A framework for modelling trojans and computer virus infection

Harold Thimbleby[1], Stuart Anderson[2] and Paul Cairns[1]

[1] *School of Computing Science, Middlesex University, Bounds Green Road,*
LONDON, N11 2NQ.
[2] *Department of Computer Science, Edinburgh University, King's Road,*
EDINBURGH, EH9 3JZ.
Email: harold@mdx.ac.uk, soa@lfcs.ed.ac.uk & p.cairns@mdx.ac.uk

It is not possible to view a computer operating in the real world, including the possibility of Trojan Horse programs and computer viruses, as simply a finite realisation of a Turing Machine. We consider the actions of Trojan Horses and viruses in real computer systems and suggest a minimal framework for an adequate formal understanding of the phenomena. Some conventional approaches, including biological metaphors, are shown to be inadequate; some suggestions are made towards constructing virally-resistant systems.

1. INTRODUCTION

Computer viruses are not merely an irritating and destructive feature of personal computing, they also mimic the behaviour of biological pests. Computer viruses hijack the facilities provided by the host computer, lie dormant, cross-infect other hosts, and when activated cause various forms of damage, from obvious destruction of host data to more subtle changes that are much harder to detect.

The predominantly human medical metaphors employed when discussing computer viruses are misleading because of ethical connotations. The word "virus" itself is a rich metaphor, and terms like "infection," "viral attack," "disinfectant," "viral damage" are frequently used of computer viruses. While these terms lack any precise meaning in relation to computers, we know *roughly* what they mean. Furthermore the debate about the computer virus threat is predominantly experiential: awareness of viruses is biased towards frequent infections caused by successful viruses; debate centres around newly identified virus cases.

In order to begin to tackle viruses effectively we should define the essential, general features exhibited by viruses without appealing to metaphor or limited experience to fill in the gaps in our understanding.

Computer viruses seem to be a simple phenomenon (hackers manage to write them in quantity, and there are virus construction kits that anyone can use to develop sophisticated viruses) yet they do pose a real hazard. The science fiction anticipation of viruses was thought to be obvious [7]. It is of interest, then, to explore the fundamental limits of detection and prevention and other issues. On the face of it, one would look to standard models of computation, such as Turing Machines, as a starting point for this investigation. However a closer analysis requires concepts not usually considered in the standard models of computation. We agree with Wegner [48] that Turing equivalent formalisms are not sufficiently expressive for systems that interact. A virus has to *enter* a system to infect it, and this is such an interaction. And as we shall argue, the biological metaphor is inadequate too (though it can still inspire computational ideas, both good and bad). The computational process undertaken by a virus does not terminate; moreover, when the (possibly) Turing process of an infected program terminates, the virus has already infected another program that will subsequently execute — a virus that terminated would not be infective.

Viruses should not be dismissed as a trivial problem for users of computers that do not run secure operating systems. Together with Trojan Horses, they are a major menace for any sort of system, and are a particular concern for computer systems connected to networks [31]. As well as being a practical problem, this paper will show that viruses pose theoretical problems also. Indeed, this paper, by laying out some of these problems, begs many questions that raise many further research questions. Some of these research questions will be pointed out explicitly throughout the paper.

In summary, this paper highlights the limitations of applying the conventional computational models to computer virus phenomena, and it offers a new framework. We motivate our paper by defining and naming the phenomena under discussion. We then examine a variety of related phenomena, such as: compiler viruses, Cohen's proof about virus detection, the possible structures of virally resistant systems, and, finally, Koch's Postulates and other biological issues. The reason why people write viruses is beyond the scope of the present paper; see [5] for an introduction and overview of the literature.

1.1. Orientation and terminology

Viruses and Trojan Horses make computers do things that their users do not want them to do. The term Trojan Horse is due to the Greeks' use of a hollow wooden horse filled with warriors to deceive the Trojans. The Trojans were violently torn between accepting the horse or rejecting it; indeed, they did start checking it, but were fatefully tricked into tearing down their own city walls and pushed it in themselves [46]. More recent discussions of viruses include [2, 11, 14, 17, 25, 36]. The interested reader is referred to [47] for up to date information on *specific* viruses, which are not the concern of the present paper.

For clarity and to avoid an air of pedantry we abbreviate "Trojan Horse" by the single word "trojan" when referring to computer programs. Trojan programs, sometimes with names like `sex`, often seem so attractive that users are sooner tempted to "bring them into their city," like the original Trojan Horse, than to test or to reject them — so bringing upon themselves the results intended by the trojans' designers. The complication is that trojans do something unwanted yet they offer to provide a wanted service.

One might start by considering four simple categories of trojan:

Direct masquerades pretend to be normal programs.

Example: a program called `dir` that does not list a directory, the normal use of the command of that name. Computer systems often permit many programs of the same name to co-exist, with consequent opportunities for masquerading trojans.

Simple masquerades do not masquerade as existing programs, but rather masquerade as possible programs that are other than they are.

Example: programs with names like `sex` above.

Slip masquerades have names approximating legitimate program names.

Example: a program called `dr` that might be activated if the user miss-typed `dir`. Since users want to install programs (e.g., for maintenance or upgrade purposes) and perhaps write their own programs, in practice few systems provide restrictions on the names of programs; even if they do, there must be fewer restrictions on the programmers who design them. The consequence of this liberality is undetectable/unidentifiable trojans.

Environmental masquerades are not easily identifiable programs invoked by the user, but are typically already-running programs that provide an unwanted interpretation on the user's commands or other activities.

Example: An operating system whose login prompt to the user is an otherwise clear screen and the prompt '`login:`' can be indirectly trojanised by constructing a program that intercepts the user, by clearing the screen and issuing the login prompt itself. It can then embark on any activity it pleases, typically recording the user's name and password; to avoid detection by the user it would then transfer control to the authentic login program. (If this is not feasible, it could misleadingly report a password error, ask the user to try again, and then terminate so that the authentic login program takes over completely.)

Example: When an entertainment CD is inserted, it may start playing automatically — if it contains executable code, it may be executed and do (or prepare to do) damage. The Macintosh `AutoStart 9805` is a trojan of this sort; it commences execution as a side-effect of inserting removable media.

Beyond these basic categories, it is fruitless providing precise definitions of the categories or types of unwanted behaviour of trojans (or, indeed, viruses) since their (i.e., their programmers') intention is to be devious. An actual trojan may exhibit mixtures of these behaviours and random effects to confuse and mislead the user. There is one trojan which, itself doing no direct damage, instructs the user to switch off the computer (pretending to have detected a serious hardware fault); if the user responds to this advice, they are liable to lose any data that has not been saved to disc. If we admit these programs that merely *say* things, what should we make of programs constructed by experimenters that say what they *would* do had they been actual viruses or trojans?

Given the wide range of behaviour, both intended and accidental, not surprisingly the terms trojan and virus are the subject of lively debate [21], based on a number of architectural and other distinctions. Most work in the area has concentrated on purely representational concerns, where every 'variant' of a virus technique (such as the Vienna virus represents) are considered different. (This has resulted in particular interest in 'polymorphic' viruses, which modify their representation on each infection.) Our approach is more semantical.

Whatever its behaviour, a viral infection has three core components:

A trojan component An infected program does something unwanted in certain circumstances. The trojan component is sometimes called the *payload.*

A dormancy component The viral infection may conceal itself indefinitely. Trojans, too, may use dormancy to conceal their presence, but with a virus dormancy (or, equivalently, unnoticed trojan damage) is essential for the effectiveness of their third component:

An infective component Infected programs infect further programs, which then behave in a similar way. (Viruses may wish to avoid re-infection, because re-infection takes time or space and may therefore make an infection more obvious. Viruses often include a heuristic for self-detection, a procedure which, if identified, might be directed against them.)

Even this abstract list has problems! A virus that only replicates consumes computer resources (it uses disc space, it wastes time. . .) even though it may not have an explicit trojan component. Or a virus that is never dormant might still be able to infect, provided it uses some other means of avoiding detection.

If we allow programs to run on various machines, across platforms, why stop at digital computers? Are viruses with no infective component, but which trick humans into spreading them, rather than spreading themselves, really viruses? Are chain-mail, spam [38] and email panic scares, which are replicated as people email the panic to people they wish to warn, really viruses [18, 35]? Certainly, these are serious problems. As communications systems become more sophisticated, performing actions (such as forwarding email) on behalf of users, the traditional distinctions between trojans and viruses become harder to maintain.

In particular the trojan component may be 'harmless' (viewed inside a computer) but precipitate a chain of behaviour *outside* any computer, ending with a harmful conclusion. Although the trojan component may run at a definite time, the user may in principle be unaware of the consequences until much later. A particularly pernicious program, `ADTrojan`, sends rude email to other users. The only action a computer could detect would be the (not necessarily) unauthorised mailing. The recipients of the email, however, can complain, leading to repercussions for the original user. Other sorts of delayed trojan activity include programs that steal passwords: the noticeable damage occurs much later when the passwords are used to enable third parties to do other work — which of course may be part of a larger chain of activities, only the end of which is blatant damage. The legal consequences of this sort of indirect behaviour involving innocent parties have not yet been worked through satisfactorily.

We might also add that a virus has a **survival component**. Crucial to the long-term practical success of a virus is that it can infect in some sense faster than it can be eliminated; to do so, it may infect other computers — these other computers are often the responsibility of other people or organisations, and so the elimination of a virus requires human co-operation, which is typically slower to recruit than the spread of the virus. Clearly, virus survival is of practical significance, and the organisational structures anti-virus manufacturers use to distribute their products is of commercial interest. However, survival is an emergent property, that follows from dormancy and infection, so we do not formalise it here. We consider it an epidemiological issue (see, e.g., [23]).

Because we use words like virus and trojan, and often give them names (e.g., *the* Vienna Virus), it is easy to think the problem is just one of identification.[2] However, bugs, viruses and worms can, under appropriate assumptions, be benign. Bugs are used constructively in certain forms of AI programming; viruses can be used to spread information [49]; worms can be used to distribute processing [12]. Moreover, the people who write destructive programs may think their programs are benign, even if others disagree. Clearly identifying 'the' problem is a human issue, related to what one wishes to define as a problem. From the formal point of view, the question is, if one wished to classify something as a problem, whether that thing could be distinguished to be so identified. Of two programs, before one can say one is a trojan, it has to be established that they are different.

Thus to develop our framework, we wish to be able to model the essential issues, but perhaps not make all of the distinctions some authors have made. An adequate framework must involve notions of behaviour, invisibility, infection — what is the 'same' virus? These questions have practical implications: How should viruses be named? Are all strains of the Vienna virus the same virus, or are they different?[3]

A framework must address, or relate to, the confusion that can exist over the interpretation of 'damage.' If a user, being malicious, *intends* damage, are we then to consider the activities wrought by a trojan on his or her behalf as 'constructive'? The notion of damage clearly assumes particular value systems and intentions on the behalf of particular classes of user. Indeed, the behaviour of a user developing a compiler system, who in the course of their legitimate work, compiles code, replaces system resources, and recompiles various programs and so forth, is hard to distinguish on purely technical grounds from a user who replaces programs with malicious intent. (A tongue-in-cheek article argues that research ideas behave like viruses [37]: which explains the proliferation of research publications and journals.)

Although both trojans and viruses may be difficult to isolate after damage has been detected, the additional components of viruses ensure they are not so easily eliminated. As a result of dormancy and infection, a virus normally makes many copies of itself (not necessarily identical) before the damage is detected. The virus may make copies of itself essentially outside the computer it

[2] This is what most virus detection programs do: they look for viruses whose signature they recognise. If a user (or a program they are running) knows what virus to look for, there is a partial oracle for (what we will show to be) the non-computable function that identifies the virus. See [45].

[3] The Vienna virus has many variants, partly due to the publication of source code for one variation of it in [8].

is running on: it may make copies of itself on removable media (such as a floppy disc) so that the user, on putting the disc in another computer, infects a remote computer; or it may use networks to do the infection more directly itself. When — if — the virus is detected it will already have spread somewhere else, where its management is someone else's responsibility.[4]

We assert that a virus can only be defined semantically, that is in terms of the meaning of programs, rather than in syntactic patterns appearing in code. This belief is strongly at variance with most of the literature on the topic [9, 15]. All viruses to date have been trivial, and this gives the impression that viruses might be classified as syntactical, with various identifiable structures — for example, fragments of code that correspond with the three conceptual components listed above. With the prevalence of so-called polymorphic viruses (ones that have different syntactical forms [32]), this view is of course questionable, but there is a lingering (and incorrect) view, perhaps partly inspired by the relatively concrete notion of biological chromosomes or genes, that there 'really' is a syntactic basis. If any biological metaphor is required, computer viruses are more like Dawkins's memes, "viruses of the mind," than biological genes [13], a point we return to later (§5). In our view, what is important about a virus is not how it works, but what it accomplishes.

A virus is best defined by saying what it is to be infected. An infected version of a program p is a program p' that behaves indistinguishably from p on 'most' inputs, behaves distinguishably from p on some inputs, and sometimes when p' is executed it infects other programs in the 'same' way. This preliminary definition begs a lot of questions and uses terms without definition. The remainder of this paper, then, is directed to making these notions precise. However, one thing is clear from this definition, the notion of viral infection is not linked to any particular syntactic representation as code.

Our paper will necessarily introduce a new definition of virus, which we will provide after justifying a new framework within which to express the definition. In particular, so-called worms, which some authors distinguish from viruses as autonomous programs rather than programs physically carried by users, are not distinguished in our framework.

We will find it convenient to use the general term *infection* to indicate the presence of a trojan or a virus: an infection is, from the hacker's point of view, an intentional bug. In keeping with our abstract approach (and indeed the medical usage) this does not imply a particular manner of acquiring the infection, nor that the infection is transmissible to other programs or computer systems.

1.2. Previous work

We are aware of previous work on the theoretical limits to the detectability of trojans (e.g., [1, 9, 10]). Our own first undetectability proof for masquerades [42], based on this work, was questioned for not explicitly addressing security issues [15] (a criticism that also applies to the other methods). We have also attempted to identify and distinguish the causes of co-operative behaviour and destructive behaviour [43], but this was informal; and we have found problems in a proposed detection mechanism [26], but our findings did not constitute a general theory.

2. INADEQUACY OF TURING MACHINE MODELS

It might seem obvious that personal computers are Turing Machine equivalent, and as personal computers get viruses, so Turing Machine models are an appropriate basis for studying issues of infection. However, this view is false.

First, we will make a plausible argument, then we shall provide a proof. Turing Machines are infinite machines, whereas personal computers are finite. Clearly, Turing Machines are an idealisation. Personal computers have properties not shared with Turing Machines. Some real-world properties of personal computers — plausibly including the issues of trojans and viruses — are not modelled by the idealisation.

Now, more formally, suppose we define 'infected' as a predicate on Turing Machine programs, and we do not try to formalise what can be observed as a result of the execution of the program. What can this mean? It can only mean that the result of running the program is either unchanged (the infection has no observable effect) or that the result is incorrect (the infection is not 'hidden'). In neither case do we have a non-trivial problem.

All previous work is based on abstract models of computation together with a concrete notions of replication. Such approaches cannot capture various subtle notions of, for example,

- **masquerading**, where a user knows names of programs and anticipates their likely behaviour;
- **infection**, where viruses may encrypt themselves and be different each time they infect.

Our comments will apply equally to any Turing equivalent model of computation, such as λ-calculus. Note that it is important to distinguish between a representation of viruses (clearly, any computable theory of viruses is directly representable on Turing Machines), and a definition of an effective model.

There is a tension that, on the one hand, the conventional undetectability results are too pessimistic; on the other hand, naïve notions of infection may lead to

[4] It follows that successful elimination of viruses requires a distribution mechanism for the antidotes: this is usually done by conventional marketing channels (that do not use the same vectors as the virus), rather than by virus-like computer-based replication.

excessive optimism — in that specific remedies may be attempted that rely on these notions of infection. Questions remain whether virus activity, replication and infection, can be usefully constrained in suitable computational models where such activity is explicitly but generally modelled. We will explore some of these issues below, in §4.

2.1. 'Other' programs

The notion of a virus infecting other programs has to be modelled: such a notion is essential to any discussion of viruses. But 'other' programs cannot be modelled by a conventional Turing Machine model since there is no other program — even if the Turing Machine tape contains several interleaved programs whose execution is multiplexed.

2.2. Self-awareness of infection

In principle, a program could be infected by a virus in such a way that it could not tell it was infected (the malicious code could interfere with any infection testing): thus, the reliable judgement whether a program is infected depends on an external mechanism that is not affected by that infection. In fact so-called armoured viruses [3] exist for the Intel 80486 processor, which detect attempts to single-step (and hence understand their behaviour) them by rewriting already-pipelined instructions so that they can distinguish whether they are being run directly by hardware or being interpreted.

We will avoid this problem by introducing the concept of trojan and viral methods, which are *outside the system* — they are outside the standard Turing model — whose infection is being considered.

2.3. Self-replication as a special case

If infected programs were empty, computer viruses would involve self-replication. Self-replication of programs has attracted a great deal of attention, particularly because of the apparently paradoxical problem of encoding concrete representations of programs *inside* themselves: that is, when a self-replicating program is run, its output is its own source code. From this point of view of self-replication, a program that merely accesses its representation (e.g., from a source code file) would be cheating!

Turing completeness (sufficiency for implementing a universal Turing Machine) is neither necessary nor sufficient for self-replication.

It is not necessary since self-replicating programs can be constructed as non-recursive straight-line programs. A programming language could therefore be constructed that was not Turing complete, but which was sufficient to implement non-cheating self-replicating programs.

It is not sufficient, since Turing completeness is up to representation. We can construct a programming language that is Turing complete, but where the language uses symbols that running programs cannot output. We then have Turing completeness where self-replication is impossible. A proof of insufficiency where the domain and range of a Turing complete language are identical is given in [39].

Fokkinga [16] gives a construction for self-replicating programs, and adds a variation: a program that recognises its own source code — which is a pre-requisite for a virus not to repeatedly re-infect a file (see §2.2). Kanada gives an example of a self-replicating program that runs on the World Wide Web [24].

2.4. Time

Turing machines are infinite machines, and their speed of operation is immaterial. The real world of practical computers is finite; space is finite and time is significant. Since, for small n, every program on an n word (RAM + backing store) computer can be enumerated and classified as acceptable or infected by inspection, it follows that an adequate framework should allow for the complexity of classification.

Also, some viruses do their damage 'merely' by consuming resources. This reduces the effective speed of the computer, or looses free space. Neither of these effects are of any consequence in a computational model that admits an infinitely fast, infinite memory machine.

3. A NEW FRAMEWORK

In any rigorous framework for the study of viruses there must be a mechanism to create and distinguish between various programs. Without loss of generality, the necessary extension is an environment, a mapping of names to programs, equipped with the appropriate operators to make enquiries of it and to manipulate it.

The object of our study are computer programs together with their inputs and outputs.

We could imagine a real computer to be an array of bits, including its RAM, screens, backing store, and the state of its CPU. Parts of this array correspond to programs, part to data files, and various parts to such components as boot sequences and directory structures. The meaning of certain sequences of bits may depend on where they reside in the computer. A bit pattern may be a graphic image, but somewhere else it might be a program that can be run. After an error, such as stack overflow and obtaining a bad return address, the computer might be directed to interpret a graphic as program.

We call all these things together (the full state of a machine, program texts, graphic images and so forth) the *representation*. The collection of all possible representations we denote R. Any given representation, $r \in R$, is finite. Note that R includes all fault conditions, such as the computer 'locking up.'

The user is not concerned with representations. Indeed much of the memory of a computer is hidden from the user, in boot sectors and other places. The user is concerned with names of programs, and the computer uses those names, following various rules, to obtain the representations of the corresponding programs. We therefore introduce the *environment map*, E, which is a fixed map from a representation (typically the current configuration of the computer) to an environment, a name-to-representation map, which takes names (chosen from a countable set of labels L) and, if they are defined, obtains their corresponding programs or other data: $E: R \to (L \twoheadrightarrow R)$. The domain of $E(r)$, $\mathsf{names}\, r = \operatorname{dom} E(r)$, is finite and computable and (as is made clear below) will normally include some fixed names independent of r.

Note that 'names' are a very general concept, and include, for instance, locations on a graphical user interface screen, or textual names in a conventional file store. In practice, the environment will have structure that may have security implications, but this is not required for our framework.

Programs may be run, and running programs usually changes the state of the computer. We say that the *meaning* of a program is what it does when it is run. If $r \in R$ is a representation that includes a program p, then $[\![p]\!]$ is its meaning: $[\![\cdot]\!]: R \to (R \to R)$. The meaning of a program is that, when run, it transforms representations into representations. Note that our approach admits everyday complexities such as operating systems, virtual machines, spreadsheet macro programs, dynamically loaded Java applets, and so forth — but it is not necessary to model communication, nondeterminism or concurrency to capture what viruses do. A more thorough definition of programs and representations could certainly be developed (and would be a useful research project), but for our purposes we do not think it would provide any additional clarity to do so — as more structure is introduced, it is very hard to avoid implementation bias, and the associated obscurities of 'real' machines.

Where no ambiguity arises, we abbreviate the structure $\Omega = [R; E; [\![\cdot]\!]]$ by R.

Our framework does not require a virus to reside in 'a' program; conceivably it could reside in two or more cooperating fragments, none of which alone behave like a virus. A trivial way to do this is by threading the virus code around the representation, but more subtle techniques are possible: see §4.4.

Crucial to a framework for examining infection is that programs appear, perhaps for long periods of time, to be other than what they really are. A user may find two programs to be indistinguishable even though they are not equal. We define these terms precisely as follows:

Two programs p and p' are *equal* when

$$\forall r \in R: [\![p]\!] r = [\![p']\!] r$$

However, unlike identity, equality is not a computable relation,[5] and even to check equality of program outputs for a small collection of inputs would require the exhaustive examination of the entire state of the computer. More practically, in an attempt to take account of the amount of time one is prepared to devote to checking an output is what is required, we define *similarity* to be a poly log computable relation on R (see below), denoted $\sim$.

We do not assume similarity is an equivalence relation. In particular, similarity is not transitive: we may know $a \sim b$ and $b \sim c$ but also that $a \not\sim c$, given the poly log time restriction. Since $\sim$ is computable it must be that either $a \sim c$ or that $a \not\sim c$, and that this result is known in finite time: the computation of similarity may make 'mistakes.' There then arises the possibility that unequal programs are similar: although two programs are different, we may not be able to decide that in the time available. The key point is that similarity ($\sim$) of programs is not equality ($=$) of programs, for if it was there would be no serious problem in detecting different programs.

We define *poly log*, and a related quantifier, *for most*:

Poly log computable Poly log computable is a restriction that a function can be computed in less than linear time on the total size of its arguments. Poly log is a requirement that a function of representations can be evaluated without examining the entire computer representation (which can be done in linear time). If the entire computer representation could be examined at every step in a process a number of detection questions become trivial; furthermore it would implausibly suggest the user is aware of the entire configuration of the computer, including boot sectors, operating systems and so forth.[6]

$\mathcal{M}$ **(for most)** We need to introduce a new quantifier, *for most*, written $\mathcal{M}$. A technical definition of this notion is not required in what follows; a definition that captures the intuition and the relation with poly log computable (or some other measure) is a research project.

Two programs are indistinguishable when they produce similar results for most inputs. If p and p' are two program representations they are *indistinguishable*, written $p \approx p'$, if and only if

$$\mathcal{M}\, r \in R: [\![p]\!] r \sim [\![p']\!] r$$

We need some convenient notation:

- $\widehat{p}$: A representation or program $\widehat{p}$ is an attempt to trojan p; $\widehat{p}$ is to be taken as a metaname, and may, in fact, have no relation to p.

[5] The program fragments $1+1$ and 2, suitably interpreted, are equal but not identical.

[6] For small computers, say handheld calculators, the poly log restriction may make it feasible to examine the entire representation space.

- $r \xrightarrow{l} r'$: We write $r \xrightarrow{l} r'$, iff $l \in \mathsf{names}\, r$ and $r' = [\![E(r)l]\!]r$; this extends naturally to finite sequences of program names $l_1 l_2 \ldots l_n \in L^*$.
- s/c: Let s/c be the object code corresponding to s when compiled by a compiler c. This seems intuitive enough, but it assumes that out of the entire machine representation it is possible both to choose the source code and the object code resulting from compiling the source. In fact, just before a compiler (or any other program is run) some other program (e.g., the operating system) places the parameter of the compiler in a known part of the representation; some convention in the program (no doubt enforced when it was compiled!) then specifies the location in the representation of its parameter. If we label these locations $\Lambda_1, \Lambda_2 \ldots$ then we have (allow s/c to be empty in the case that s is not well formed with respect to c):

$$\begin{aligned} &\exists \Lambda_1, \Lambda_2 \in L \colon \forall r \in R\colon\\ &\quad s/c = E[\![c]\!]r\Lambda_2\\ &\quad \textbf{where } s = E(r)\Lambda_1 \end{aligned}$$

  If c_s is a compiler c in source form, $c = c_s/c$. The notation extends naturally to finite sequences of applications of a compiler: $s_n/s_{n-1}/ \ldots\ s_0/c$.

With these preliminaries, we may now define trojan and virus. In attempting to do this we find that we cannot eliminate the environment from the definition. The notion of trojan and virus can only be understood relative to their binding environment. Hence, rather than define trojan and virus as such, we will define a recursively enumerable relation to capture the *method* ('infection') employed by the trojan or virus respectively.

3.1. Trojans

Trojans may corrupt something unnamed (say, a boot sector) which when run at a later time results in an 'obvious' trojan effect — but even that 'obvious' trojan effect cannot usually be determined except by running a program, for example to check that certain files are still present.

As a first approximation, we might say that programs $p, \widehat{p}$ would stand in the relation trojan when there is some representation r that distinguishes their behaviour; informally, $p \;\mathsf{trojan}\; \widehat{p} \Leftrightarrow \exists r \in R\colon [\![p]\!]r \not\sim [\![\widehat{p}]\!]r$.

Notice that the trojan relation is symmetric: without assuming what p or $\widehat{p}$ is supposed to do, we can't know which program is intended as a trojan of which. We could imagine a hacker making the ironic comment that a real login program trojanised their subversive login program. Since it is not obvious that one can sufficiently easily make a formal distinction between what some humans mean and others mean, we will leave the relation as symmetric — and see how far we get!

It is crucial that the trojan programs exist 'as such' in the particular computer as programs that the user can run: they must have names in an environment. We therefore introduce the concept of a trojan method that characterises the appropriate constraints. For each type of trojan there will be a different trojan method; but by using the abstraction of a method, we do not consider different representations (i.e., different implementations) of trojans as essentially different. Each trojan method specifies a particular program and a computer configuration supporting an environment in which it can be trojaned. (This pedantry — which is implicit because Ω is 'hidden' in the use of R, E and $[\![\cdot]\!]$ — is not only a useful clarification, but reminds one that a trojan of a Unix program `sh`, say, won't necessarily be a trojan for a different user with a different name space.)

Definition A *trojan method* is a non-empty recursively enumerable relation $T \subseteq R \times R \times L$, such that if $\langle r, \widehat{r}, l\rangle \in T$ then:[7]

$$\begin{aligned} &\wedge\, r \sim \widehat{r}\\ &\wedge\, E(r)l \approx E(\widehat{r})l\\ &\wedge\, \mathcal{M}\, t \in L^*\colon\\ &\qquad \wedge\, [\![E(r)l]\!]r \xrightarrow{t} r'\\ &\qquad \wedge\, [\![E(\widehat{r})l]\!]\widehat{r} \xrightarrow{t} \widehat{r}'\\ &\qquad \wedge\, r' \not\sim \widehat{r}' \end{aligned}$$

The idea of this is that if $\langle r, \widehat{r}, l\rangle \in T$ for some trojan method T, then $\widehat{r}$ has an environment which is similar to r, but in which the program named l, although looking the same in the two environments if it is executed for most potential inputs, eventually a difference can emerge.

The second line of this definition (i.e., that a trojan does not immediately reveal itself) is optional. The formalism helps make explicit the choices available in the definition of the terms. We feel, however, that it is appropriate, for it is saying a trojan is *initially* indistinguishable to another program but eventually obviously different.

A central contribution of our definition is the notion of a trojan as a relation; however details in the definition could easily be debated. For example we could replace the uncertainty of $\mathcal{M}$ by requiring that $\forall s \in L^*$ (i.e., using a *for all* quantifier, rather than the *for most* quantifier) there is an extension t of s with similar properties; the uncertainty has then been pushed into the uncertainty of the length of t. Since trojans generally intend to appear at some point in the future, the majority if not all of them would satisfy this variant definition.

Detection of trojans is built into this definition. A trojan is defined in terms of not being distinguishable from the original (using $\sim$). If a trojan was detectable because it was different it would not be a trojan — it would just be a "wrong program."

[7] We use Lamport's method of writing long formulas [27].

3.2. Viruses

There are, of course, good viruses and other variations, but we define a virus to be a trojan that additionally infects other named programs, infection being the modification or creation of some program to be a virus. In our framework, then, we do not distinguish a program that *is* a virus and a program that *contains* a virus: to do so would presume an identification of the virus 'code.' (Of course most virus writers write such simple viruses that the distinction has practical use even if no general semantic basis.)[8]

Two representations $r, \widehat{r}$ are virally related on name l if they are part of a trojan method, and if the capacity to trojanise and infect is transmitted to other programs. Thus a viral method is a trojan method with added conditions requiring that the method is infectious.

Definition A *viral method* is a trojan method $V \subseteq R \times R \times L$ satisfying the additional condition, such that if $\langle r, \widehat{r}, l\rangle \in V$ then:

$$\begin{aligned} &\wedge \mathcal{M}\, r_1, r_2 \in R\colon r_1 \sim r_2 \\ &\wedge \exists l' \in (\mathsf{names}\, r_1 \cap \mathsf{names}\, r_2)\colon \\ &\qquad \langle [\![E(r)l]\!] r_1, [\![E(\widehat{r})l]\!] r_2, l'\rangle \in V \end{aligned}$$

This additional clause is a closure property, saying that evolving two similar representations by virally related programs results in virally related representations. Given a viral method V and a 'normal' representation r, then $\widehat{r}$ is infected by V at l if $\langle r, \widehat{r}, l\rangle \in V$.

It is useful to distinguish an infected system from an infected program, since the cost of establishing whether a system is infection-free is much higher than to establish whether a program is infected.

The definitions do not require a virus to infect with a copy of itself, and in particular they allow a virus to encrypt itself in different ways when it infects. Thus we do not require infection to be transitive, since the definition of a virus does not require it to infect with itself (a typical encrypting virus would choose to infect with a differently encrypted variant of itself).

There is nothing in the above definition which requires some syntactic relation to hold between the 'normal' and 'infected' program. This is appropriate, since one could easily imagine a virus incorporating a simple, semantics-preserving re-write system that could be used to transform the combination of the viral code and the new host into some equivalent but syntactically quite different form.

3.3. Summary

An important difference between virus and trojan is now clear: a virus requires to modify the name space of the representation, thus suitable precautions on naming could inhibit viral spread (under suitable assumptions), whereas a trojan in some sense makes the user do its work, and therefore cannot be identified or inhibited if the user is anyway permitted to perform such operations. Trojan compilers form an interesting case where a user may be tricked into performing an evaluation step that can then behave as a virus (§4.1.1).

The definitions clearly do not require Ω to be Turing Complete in order to support trojan or viral methods. It would be possible for a machine to support programs of viral methods only. Such an apparently limited machine might be of interest for investigations in artificial life [28].

In considering any formal framework of artificial systems, there is always a balance between formalising what *is* and formalising what *should be*. Our framework does not handle certain concrete aspects of viruses explicitly: is this a shortcoming in our framework, or is it an indication that the complex systems that support them should not exist? We think, while it would be a significant achievement to handle more virus behaviour within an elegant framework, it would be a greater achievement to eliminate the possibility of certain sorts of behaviour by better system design.

4. APPLICATIONS OF THE FRAMEWORK

In this framework, where the notion of trojan and virus is inextricably bound up with the environment and the realistic time complexities of testing, the questions one can ask about these phenomena differ from the usual questions. We might consider the following:

- Given a representation r, and a viral method V, it is semi-decidable to check whether some other representation $\widehat{r}$ is virally related to r in V.
- Given some finite number of infected/non-infected pairs of environments in some unknown viral method V it is impossible to 'infer' V from the data.
- The question, assuming we have to hand a putative virus, "is p a virus?" makes no sense. For many reasonable notions of $\sim$, even deciding $p \approx \widehat{p}$ is undecidable. For very restricted notions of infection (e.g., syntactic modification) limited decidability results are obtainable.
- Is it possible, by elaborating the model of the computing system, to provide a system which resists, detects, or is tolerant to viral spread? The affirmative answer changes our attitude to third-party anti-virus products, and suggests a requirement, antitrust notwithstanding, that anti-virus components be integrated into operating systems.
- Following from the previous point: if a (particular) viral method can be recognised, can the representation including it be disinfected, where we take 'disinfected' to mean some more conservative operation than deletion of all programs overlapping the virus?

[8] Some authors would distinguish a *virus* that only modifies existing programs from a *worm* that can also create programs, typically on another node in a distributed system.

- Many programs are constructed out of modules. Our framework does not address this since any collection of modules is just part of the representation. However, in practical terms, there is a difference in convenience or efficiency if we can reliably check modules individually. Most anti-virus products do just this: they normally only check the parts of the representation that are known to change through interaction with the rest of the world — such as when a floppy disc is inserted. The problem does not arise in our framework, but any framework that did model modules (or access rights) would have to beware of the problem that trojan methods need not be in the modules where they "should" be — see §4.1.1.
- Because anti-virus products are commercial, there are industry tests and league tables. League tables encourage simplistic comparisons, such as 'percentage of wild viruses recognised.' However, hit rates assume a classification of viruses, typically a syntactic one — which arguably inflates the apparent scale of the problem, and the efficacy of the anti-virus programs. How should anti-virus products be compared?

4.1. Detectability of trojans

The problem of detecting trojans is at least as hard as determining whether functions are equal, which is undecidable.

There is, of course, a difference between detecting a trojan and resisting the installation of a trojan: security measures are aimed at the latter problem. However, as regards security assumptions precluding an arbitrary program $\widehat{p}$ from, in some sense, being related to a program p, by assumption the program $\widehat{p}$ is explicitly constructed to trojanise p.

Trojans are not effectively detectable. In fact most trojan and virus detection programs attempt to detect classes of program: the importance of the following result is that many detectors search program representations for patterns (equivalent in biological terms to antigenic epitopes) and attempt to detect any program in the class with that characteristic pattern.

Assuming the classification process is computable and that detection is undecidable, the decidability of detecting classes of trojan would be a contradiction; if the classification process is not computable, then there are trojans that cannot be classified, and hence cannot be detected. This has implications for trojan detector programs that attempt to identify specific trojans by their known structure, such as by using a signature.

In many computer environments it is also possible for trojans to dynamically change their code ('mutate'): this would mean that a recently acquired trojan could have a different form than the standard recognised by the trojan detector. By considering the equivalence classes of the behaviours of trojans, we immediately conclude that trojans are not detectable by inspecting their behaviour: this result is of consequence for so-called gatekeeper detectors that hope to detect trojans or viruses by their (presumably forestalled) actions. They cannot work in general. In practice a useful but insufficient measure of protection is achieved by interpreting primitive operations (such as operating system calls) and intercepting certain computable classes of operation (such as low level formats); there may be options to enable an operation to proceed if the user deems it to be safe. Inevitably, such methods presuppose a human is able to make decisions that we have proven undecidable. Inevitably, human mistakes will be made.

Recall that Cassandra, the Trojan prophetess, though correctly warning what the Trojan Horse was, was doomed not to be believed!

4.1.1. Thompson's trojan

The intention of Thompson's construction is to use the trapdoor effect of compiling to conceal a trojan from effective detection: $r/c \leftrightarrow r$ is not bijective, and r cannot be recovered from r/c (unless c is specially constructed to make this possible). In fact, it may be much worse, there may be no s derivable from $\{c, r/c\}$ such that $r/c = s/c$. This is the well-known 'disappearance' of semantics in metainterpreters (virtual machines) [20]; in Thompson's trojan the semantics that disappear are trojan methods.

Normal compiler bootstrapping is expressed as $c_s/c = c$, where the subscript $_s$ conveniently denotes the appropriate source code. Bootstrapping is typically achieved by constructing, by hand or by some other means, an executable program p such that $c_s/p = c_s/ \ldots c_s/p$ (it is not necessary that $c_s/p = p$); once p has been successfully applied to c_s, p can be discarded — although this is risky, as is made clear below. The source code c_s, too, may be discarded or made unavailable (perhaps for commercial reasons). Yet it is still possible to compile all programs. The language c compiles will be complete in some sense (the properties described are not sufficient for Turing completeness).

To follow Thompson's argument it is necessary to introduce conditionals, notated $x \Rightarrow y : z$; we assume that the language processed by c can implement the intended semantics, $x \Rightarrow y : z \overset{\text{def}}{=} \textbf{if } x \textbf{ then } y \textbf{ else } z$. It will be sufficient to consider only top-level conditionals and (computable) tests based on identity.

Thompson's discussion is based in C, C compilers, Unix and Unix's login program. We will assume: a non-trivial security-critical program u (say, a login program), and its compiler c, also capable of compiling itself (c_s). We wish to construct a trojan $\widehat{u}$ that is undetectable, *even* given the assumption of the presence of source code u_s of u, which would have provided oracles.

The intention is to leave c_s and u_s unchanged but to have replaced c and u by $\widehat{c}$ and $\widehat{u}$ such that $c_s/\widehat{c} = \widehat{c}$ and $u_s/\widehat{c} = \widehat{u}$, and for $\widehat{c}$ otherwise to behave as c. Once this has been achieved, the trojans will be self-preserving: the trojan method cannot be eliminated easily since everything apart from the behaviour of $\widehat{u}$ will be indistinguishable from normal and it will probably be indistinguishable for 'long enough' from its expected behaviour.

First note that a trojan $\widehat{u}$ of u is easy to detect given u_s, since $\widehat{u} \neq u_s/c$ and we know what u is by applying u_s/c. In other words, with the source u_s we can determine that $\widehat{u}$ is a trojan. In practice one must check all (or only suspect) $u \in \mathsf{names}\, r$; however, $\mathsf{names}\, r$ is finite and each test is linear.

Suppose now that a compiler c' is constructed, where $s/c' \overset{\text{def}}{=} s = u_s \Rightarrow \widehat{u} : s/c$. When applied to u_s, c' trojanises it to $\widehat{u}$. Note that the test $s = u_s$ is an unproblematic test of identity of representations. Since in all other respects $c' = c$, c can be surreptitiously replaced.

At this stage, there is an undetectable trojan, but the tampering with the compiler is still readily detected since $c_s \neq c'_s$ and $c' \neq c$. The final stage of Thompson's argument removes this loophole.

A further compiler c'' is constructed, where $s/c'' \overset{\text{def}}{=} s = u \Rightarrow \widehat{u} : (s = c_s \Rightarrow c'' : s/c)$. This compiler has the remarkable property that it compiles the original c_s to itself, c'', and compiles the target program u_s to a trojan $\widehat{u}$. Since c_s and u_s are original, the new trojan is undetectable. The compiler c'' is bootstrapped as follows:

1. c''_s is constructed. This is easy, given c_s and the definition of c'' (above).
2. c''_s is compiled using the original compiler: $c''_s/c \mapsto c''$.
3. The original compiler's object code c is discarded and replaced by $\widehat{c} = c''$.
4. The source program c''_s is discarded.

Then the following ensues:

$$s/\widehat{c} = \begin{cases} \widehat{c}, & s = c_s \\ \widehat{u}, & s = u_s \\ s/c, & \text{otherwise} \end{cases}$$

The source program u_s can now be compiled by $u_s/\widehat{c}$ giving $\widehat{u}$ as required.

We now have a trojan $\widehat{c}$ and a trojan $\widehat{u}$ and no source code other than what is expected, u_s and c_s which have been restored to their originals. All programs compile correctly, except c_s and u_s themselves, but these two cases compile consistently, since the compiler has been trojanised to $\widehat{c} = c''$. The only way to show that $\widehat{u}$ is not in fact u is to find some $r: [\![\widehat{u}]\!]r \neq [\![u]\!]r$ — but there is no u available to do this, and even if there was, finding r would be exponentially hard. Login programs such as we have supposed u to be often have features in them specifically to make such operations difficult, since trying to find representations with particular properties has security implications. This trojan will be as difficult to detect as desired by its author.[9]

One can construct a system resistant to the Thompson attack by requiring $s/c \leftrightarrow s$ to be bijective; indeed, this is readily achieved in a system where representations are directly interpreted and/or there is no direct access to *compiled* forms. Alternatively, denial of access to the source of the compiler is sufficient, with the (awkward) proviso that the source is still required to determine whether the compiler is ever trojaned.

To show that Thompson's construction is sufficient to implement a trojan method, we need to consider his criteria for distinguishability. Having defined this, we must prove that the subverted representation is indistinguishable from the original representation. Thompson's trojan can then be a singleton trojan method provided it guarantees to manifest itself. As Thompson has not specified that it will manifest itself, we come to something of an impasse solved only in that trojans that do not manifest themselves are not a problem!

Thompson distinguishes programs by saying that two programs are indistinguishable if their source code is identical. This sounds constructive, but it is not quite in our framework. So we define two representations to be identical provided the source code of the program under the name of u is identical for both representations. This is poly log computable as the source for u is constant for all representations (given sensible definitions of representations). Unfortunately, this idea fails immediately — the trojan attacks the compiler and you can have identical source code on both representations but different object code. So two representations can be indistinguishable but have different object code for u and therefore be a trojan.

So we add another criterion: the source code for the code for u is the same on both representations, and the source code for the compiler is the same for both representations. (This is still poly log.) But this scheme fails with a trojan that attacks the compiler in a more sophisticated way.

In our terminology, Thompson gave two definitions of $\sim$ and both admitted trojans. The conclusion is not that Thompson trojans are not detectable but that reasonable-sounding definitions of distinguishability do not prevent there being trojans.

The theorem to prove is, given any poly log distinguishing relation, there is a trojan method for that relation. (There has to be some condition like poly log because comparing memory representations without restriction will clearly always show up trojans.) We will take up this challenge in a subsequent paper.

[9] Anyone who has bootstrapped compilers will know that discarding independent compilers (the initial p and the subsequent version history of compilers) is foolish: once bugs are introduced — not just deliberate trojans — they can remain even though subsequent compilations remove all signs of them in source code.

4.2. Detectability of viruses

Is virus detection decidable? If we were to define a relation p virus $\widehat{p}$ just when $\widehat{p}$ is virally related to p for some name l in some viral relation V, we cannot decide the relation because it is at least as hard as function equivalence.

Is virus activity detection decidable? This depends on the computational model assumed, but we can show the infection process can be inhibited under reasonable assumptions. If the environment is fixed, detection is trivial, there being no viruses to detect.

4.2.1. Cohen's proof of virus undetectability

The Cohen proof [9] of the non-computability of detection of viruses is a direct variant of the Halting Problem of Turing Machines, and is therefore subject to the limitations of Turing computability frameworks, as outlined above. The Cohen proof relates to the detection of viruses (i.e., assumed as defined objects), not their methods or behaviour, and it implicitly assumes a fixed Ω. Here we show that this standard proof (widely repeated, for instance in [8, 29] and elsewhere) about the detectability of virus infection is inadequate for a more interesting reason. (In criticising a proof the intention is to determine exactly what was proved, and whether advancements may be made by tightening the proof itself, its assumptions, or the theory in which it is expressed.)

We quote Cohen's central argument, then discuss a shortcoming:

> "In order to determine that a given program `P` is a virus, it must be determined that `P` infects other programs. This is undecidable since `P` could invoke any proposed decision procedure `D` and infect other programs if and only if `D` determines that `P` is not a virus. We conclude that a program that precisely discerns a virus from any other program by examining its appearance is infeasible. In the following [program `CV`, shown below], we use the hypothetical decision procedure `D` which returns "true" iff its argument is a virus, to exemplify the undecidability of virus detection.

```
contradictory-virus :=
{ ...
  main-program :=
  { if ¬ D(contradictory-virus) then
    { infect-executable;
      if trigger-pulled then
         do-damage;
    }
    go next;
  }
}
```

> [...] we have assured that, if the decision procedure `D` determines `CV` to be a virus, `CV` will not infect other programs and thus will not act like a virus. If `D` determines that `CV` is not a virus, `CV` will infect other programs and thus be a virus. Therefore, the hypothetical decision procedure `D` is self-contradictory, and precise determination of a virus by its appearance is undecidable."

We infer that `D` does not necessarily evaluate its argument when it attempts to determine whether it is a virus: clearly, to do so would run the risk of activating the virus itself. Cohen implicitly assumes this, since a conventional eager evaluation of his code would abort:[10] evaluating `contradictory-virus` would not terminate, and indeed would never progress beyond the conditional expression! Instead, `D` must examine its argument in some safe way which is not specified — the proof is assuming a syntactical representation of a virus. Cohen would like to prove that `D` cannot work however it is specified.

However, the code `infect-executable` or `do-damage` is not actually used in the proof, and therefore have only rhetorical value in making the program fragment look like a virus. Since, without affecting the proof scheme, any program code (say, `x:=0`) can be substituted (with the corresponding trivial changes to the specification of `D`) the putative proof is seen to be about the undecidability of program equality — not, more specifically, about virus detection.

We have here, then, an informal proof of a standard result, plus the unjustified assumption that viruses are *modelled* in the formalism of that proof, whereas here they have only been *named.* We agree that to prove that there is no decision procedure, one only needs to exhibit a counter example, but we do not agree that `contradictory-virus` is in fact an actual example of a virus. What has happened is that the names `infect-executable` and `do-damage` appeal implicitly to a virus method that may — or may not — be bound to these names *in the computer's representation.* The viral and trojan methods V, T such that

$$\begin{aligned} \langle \cdot, \cdot, \texttt{infect-executable} \rangle &\subseteq V \\ \langle \cdot, \cdot, \texttt{do-damage} \rangle &\subseteq T \end{aligned}$$

are not specified.

4.2.2. Restricted environments

Viruses require to be able to re-bind identifiers in the environment in order to propagate and cause damage. The obvious solution to this problem is to construct a system which never re-binds names. Though this might seem like a radical proposal it is common in many practical systems.

[10] In a programming language like Pascal, the parameters of a function call are evaluated before the function can be called. In Cohen's example, this would require invoking `contradictory-virus`.

It is worth making a small technical point here. In most operating systems, a file is bound not to its data (which is what we are modelling), but to where data may be located. In this case, a binding need not change even though the data is changed — for example, text editing a file still leaves it the same file, but with different content. We are deliberately not modelling where data is held, and therefore restricting an environment (in our model) to be non-rebinding is an effective restriction on changing the contents of a particularly-named file.

Non-rebinding would require that if $r \xrightarrow{l} r'$, where obviously $l \in \mathsf{names}\, r$, then $E(r) \subseteq E(r')$. From this restriction it is immediate that viruses can only infect 'new' name bindings introduced by their execution.

Many task-specific systems such as calculators, dedicated word processors, and personal organisers have fixed environments. Even for general purpose computers, many users might be happy to lock the environment so that no names can be rebound; this solution is implemented in a rather crude way in a number of proprietary hardware devices that write protect all of, or parts of discs, though the idea can be generalised [41]. As our framework indicates, though disc locking stops subsequent virus infection, it does nothing to help detect existing infections.

On many computers, the address space of a computer acts as an environment: it maps numbers (very easily generated names!) into representations. Hardware memory protection schemes are practical ways of restricting the address environment so that programs cannot generate names that are mapped into representations in other address spaces. This is quite conventional, but it is a useful example of a restricted environment whose use does not restrict higher level operations — indeed, the reliability and additional confidence about the behaviour of programs that it confers is highly desirable.

If a system includes a version number in the file names, then no name is ever re-bound, therefore it is impossible for a virus to 'infect' a system unobserved. The user should be able to tidy up the environment from time to time, but this could be a restricted facility requiring direct interaction with the user. The standard advice to users to make backups is no more than a manual (hence unreliable!) procedure to maintain such a non-rebinding environment.

Such a naming proposal seems to go a long way to protecting against viral damage provided the file system functions[11] are effectively virus-proofed. But this neglects a major component of our framework, namely observation (formalised by the $\sim$ notion that captures the idea that two representations cannot be distinguished by an observer). In many file systems using version numbers, the human interface to the unique names in the file system is constructed to make differences in version number unobservable, for example typically the most recent version of a file will be used by default. In order for the naming scheme to be effective the reverse must be the case — the user must see name changes when they are potentially damaging. This clearly requires major changes in the way user interfaces are constructed.

Turing Complete operations on environments (e.g., being able to compute names in an infinite domain) ensure that detection of infection is again undecidable. However, remaining within computability, we can arrange the environment so that certain computations are infeasible without passwords: for example, by using trapdoor functions. The relevance of trapdoors is that (under the appropriate security conditions) the observer and the virus stand on opposite sides of the trapdoor.

Suggested by the framework is the creation and use of names within a system: one can restrict the free use of names usually allowed in current systems. The names in the domain of the environment mapping can be encrypted, but accessed at the observer level via their unencrypted form, thereby making it arbitrarily harder for the virus to find bindings which could be changed unobserved. For example, a programmer writing a program to access a file server demands a key from the environment by supplying a name and password. This key accesses the binding of that name. Such a scheme would permit programs to compute names (in the encrypted domain of keys), but the probability of computing an unauthorised, but valid, name in the domain of the environment can be limited.

A variety of possible naming schemes might help: indeed it is possible to have an unbounded number of schemes, dynamically created. Various kinds of name servers which impose a management discipline on the creation and use of names could contain the spread of viruses to within any particular set of naming schemes. An obvious application of this is to ensure security when a system changes owner (e.g., when it is first or subsequently sold). A special case is when the naming schemes each contain precisely one binding.

It is often suggested that write-protected executables are immune from infection [33] (who claim that they are immune but impractical). This forgets the environment. If an executable program is unmodifiable that does not imply its access to the environment is unmodifiable: for example, a fixed program may compute names as arguments to the environment. A virus could therefore change the behaviour of a program by affecting this computation (e.g., by initialising it with different data). A realistic case of this situation is that of a program that runs some background server or other process: it computes a name to access the environment (e.g., in the simplest case, by reading a data file of server names) to load the background program, but a virus might simply cause it to load the wrong program.

The what-might-be-called 'the write-protected executable fallacy,' that one is supposedly safe when exe-

[11] More precisely, … functions on R.

cutable programs are write protected, confuses the security of the program for the security of the environment.

4.2.3. Viral resistance

In the practical use of a computer, the user only observes some of the outputs of a computation and only provides some of its inputs. The problem of viruses is that they are primarily concerned with inputs and outputs that the user normally ignores at the time of the computation. For example, the program code itself is not normally considered one of its own inputs, but this is precisely where most viruses reside, and how they directly affect the input of the program; a virus's output may alter files containing other programs, of which the user is unaware of.

A virally resistant system can be constructed by introducing observations O, which are to be communicated to the user. We extend $E\colon R \to (L \twoheadrightarrow R \times O)$ and $[\![\cdot]\!]\colon R \to (R \to R) \times (R \to O)$. Names are now bound to pairs $\langle p, o\rangle$ and the meaning of the pair is a pair of functions, one of which computes the result of doing the command and the other 'observes' the result to see it passes some checks. Observes, in the sense we are using it, means "prepared in some way that can be tested by a user."

In running a program $\langle p, o\rangle$ the system runs p as before to obtain the results and the new environment and runs o to observe the result, presenting the observation to the user. Programs have lots of inputs and results over the representation space, but a user's tests don't explore the domain uniformly, being only interested in conventional inputs — likewise, they only examine conventional outputs, say on the screen, not outputs that change files. The component o makes it clear that all the output must be observed.

By definition, a virus changes a binding of some name from $\langle p, o\rangle$ in the environment to some new $\langle \widehat{p}, \widehat{o}\rangle$. In general it is clearly not possible to compute $\widehat{o}$ from $\langle p, o\rangle$ to ensure that in an arbitrary environment $\widehat{o}$ computes the same value after a run of $\widehat{p}$ as o does after a run of p. The value of o must be interpreted by the observer; it is insufficient for o to yield a specific token (say `true`) for any authenticated binding, since any predetermined token can easily be computed by $\widehat{o}$. Thus given some notion of an external observer (e.g., the user) eventually any virus can be detected. Astute choices of o and observer make the likelihood of prompt detection much higher — the observer can be hardware (the range of o can be digital signatures).

A more intriguing idea is for the result of o to be a pattern (e.g., a video bitmap) and to rely on the human skill of recognising patterns and changes in patterns [34] — maybe a virus would show up as, say, an irritating line across the screen. This is an attempt at distribution free testing [30], aided by human sense perception. Distribution free testing is a more mechanical process that systematically samples the input/outputs so that o gives a 'fair' (distribution free) sample of the program's complete effect, though doing this depends on some properties of the program, but does *not* depend on knowing what the correct output should be. (Good cryptographic detection techniques are to some extent attempts to find suitable distribution free sampling functions.) Finally, so that it cannot be compromised, o may be implemented by hardware.

Implementations of such schemes must be undertaken very carefully, and some obvious implementations are suspect, simply because an implementation that (say) provides programs as pairs $(E \to R \times E) \times (E \to O)$ may accidentally provide operations that compromise the system. Thus, an unadorned Turing Machine can readily implement this scheme, but does not ensure that access functions for the pairs $\langle \widehat{p}, \widehat{o}\rangle$ are excluded: which, of course, would defeat the whole object of the distinction — it would be possible to construct an $\widehat{o}$ that simply replayed the output of o. See §4.3 for further discussion.

The invention of asymmetric (public key) cryptography challenged a central assumption, that security could be achieved through secrecy and obscurity. (Equally, the secrecy could conceal incompetence.) Now, new cryptographic algorithms have been widely published and widely scrutinised [50], and this scrutiny increases confidence in their effectiveness. It is interesting, then, to note that many discussions of viruses (e.g., [6]) do not wish to reveal anti-virus methods. Perhaps we need a similar breakthrough in viral resistance?

4.3. Virtual machines

Many programs (such as spreadsheets, language interpreters like TEX and commercial word processors)[12] introduce virtual machine environments. These virtual machines may be 'vectors' for infecting with viruses even though they run on otherwise secure operating systems. Virtual machine environments overcome attempts at protecting the implementation machine.

Since some programs written in a system $\mathcal{L}$ (BASIC, Java, Microsoft Word macros ...) need to (say) delete files, or have other permissions, then $\mathcal{L}$ needs those capabilities. An $\mathcal{L}$ system runs on a system which may itself be protected from virus activity, but the $\mathcal{L}$ system creates an environment for running $\mathcal{L}$ programs. This not only enables rebindings but changes the observed behaviour of the computer — of course, it must, since one wants to run the system $\mathcal{L}$! Thus $\mathcal{L}$ creates a virtual machine: a $\mathcal{L}$-machine simulated by the PC-machine. Clearly, our framework applies at each level of virtual machine and this has significant repercussions for the sort of virtual machine one would like to support in a secure system. In particular, the user interface

[12] Some authors call such viruses *macroviruses*; however, the viral methods typical of macroviruses (see [6]) are not restricted to macro languages *per se*. We suggest this terminology is misleading.

must make an *observable* distinction between each virtual machine (otherwise they could alias each other). Even in Java, which is designed with networked programming in mind, this distinction is made by libraries, not intrinsically.

The severity of the problem introduced by virtual machines is shown by Thompson's demonstration that explicit code (including code containing viral methods) in a virtual machine can be made to disappear from one level of the virtual machine by embedding it in an appropriate form in the implementation machine (§4.1.1). If the virtual machine supported is Turing complete and supports actions such as modifying the environment map (e.g., by permitting writing to files), then it is not possible to detect viral methods. All 'useful' virtual machines meet these two conditions.

4.4. A note on object-orientation

The increasing popularity of object-oriented programming and icon-based user interfaces (where there are very many observable objects in the environment) is based on claims on their efficiency and convenience of programming. Although the run time systems of object-oriented systems (Java being an example) may take steps to be secure, object-orientation itself is at odds with secure computation. To the extent that object-orientation has an impact on programmer convenience, it is clearly dependent on large numbers of computationally simple bindings. Inheritance is a problem because it provides a recursive environment. Indeed, Java has recently suffered from the `Strange Brew` virus, which infects the Java environment — and Java, being platform independent, ensures that the virus can run on almost any type of computer 19.

In systems that have inheritance, operations have default behaviour. Bontcher [6] gives several concrete examples based on a macro language. We give an abstract characterisation of one of his examples: suppose there is an easily recognised virus consisting of a set of macros, S. (Typically, one of the components will be activated by user activity, such as opening a file, and on running it will install another component as its payload.) A virus writer modifies S to make a variant. Now anti-virus software may recognise only the original components of this new virus, and eliminate them; however what remains may be an intact virus because the 'missing' components inherit default implementations. Ironically, this third, new, virus was created by the anti-virus procedure!

A thorough analysis of these issues is beyond the scope of this paper, except to note that any correct formal computation expressible in an object-oriented paradigm is expressible in another, possibly more secure, paradigm — but the real issue here is actually the trade-off between observable properties, the relationships of names in the environment and other aspects of usability and security.

5. KOCH'S POSTULATES

Robert Koch, the distinguished bacteriologist, contributed four criteria, known as Koch's Postulates, for identifying the causative agent of a particular biological disease.

1. The pathogen must be present in all cases of the disease;
2. The pathogen can be isolated from the host and grown in pure culture;
3. The pathogen from the culture must cause the disease when inoculated into a healthy, susceptible host;
4. The pathogen must be isolated from the new host and shown to be the same as the original.

To make sense of Koch's Postulates in our framework we may equate *pathogen* with *viral method.* It follows that a biological-type 'isolation' (Postulate 2) is non-computable. To the extent, then, that Koch's Postulates capture biological criteria, biological metaphors cannot be applied with any felicity to computer virus phenomena. Possibly Koch would have had a different view if biological pathogens were better able to mutate rapidly and maintain their (viral) method.[13] Because biological pathogens do not do this, Koch's Postulates can be usefully expressed with respect to representations rather than interpretations. A more appropriate biological metaphor for computer viruses is Dawkins's *meme* [13], for this corresponds to a software configuration running in the virtual machine provided by the hardware of a brain. (Dawkins makes explicit the connection with computer viruses.)

Given current interest in prions and transgenic infections (e.g., via xenotransplants) a formal framework for biological applications would be desirable. The way in which semantics in metainterpreters disappears (exploited in Thompson's trojan) obviously has profound implications, and may help understand prions. In any case, such results would certainly apply to replication using DNA. Unfortunately our framework makes certain assumptions that apply specifically to what might be called typical electronic digital computers: whilst certain sorts of computation can be performed to order (e.g., to detect infection), one is not prepared to devote excessive resources to this. In a biological context, the resources available and how they can be recruited are very different. Immune systems are massively parallel and autonomous, yet they are very slow to produce new antigens (vaccination is a rehearsal for the immune system). Biological replication, whilst comparatively slow, occurs in parallel at a molecular or species level but serially at an individual level. Computer viruses typically do not mutate using genetic algorithms, but rather use more specialised techniques (e.g., encryption) that guarantee viable replication. Thus there are significant

[13] Biological viruses mutate rapidly (in biological terms) but do not evolve rapidly [22].

differences, which are beyond the scope of this paper to explore satisfactorily.

Notwithstanding the fundamental biological differences, there is of course promise in biologically-inspired techniques for detecting and fighting viruses. See [17] for an insightful general discussion, and [25] which describes a prototype 'digital immune system.' (Coincidentally, the preceding article in the same journal gives examples of biological viruses that successfully suppress their hosts' immune systems [4]!)

6. CONCLUSIONS

A new framework has been introduced that appears to be better than previous attempts at addressing trojan and viral issues. Its main merit is that it is productive in raising and helping clarify the sorts of issues that need addressing. Although it abstracts away from the richness of the phenomena, it accounts for most of the concrete features: it makes clear that viruses are a very complex notion — involving the naming of objects, their behaviour and the observation of that behaviour.

Our framework for computer virus infection show that Koch's postulates are inadequate for the phenomena of computer viruses; in other words, the medical/biological metaphor for computer virus behaviour is seriously misleading.

A virus is a program that, in addition to having a trojan activity, infects other programs. We have shown that a Turing Machine equivalent model is insufficient to capture important details of virus behaviour. As contributions towards a theory of computer viruses we pointed out that formalism as such has no notion of expected behaviour, against which undesirable behaviour can be compared. Infection is with respect to an environment and must be identified by an observer using finitary tests. It follows that suitable constraints on environment operations can inhibit both trojan and virus infection.

We have given a proof that trojan code in general cannot be detected. Classes of trojan cannot be detected either, and this result puts limits on what can be expected of both pattern-matching type detectors and detectors that rely on intercepting certain sorts of behaviour. We have suggested various forms of observation as appropriate to control viruses.

We have shown that virus infection can be detected and limited. It follows that the spreading of viral methods can be restricted, but once infected by a virus there are limitations on what can be done to detect it, either by its unwanted behaviour, its code signature or any other characteristic. Whether users of computers would wish to convert to a new architecture more secure against infection is a question we do not address here; necessarily such computers would be incompatible with existing computers [40] — merely being discless network computers will not be sufficient.

Finally, we admit we are not yet satisfied. Although we have introduced and motivated important distinctions, the framework itself is unwieldy and the distinctions are hard to maintain in applied reasoning. It is hard to derive interesting theorems. Nevertheless we have successfully shown that viruses are a very complex phenomenon, despite frequently exhibiting such utterly banal behaviour that we would rather dispel them from our minds — if not just from our computers. Just as the current variety of viruses is not the last word in deviousness, our framework is not the last word in theoretical work with computer viruses. We hope our lasting contribution will be a greater awareness amongst system designers of the possibilities unnecessarily liberal programming environments provide hackers. We hope, too, to have challenged other theorists to pursue some of the interesting and important formal questions begged by taking viruses seriously.

ACKNOWLEDGEMENTS

Prof. Ian H. Witten (Waikato University, New Zealand) made very helpful suggestions for which the authors are grateful. The referees made useful comments that improved the presentation of the paper enormously.

REFERENCES

[1] Adleman, L. M. (1988) "An Abstract Theory Of Computer Viruses," in *Advances in Cryptology—CRYPTO'88*, Goldwasser, S., ed., Lecture Notes in Computer Science, **403**, Springer-Verlag: Berlin, pp354–374.

[2] Anderson, J. P. (1972) *Computer Security Technology Planning Study*, ESD-TR-73-51, **I** & **II**, USAF Electronics Systems Division: Bedford, Mass.

[3] Bates, J. (November 1990) "WHALE … A Dinosaur Heading For Extinction," *Virus Bulletin*, pp17–19. See [47].

[4] Beckage, N. E. (1997) "The Parasitic Wasp's Secret Weapon," *Scientific American*, **277**(5), pp82–87.

[5] Bissett, A. & Shipton, G. (1998) "Envy And Destructiveness: Understanding the Impulses Behind Computer Viruses," Proceedings Fourth International Conference on Ethical Issues in Information Technology, *Ethicomp'98*, pp97–108.

[6] Bontcher, V. (1998) "Macro Virus Identification Problems," *Computers & Security*, **17**(1), 69–89.

[7] Brunner, J. (1993) "Sometime In The Recent Future …," *New Scientist*, **138**(1868), pp28–31.

[8] Burger, R. (1989) *Computer Viruses, A High-tech Disease*, 3rd. ed., Abacus: Data Becker, Düsseldorf.

[9] Cohen, F. (1987) "Computer Viruses," *Computers & Security*, **6**(1), pp22–35.

[10] Cohen, F. (1989) "Computational Aspects Of Computer Viruses," *Computers & Security*, **8**(4), pp325–344.

[11] Cohen, F. (1994) *A Short Course On Computer Viruses*, 2nd. ed., John Wiley: New York.

[12] Coulouris, G. F. & Dollimore, J. (1988) *Distributed Systems*, Addison-Wesley: Reading.

[13] Dawkins, R., *The Selfish Gene*, 2nd. ed., Oxford University Press: Oxford, 1989.

[14] Denning, D. E. R. (1983) *Cryptography And Data Security*, Addison-Wesley: Reading, Mass.

[15] Ferbrache, D. (1992) *A Pathology Of Computer Viruses*, Springer-Verlag: London.

[16] Fokkinga, M. (1996) "Expressions That Talk About Themselves," *Computer Journal*, **39**(5), pp408–412.

[17] Forrest, S., Hofmeyr, S. A. & Somayaji, A. (1997) "Computer Immunology," *Communications of the ACM*, **40**(10), pp88-96.

[18] Goodenough, O. R. & Dawkins, R. (1994) "The "St Jude" Mind Virus," *Nature*, **371**(6492), pp23-24.

[19] Hancock, B. (1998) "Security Views (Java Gets a Foul Taste — First Reputed Java Virus)," *Computers & Security*, **17**(6), 462–474.

[20] Henderson, P. (1980) *Functional Programming*, Prentice-Hall.

[21] Hoffman, L. J. (1990) *Rogue Programs: Viruses, Worms And Trojan Horses*, Van Nostrand Reinhold: New York, p*xi*.

[22] Huang, A. S. & Coffin, J. M. (1992) "Virology: How Does Variation Count?" *Nature*, **359**(6391), pp107–108.

[23] Jones, S. K. & White, C. E. Jr. (1990) "The IPM Model of Computer Virus Management," *Computers & Security*, **9**(5), pp411–418.

[24] Kanada, Y. (1997) "Web Pages That Reproduce Themselves By Javascript," *ACM SIGPLAN Notices*, **32**(11), 49–56.

[25] Kephart, J. O., Sorkin, G. B., Chess, D. M. & White, S. R. (1997) "Fighting Computer Viruses," *Scientific American*, **277**(5), pp88–93.

[26] Ladkin, P. B. & Thimbleby, H. W. (1994) "Comments on a Paper by Voas, Payne and Cohen, 'A Model For Detecting The Existence of Software Corruption in Real Time'," *Computers & Security*, **13**(6), pp527–531.

[27] Lamport, L. (1994) "How To Write A Long Formula," *Formal Aspects of Computing*, **6**, pp580–584.

[28] Langton, C. (1988) "Artificial Life," *Artificial Life*, Santa Fe Inst. Studies in the Sciences of Complexity, Langton, C. (ed.), Addison-Wesley: Reading, Massachusetts, pp1–47.

[29] Leiss, E. L. (1990) *Software Under Seige*, Elsevier Science Publishers: Oxford.

[30] Lipton, R. J. (1991) "New Directions In Testing," Proceedings DIMACS Workshop in Distributed Computing and Cryptography, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, **2**, Feigenbaum, J. & Merritt, M., eds., pp191–202.

[31] Meinel, C. P. (1998) "How Hackers Break In... And How They are Caught," *Scientific American*, **279**(4), pp70–77.

[32] Nachenberg, C. (1997) "Computer Virus–Antivirus Coevolution," *Communications of the ACM*, **40**(1), pp46–51.

[33] Pozzo, M. M. & Gray, T. E. (1987) "An Approach To Containing Computer Viruses," *Computers & Security*, **6**(4), pp321–331.

[34] Race, J. (1990) "Using Computer Graphics To Find Interesting Properties In Data," *Computer Bulletin*, Series IV, **2**(7), pp15–16.

[35] Rhodes, R. (1994) "Chain Mail," *Nature*, **372**(6503), p230.

[36] Spafford, E. H. (1994) "Computer Viruses As Artificial Life," *Artificial Life*, **1**(3), pp249–265.

[37] Stevens, K. (1994) "Mind Control," *Nature*, **372**(6508), p734.

[38] Stevens, M. (1998) "Pest Control," *New Scientist*, **159**(2144), p64.

[39] Thimbleby, H. W. (1987) "Optimising Self-Replicating Programs," *Computer Journal*, **30**(5), pp475–476.

[40] Thimbleby, H. W. (1991) "Can Viruses Ever Be Useful?" *Computers & Security*, **10**(2), pp111–114.

[41] Thimbleby, H. W. (1991) "An Organizational Solution To Piracy And Viruses," *Journal of Systems Software*, **25**(2), pp207–215.

[42] Thimbleby, H. W. & Anderson, S. O. (1990) "Virus Theory," Institution of Electrical Engineers Colloquium, in *Viruses and Their Impact on Future Computing Systems*, Institution of Electrical Engineers Publication No. 1990/132, pp4/1–4/5.

[43] Thimbleby, H. W., Witten, I. H. & Pullinger, D. J. (1995) "Concepts Of Cooperation In Artficial Life," *IEEE Transactions on Systems, Man & Cybernetics*, **25**(7), pp1166–1171.

[44] Thompson, K. (1987) "Reflections On Trusting Trust," in *ACM Turing Award Lectures*, Ashenhurst, R. L. & Graham, S., eds., Addison-Wesley: Reading, Mass, pp171–177.

[45] Turing, A. M. (1939) "Systems Of Logic Based On Ordinals," *Proceedings London Mathematical Society*, Series 2, **45**, pp161–228.

[46] Virgil (19BC) *The Aeneid*, Book II.

[47] *Virus Bulletin*, ISSN 0956–9979.
URL: `http://www.virusbtn.com/`

[48] Wegner, P. (1997) "Why Interaction Is More Powerful Than Algorithms," *Communications of the ACM*, **40**(5), pp80–91.

[49] Witten, I. H., Thimbleby, H. W., Coulouris, G. F. & Greenberg, S. (1991) "Liveware: A New Approach To Sharing Data In Social Networks," *International Journal of Man-Machine Studies*, **34**(3), pp337–348.

[50] Zimmermann, P. (1995) *PGP: Source Code And Internals*, MIT Press.