion theoretic perspective. *Journal in Computer Virology*, 1(3-4):45–54, 2005.

[BKM07] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. A classification of viruses through recursion theorems. In *CiE'07 - Computation and Logic in the Real World*, 2007. To appear.

[CA06] CA. Virus Information Center: Win32/Sality Family, 2006. `http://www3.ca.com/securityadvisor/virusinfo/virus.aspx?ID=52797`.

[Cas74] J. Case. Periodicity in generations of automata. *Mathematical Systems Theory*, 8:15–32, 1974.

[Cas94] J. Case. Infinitary self-reference in learning theory. *Journal of Experimental and Theoretical Artificial Intelligence*, 6:3–16, 1994.

[Coh89] F. Cohen. Computational aspects of computer viruses. *Computers and Security*, 8(4):325–344, 1989.

[Pos44] E. Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50:284–316, 1944.

[Rog58] H. Rogers. Gödel numberings of partial recursive functions. *Journal of Symbolic Logic*, 23:331–341, 1958.

[Rog67] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw Hill, New York, 1967. Reprinted, MIT Press, 1987.

[Sec01] SecuriTeam. Remote Shell Trojan: Threat, Origin and Solution, 2001. `http://www.securiteam.com/unixfocus/5MP022K5GE.html`.

[Sop06] Sophos. First ever virus for Mac OS X discovered, 2006. `http://www.sophos.com/pressoffice/news/articles/2006/02/macosxleap.html`.

[ZZ04] Z. Zuo and M. Zhou. Some further theoretical results about computer viruses. *The Computer Journal*, 47(6):627–633, 2004.

[ZZZ05] Z. Zuo, Q. Zhu, and M. Zhou. On the time complexity of computer viruses. *IEEE Transactions on Information Theory*, 51(8):2962–2966, 2005.