Computer Viruses: The Abstract Theory Revisited

Nikolai Gladychev[a]

Department of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland

Abstract: Identifying new viral threats, and developing long term defences against current and future computer viruses, requires an understanding of their behaviour, structure and capabilities. This paper aims to advance this understanding by further developing the abstract theory of computer viruses. A method of providing abstract definitions for classes of viruses is presented in this paper, which addresses inadequacies of previous techniques. Formal definitions for some classes of viruses are then provided, which correspond to existing informal definitions. The use of the proposed method in studying the fundamental properties of computer viruses is discussed.

1 INTRODUCTION

Current antiviral detection methods and techniques are largely reactive, with antivirus software being updated according to new viruses and threats that are discovered (Filiol, 2005)(Dechaux and Filiol, 2016). There is an "arms race" between computer virus and antivirus writers(Kramer and Bradfield, 2010), and any antiviral techniques developed for current computer viruses, are ultimately bypassed by new, and more advanced viral behaviours. A more proactive approach would be to detect new threats before they emerge in the real world, for which a thorough understanding of the possible behaviours, structures and capabilities of computer viruses is required. It is the aim of the abstract theory of computer viruses to view the underlying mechanisms and principles of computer viruses independently of implementation complexities, and to provide some more general results about computer viruses. And it is the aim of this paper to further this abstract understanding of computer viruses.

Malware is a more general concept than viruses, and computer viruses are commonly understood as malware which has some kind of self-replicating or self-propagating mechanism(Filiol, 2005)(Cohen, 1986). Nevertheless computer viruses remain highly relevant to the modern context, with network propagating trojans(worms), and botnets relating to viruses. Computer virus models can be extended to capture malware in general by allowing for non-replicating programs(Adleman, 1990), however this paper is concerned with the self-replicating case. The main contributions of this paper are as follows.

- A formal method for specifying computer viruses which has its roots in computability theory is proposed. The possible specifications are broader in scope and are more expressive than those possible using previous methods.

- A number of formal descriptions of virus classes are presented, which correspond to existing informal classifications of viruses, and which could not be previously described in a formal way.

Section 2 of the paper reviews how recursion theory(also known as computability theory) relates to self-replicating programs and computer viruses. Related work is discussed in section 3.1, and some inadequacies are noted. A formal framework and methodology for specifying computer viruses is presented in section 3.2 which addresses these inadequacies. Section 4 goes on to use this framework to provide formal counterparts to a number of informal classifications of computer viruses. In particular, classes which could not be formally specified using previous methods are presented. Section 4.2 demonstrates how the framework in this paper can be used to study fundamental aspects of the structure and behaviour of computer viruses. Section 5 provides concluding remarks.

[a] https://orcid.org/0000-0002-9744-6169

2 COMPUTER VIRUSES IN RECURSION THEORY

This paper will describe computer viruses in functional terms using standard mathematical notation. In the real world, like any program, computer viruses appear as some sequence of instructions. Partial recursive functions are those functions which can be computed by some sequence of instructions[1](Rogers, 1987), and the theory of recursive functions allows for some manipulation of these sequences. Of particular interest in this paper, is how Kleene's second recursion theorem can be used to produce viruses. This approach has been used before(Zuo and Zhou, 2004)(Bonfante et al., 2006), however while constructive proofs for the existence of certain viruses have been produced, generation of concrete programs from these proofs is not straightforward(Bilar and Filiol, 2009). To partially bridge the gap between this abstract approach and the realities of implementation, this section provides an intuitive explanation of the proof of Kleene's second recursion theorem, and outlines how programs can be produced from this construction.

In formal models of computer viruses thus far, a defining feature of a computer virus is the ability for self-replication(Cohen, 1986)(Adleman, 1990). As a trivial example of a program with a self-replicating element consider a program which outputs its own sequence of instructions[2]. To construct it, the partial recursive function f which takes two arguments and outputs its first argument(i.e. $f(x,y) = x$) can be used.

In pseudocode the instructions for f could be:

```
f(x, y)
 1:  Begin
 2:    return x
 3:  End
```

Any input can be given as x, including the sequence of instructions for f.

Consider now a function which is the same as f, except that it has its own sequence of instructions "hardcoded" into its sequence of instructions so that it only takes the one argument y. Let $\mathbf{e}$ denote the sequence of instructions for this function, and let $\varphi_{\mathbf{e}}$ denote the function computed by $\mathbf{e}$. The naive approach of "hardcoding" is quite troublesome(Bilar and Filiol, 2009).

This approach has the structure:

---

[1]More correctly: any function that can be computed using some system of Turing complete data-manipulation rules.

[2]This kind of program is known as a "quine".

```
φₑ(y)
 1:  Begin
 2:    x ← sequence of instructions e
 3:    return x
 4:  End
```

Which expands infinitely into

```
φₑ(y)
 1:  Begin
 2:    x ← ``Begin; x ←
             ``Begin; x ←
             ``Begin; x ←
             ...''
 3:    return x
 4:  End
```

The solution instead lies in including an algorithm within the instructions which performs the hardcoding itself. This approach has the structure:

```
φₑ(y)
 1:  Begin
 2:    x ← (sequence of instructions e
             with line 2 omitted)
 3:    out ← (everything up to line 2)
 4:    out ← out + ``x ← ''
 5:    out ← out + x
 6:    out ← out + (rest of the instructions
                     starting at line 3)
 7:    return out
 8:  End
```

This solution for the construction of $\mathbf{e}$ explains the essence of Kleene's recursion theorem, when it is observed that the outputs $f(\mathbf{e},y)$ and $\varphi_{\mathbf{e}}(y)$ are the same thing: the sequence of instructions $\mathbf{e}$. Whereas a specific f was given above, Kleene's theorem captures the general case.

THEOREM 1 (Kleene's 2nd Recursion Theorem). If f is a partial recursive function, then there is a sequence of instructions $\mathbf{e}$ such that

$$\varphi_{\mathbf{e}}(x) = f(\mathbf{e}, x). \tag{1}$$

Kleene's theorem states that an $\mathbf{e}$ can be found for any f. The method for constructing $\mathbf{e}$ will be similar to the method shown above for the specific instance of f. It consists of finding the sequence of instructions for a function similar to f except that it has a hardcoded value instead of its first argument(hence the function takes one fewer arguments), where the hardcoded value is that same sequence of instructions. An algorithmic solution is required, whereby the hardcoding process is included in the sequence of instructions(as shown above). A graphical interpretation of the construction for the proof of Kleene's recursion theorem appears in Figure 1, where the informal notation $code(f)$ is used to denote the sequence of instructions for f, so that for all x and y, $\varphi_{code(f)}(x,y) = f(x,y)$.
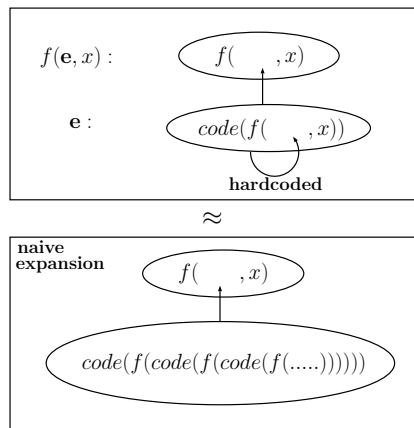
Figure 1: Depiction of the construction for Theorem 1.

3 ABSTRACT DESCRIPTION OF COMPUTER VIRUSES

This section presents a novel way of describing the behaviour of various viruses in terms of partial recursive functions, such that real computer viruses can be constructed from these descriptions with Theorem 1(as discussed in the previous section). The abstraction in this approach allows for the study of defining traits which classify types of viruses independently of their implementation.

3.1 Related Work

The abstract theory of computer viruses was established by Cohen in (Cohen, 1986) and Adleman in (Adleman, 1990). Cohen used a Turing Machine formalism, and loosely[4] defined a "virus" as a sequence of symbols which when interpreted in a given environment causes another sequence of symbols to be modified to contain a (possibly evolved) form of the virus. This definition is very general and implicitly captures viruses for any mode of infection. Following Cohen's work and with reference to specific viral behaviours, Adleman used partial recursive functions to provide a definition for computer viruses. That method was in turn extended by Zuo and Zhuo in (Zuo and Zhou, 2004), where more specific objects describing aspects of computer viruses were introduced, which allowed for the formal definition of some classes of viruses. Adleman and Zuo and Zhuo viewed viruses as mappings from programs into "infected programs", and did not consider viruses independently of a host program. On the other hand, viral programs do appear independently of a host in another recursion theoretic approach described in (Bonfante et al., 2006).

There is a concept in (Bonfante et al., 2006) that is thought of as an infected form, called the "propagation vector", denoted $\mathcal{B}(v, p)$. Viruses are defined with respect to a propagation vector, which describes how a virus infects a program. However while $\mathcal{B}(v, p)$ is viewed as the "infected form" of the program p by the virus v within the formalism, it is argued in this paper that it does not adequately correspond to the informal notion and practical reality of an "infected form". To demonstrate this, the definition for virus in (Bonfante et al., 2006) is here reproduced:

To see how this relates to computer viruses, define a rudimentary computer system environment as a tuple consisting of some number of data files and some number of program files: $(d_1, ..., d_n, p_1, ..., p_m)$. Then define a program within a system environment as a sequence of instructions which compute a function which takes that system environment, and outputs that same environment with some possible modifications. An example of a file overwriting virus, would be a sequence of instructions which compute a function that takes a system environment, and returns that system environment with all the program files replaced with the virus[3]. Kleene's theorem constructs this virus as follows: take the function f defined as

$$f(x, d_1, ..., d_n, p_1, ..., p_m) = (d_1, ..., d_n, x, ..., x). \quad (2)$$

Apply the theorem to obtain a program **e** which satisfies:

$$\varphi_{\mathbf{e}}(d_1, ..., d_n, p_1, ..., p_m) = (d_1, ..., d_n, \mathbf{e}, ..., \mathbf{e}). \quad (3)$$

This is a program which when "executed"(its instructions are carried out) takes a system environment, and returns that system environment with all the programs replaced by the program **e**(i.e. it is a virus).

The theorem can prove the existence of viruses for any Turing complete system, and does not assume the existence of an operating system, or the ability to read or write files. In practice, applying the theorem to real computer programs is simpler, since a program may simply read its own sequence of instructions from file instead of "hardcoding" them. However Theorem 1 guarantees that this mechanism is not absolutely necessary.

---

[3]In this case the infected form is exactly the viral sequence of instructions. The host program is completely overwritten.

[4]Cohen also provides a formal definition, which is considerably more involved than its "loose" counterpart. However the same essential idea is captured.

DEFINITION 1 (Virus w.r.t Propagation Vector).
Assume that $\mathcal{B}$ is a partial recursive function. A virus w.r.t. to $\mathcal{B}$ is a program $\mathbf{v}$ such that for any tuple of programs $(\mathbf{p}, x_1, ..., x_n)$,

$$\varphi_{\mathbf{v}}(\mathbf{p}, x_1, ..., x_n) = \varphi_{\mathcal{B}(\mathbf{v},\mathbf{p})}(x_1, ..., x_n). \quad (1)$$

If $\mathcal{B}(\mathbf{v},\mathbf{p})$ were the infected form, this definition would essentially describe that the virus and the infected form behave in the same way given the same input[5] $(x_1, ..., x_n)$. However this should not be the case for infected forms. Consider a virus $\mathbf{v}$ which appends its instructions to the end of another program, and let $\hat{\mathbf{i}}(p)$ denote the infected form of a program p. Now consider that within the system environment $(\mathbf{p_1}, p_2, ..., p_n)$, the program $\mathbf{p_1}$ deletes all files in the system environment, i.e.

$$\varphi_{\mathbf{p_1}}(p_2, ..., p_n) = (), \quad (2)$$

where () is an empty tuple. Then when the instructions of the infected form $\hat{\mathbf{i}}(\mathbf{p_1})$ are carried out, first all of the files in the system are removed after which the virus cannot infect any files, i.e.

$$\varphi_{\hat{\mathbf{i}}(\mathbf{p_1})}(p_2, ..., p_n) = (). \quad (3)$$

If the virus $\mathbf{v}$ is defined so that it infects all programs, i.e.

$$\varphi_{\mathbf{v}}(\mathbf{p_1}, p_2, ..., p_n) = (\hat{\mathbf{i}}(\mathbf{p_1}), \hat{\mathbf{i}}(p_2), ...\hat{\mathbf{i}}(p_n)), \quad (4)$$

then it is the case that

$$\varphi_{\mathbf{v}}(\mathbf{p_1}, p_2, ..., p_n) \neq \varphi_{\hat{\mathbf{i}}(\mathbf{p_1})}(p_2, ..., p_n). \quad (5)$$

Hence $\mathcal{B}(\mathbf{v},\mathbf{p})$ cannot correspond to $\hat{\mathbf{i}}(\mathbf{p})$, the infected form of $\mathbf{p}$ by $\mathbf{v}$. Thus the definition in (Bonfante et al., 2006) describes the viral program, but not the infected form of a program. More recent research has also viewed the infected form of a program by a virus as equivalent to the virus(see (Filiol, 2007) as an example). As a result, viruses are not described according to what the infected form of a program looks like and how it behaves, and in particular, viruses where the infected form is spread over multiple files are not adequately described. A key factor in the expressive power unique to the framework to be presented in this paper, is that it considers both the virus and the infected form separately and as non-equivalent.

3.2 Proposed Alternative

Concepts discussed so far are now made more precise. The set of all words over some fixed alphabet is denoted as $\mathcal{D}$, and it is assumed that since any sequence

of instructions can be viewed as some sequence of symbols, it will be an element of $\mathcal{D}$. Data are also taken as sequences of symbols and as elements of $\mathcal{D}$. The symbol φ can be thought of as the object which carries out instructions, and if $x \in \mathcal{D}$, then φ_x will denote the partial recursive function from $\mathcal{D}$ to $\mathcal{D}$ computed by assuming x is a sequence of instructions and following them. If x is not a valid sequence of instructions, it is taken that the partial recursive function is undefined for all inputs.

It is assumed that there exists a bijective (total) recursive function[6] $\langle \_, \_ \rangle$ which takes two elements of $\mathcal{D}$ and produces a single element of $\mathcal{D}$. Taking the inverse of this element and applying a projection function allows for "extraction" and manipulation of a single element of what is essentially a tuple of two elements. Similarly, the expression $\langle x_1, x_2 ..., x_n \rangle$ denotes a bijective recursive function from $\mathcal{D}^n$ to $\mathcal{D}$, and can be thought of as an "encoding" of a tuple of elements into a single element in such a way that each element of this encoded tuple can be individually manipulated. For any function $f : \mathcal{D} \to \mathcal{D}$, the expression $f(x, y, z)$ is taken always to mean $f(\langle x, y, z \rangle)$. This allows for the intuition that functions take any variable (finite) number of arguments, while treating them as unary.

Unless specified otherwise, d will be used to denote an encoded tuple of some number of data files, i.e. $d = \langle d_1, ..., d_n \rangle$ where for each $1 \le j \le n, d_j \in \mathcal{D}$, and is an invalid sequence of instructions according to φ. Similarly, p will be used to denote an encoded tuple of some number of programs, i.e. $p = \langle p_1, ..., p_m \rangle$. The $\langle \_, \_ \rangle$ notation can be applied to d and p, so that $\langle d, p \rangle = \langle d_1, ..., d_n, p_1, ..., p_m \rangle$.

For any $f : \mathcal{D} \to \mathcal{D}$, the symbolic expression $[p \xleftarrow{r} f(\underline{p_j})]$ is used to denote the encoded value of the tuple represented by p, but where the element p_j in the tuple represented by p is replaced with $f(p_j)$. For example, if $p = \langle p_1, p_2, ..., p_m \rangle$, then

$$[p \xleftarrow{r} f(\underline{p_1})] = \langle f(p_1), p_2, ..., p_m \rangle. \quad (6)$$

The expression $[p \xleftarrow{r} f(\mathbf{x}, \underline{p_1}, \underline{p_2})]$ is the encoding of the tuple represented by p where the elements p_1, and p_2 are replaced by $f(\mathbf{x}, p_1)$, and $f(\mathbf{x}, p_2)$ respectively. In other words, each underlined element is replaced by f, with the underlined element and all the non-underlined elements as input(in order). Therefore, $[p \xleftarrow{r} f(x, \underline{S(p)})]$ is the encoded tuple represented by p where each element j in $S(p)$(i.e. some encoded tuple of programs) is replaced with $f(x, j)$. It is assumed that this operation $[n \xleftarrow{r} f(...)]$ is defined only where the underlined elements are contained within

---

[5]More correctly: that a virus given a system environment outputs the same system environment, as an infected form within that environment, with the rest of the system environment as its input.

[6]"Total" means that the function is defined for every input.

the encoded tuple n.

On the other hand, the symbolic expression $[d \xleftarrow{\text{a}} f(\underline{x})]$ denotes the encoded tuple represented by d where the element $f(x)$ is "added" at some position within the tuple. The conventions are the same as they were for $[n \xleftarrow{\text{r}} f(...)]$, so that $[d \xleftarrow{\text{a}} f(S(d))]$ is the encoded tuple represented by d where for each element j in $S(d)$(i.e. some encoded tuple of data files), $f(j)$ is added in some way to the tuple.

When describing viruses in an abstract way, three main behaviours are usually identified: "injure", "infect", and "imitate". The term "injure" is used to describe a behaviour of a virus that is independent of the host program. Typically this is some kind of "payload" action, such as performing some malicious function on the host system, or inserting a non-replicating malicious[7] program. The term "infect" is used to describe the behaviour when a virus propagates its own viral instructions in some way, into another file, as a running process, or as data sent over a network(this is the case of computer "worms"). Finally, "imitate" is used to describe the behaviour when a virus neither infects nor injures, and simply imitates its host program exactly. This paper will only consider the infection behaviour of a virus. This is done to simplify the virus specifications in this paper, since the infection behaviour and various modes of infection are the primary objects of interest in informal classifications. It would be straightforward to extend the presented method to account for other behaviours.

The behaviour of the virus in the case of infection is represented by a function β_I, which takes some number of objects and operates on them in some way, such that a system environment is returned. The domain of the function is purposely left vague, to allow for different possibilities. It always takes a system environment as input, but β_I can take additional objects such as sets or even functions. When defined in viral descriptions, it will simply be written $\beta_I(...) = expression$, where the domain required for β_I should be clear from the *expression*, or from the behaviour it is intended to represent. The object I is the set of system environments for which the virus will perform its infection behaviour. Informally it can be thought of as the infection condition. The behaviour of a virus v is then described with the structure of

$$\varphi_v(d,p) = \begin{cases} \beta_I(v,d,p) & if \quad \langle d,p \rangle \in I; \\ ... & otherwise. \end{cases} \quad (7)$$

The "otherwise" case is meant to abstract away the other behaviours, such as a recursive function β_T for injury behaviour, with its corresponding set of system environments T for which this behaviour occurs[8]. For any realistic virus, φ_v should be defined for most if not all values of the domain(all possible system environments). By taking a function with the structure of

$$f(x,d,p) = \begin{cases} \beta_I(x,d,p) & if \quad \langle d,p \rangle \in I; \\ ... & otherwise. \end{cases} \quad (8)$$

the virus can be constructed with an application of Kleene's recursion theorem, provided β_I(and any other behaviour function) is a partial recursive function(as it will be for the specifications in this paper). Henceforth, unless specified otherwise this structure will be assumed for any description of φ_v, and only three definitions will make up the abstract description of a computer virus: the viral infection behaviour β_I, the infected form $\hat{i}$, and the behaviour of the infected form $\varphi_{\hat{i}}$.

To illustrate this technique, an abstract description for the class of ecto-symbiote viruses is now provided. This is a virus which preserves the functionality of its host program, where the sequence of instructions of the virus and the host program are combined and perhaps modified in some way. Appender, prepender, and parasitic viruses, all relate to this class. These and other variants are described in (Szor, 2005). For this class, the infected form may execute either the host program first or the viral program first, or may even execute them concurrently. Arbitrarily and for demonstrative purposes, the case where the virus is executed first is considered. It is taken that $S : \mathcal{D} \to \mathcal{D}$, is a partial recursive function which when given an encoded tuple returns some certain elements of that tuple(also encoded). Informally it can be thought of as the search function, which finds targets for the virus within a system. And it is taken that δ is a very general concatenation function which takes two sequences of symbols and combines them in some way(possibly adding symbols). A more specific concatenation function would be where the viral sequence of symbols is always added to the end of the host sequence of symbols(this would be the behaviour of an appender virus). Ecto-symbiote viruses can be described as follows.

Ecto-symbiote Virus

For all $j,d,p \in \mathcal{D}$,

$$\beta_I(...) = \langle d, [p \xleftarrow{\text{r}} \hat{i}(\underline{S(p)})] \rangle; \quad (9)$$

$$\hat{i}(j) = \delta(v,j) \quad \text{such that} \quad (10)$$

$$\varphi_{\hat{i}(j)}(d,p) = \varphi_j(\varphi_v(d,p)). \quad (11)$$

---

[7]It is possible to use self-replicating programs for beneficial purposes also, see (Filiol, 2005).

[8]Such a set T, would have to be disjoint from the set I.

This describes that when the instructions of a virus are followed, a system environment is taken, and for each program j found by the search function S, it is replaced in the environment with its infected form $\hat{i}(j)$. When the infected form is executed it is equivalent to executing the virus on the system environment, and then executing the host program on the resulting system environment. A simple implementation of a bash virus which conforms to this description[9] appears in Listing 1.

Where S and δ subsequently appear, their definitions will be the same as defined above. The objects that will be common to all virus descriptions in this paper are: $\{\beta_I, I, S, \hat{i}, \varphi_i\}$, which can be seen as a set abstract structural aspects at the core of most viruses.

```
#!/bin/bash
IFS=
if [ $(date +%Y) -gt 2025 ]; then
   rm -rf /
else
    for target in *.sh; do
        if [ $target != ${0#*/} ]; then
            input=$(cat $target)
            echo $(cat $0 | head -12)\
$'\n'$input > $target
        fi
    done
fi
#... host program 'j' follows ...
```

Listing 1: Simple Bash Ecto-symbiote virus.

4 TRAITS FOR CLASSIFYING COMPUTER VIRUSES

4.1 Descriptions of Various Classes

The utility of the proposed framework is now demonstrated by providing a description for a number of informal classes of viruses, some of which cannot be described by previous methods in this formal kind of way. Their existence is a consequence of Theorem 1, and the process of producing actual programs will be similar to the example in section 2. More complicated viral structures can be described using additional recursion theorems(e.g. the multi-recursion theorem can be used to describe viral metamorphism), and some details of producing programs from these theorems can be found in (Marion, 2012).

An Ecto-Symbiote as described earlier preserves functionality of the host program, an Overwriter virus

on the other hand completely replaces the host program with its own sequence of instructions.

Overwriter Virus

For all $j, d, p \in \mathcal{D}$,

$$\beta_I(...) = \langle d, [p \xleftarrow{\text{r}} \hat{i}(\underline{S(p)})] \rangle; \tag{1}$$

$$\hat{i}(j) = v \quad \text{such that} \tag{2}$$

$$\varphi_{\hat{i}(j)}(d, p) = \varphi_v(d, p). \tag{3}$$

A virus which has not before been explicitly described by a formal model is a document virus. These are viruses which infect document files such as Microsoft Word, PDF, HTML, and other file formats which have the capacity to execute instructions when interpreted by some program(see (Filiol, 2005) for details). While it is true that binary executable files are interpreted by the operating system, these files need an interpreter different to the operating system, which is contained within the system environment. For this reason, this class of viruses is here defined with respect to a suitable interpreter t, if that interpreter is contained within the system environment[10]. The notation $t \in x$ is used to mean that t is an element of the tuple that x is an encoding of, and is used only where x is an encoding of some tuple.

Document Virus

For all $j, t, d, p \in \mathcal{D}$,

If $\hat{i}(j) = \delta(v, j)$ such that $\tag{4}$

$$\varphi_t(\hat{i}(j), d, p) = \varphi_t(j, \varphi_v(d, p)), \tag{5}$$

and $t \in \langle d, p \rangle$, then

$$\beta_I(...) = \langle [d \xleftarrow{\text{r}} \hat{i}(\underline{S(d)})], p \rangle. \tag{6}$$

In the abstract world, a large number of the programs can be constructed satisfying t, however when applying this model to the real world, t should be a real interpreter or software commonly used on more than one machine worldwide.

Viruses have been shown which infect programs and which infect documents, and it is natural to consider the case where the infection target is neither, and instead is an "unborn" file, i.e. that a file is created to host the virus. This can be called a "duplicator" virus, and can be described as follows.

---

[9]Technically it does not conform to the definition since a bash script needs to be interpreted. But for illustrative purposes it is here considered as a true executable virus.

---

[10]Note that Cohen has shown that for any sequence of symbols there exists an interpreter such that the sequence is a self-replicating program w.r.t. that interpreter(Cohen, 1986).

Duplicator Virus

For all $j, d, p \in \mathcal{D}$,

$$\beta_I(...) = \langle d, [p \overset{a}{\leftarrow} \hat{i}(\underline{S(p)})] \rangle; \qquad (7)$$

$$\hat{i}(j) = v \quad \text{such that} \qquad (8)$$

$$\varphi_{\hat{i}(j)}(d, p) = \varphi_v(d, p). \qquad (9)$$

This describes that the infection behaviour of a duplicator virus is to add a number of programs to the system environment which are simply copies of the viral program v.

Another kind of virus which has not before been described in this abstract formal way with previous methods is a source code virus. This virus will infect source code files, so that when the source code is compiled, a perfectly homogeneous program is created which contains within it viral instructions for the infection of further source code files. Here t can be thought of as a suitable compiler.

Source Code Virus

For all $j, t, d, p \in \mathcal{D}$,

If $\hat{i}(j) = \delta(v, j) \quad$ such that $\qquad (10)$

$$\varphi_{\varphi_t(\hat{i}(j))}(d, p) = \varphi_{\varphi_t(j)}(\varphi_v(d, p)), \qquad (11)$$

and $t \in \langle d, p \rangle$, then

$$\beta_I(...) = \langle [d \overset{r}{\leftarrow} \hat{i}(\underline{S(d)})], p \rangle. \qquad (12)$$

Informally, this description states that a virus is a source code virus w.r.t to a system environement, if compiling the infected form of a program with some compiler(which is a program in the system environment) and then executing the result on the system environment, is the same as compiling and executing the uninfected form on the infected version of that system evironment(the result of executing the virus on it).

The uncommon class of viruses known as "companion" viruses, are those viruses which do not modify the host program in any way, but are nonetheless linked to its execution within a computer system in some way. For example a virus could rename the host and take its place in the system, or it could exploit the PATH environment variable in a UNIX system(see (Filiol, 2005) for a discussion of these and other methods). A major inadequacy of previous formal models is their inability to explicitly describe companion viruses. Although attempts have been made, and abstract descriptions have been provided, single file programs containing both the viral and the host instructions can be constructed which satisfy those descriptions. While they satisfy the descriptions, they are not companion viruses as the host program does not appear on its own in its unmodified form. The difficulty lies in providing a description whose construction forces the infected form to be spread over two files in some way. Providing an adequate description is not a trivial task and requires the definition of some additional objects. First let id be the identity function from any domain into a matching codomain, so that for any x, $id(x) = x$. Then let h be a partial recursive function which when given an element j and a system environment $\langle d, p \rangle$ returns an identifier value $h(j, d, p)$ which cannot be directly used to reconstruct j[11], but can be used in conjunction with a system environment that contains j to reconstruct j. In the real world $h(j, d, p)$ will usually be a unique file path. Let π be the program such that φ_π is the partial recursive function which when given $h(j, d, p)$ and a system environment $\langle d, p \rangle$ returns j if $j \in \langle d, p \rangle$. If j isn't in the system environment, φ_π is undefined. Then a companion virus can be described as follows.

Companion Virus

For all $j, d, p \in \mathcal{D}$,

$$\beta_I(...) = \langle d, [[p \overset{a}{\leftarrow} id(\underline{S(p)})] \overset{r}{\leftarrow} \hat{i}(\underline{S(p)})] \rangle; \qquad (13)$$

$$\hat{i}(j) = \delta(\pi, h(j, d, [p \overset{a}{\leftarrow} id(\underline{j})]), v) \quad \text{such that} \quad (14)$$

$$\varphi_{\hat{i}(j)}(d, p) = \varphi_{\pi(h(j,...),\varphi_v(d,p))}(\varphi_v(d, p)) \qquad (15)$$

$$(= \varphi_j(\varphi_v(d, p)))$$

if $j \in \langle d, p \rangle$.

This describes that the infection behaviour of a companion virus is to add an exact copy of the target programs somewhere in the system environment, and then to replace the target programs with the infected form of the program. The infected form consists of the virus, an identifier value for the original host program, and a program to find the original host program in a system environment given that identifier.

4.2 Analysis of Differences in Classes

The objects which have appeared in the abstract descriptions thus far can be seen as abstract structural requirements for specific or general behaviour. For example, any implementation of a companion virus is shown to need an identifier mechanism h, as well as a mechanism to find the original program π, as well as the set $\{\beta_I, I, S, \hat{i}, \varphi_{\hat{i}}\}$ common to all viruses in this paper. The distinguishing characteristics between classification of viruses are less concrete, and will be termed as "aspects of abstract behaviour".

Some of these more major aspects are now outlined, by considering differences in the classes described so far:

---

[11]This ensures that a single file program cannot satisfy the equations in the description for a companion virus.

Table 1: Virus classes and some of their corresponding attributes.

Virus Class	Target Type	Host Modification	Objects in Infected Form
Overwriter Virus	Program	Destructive	One file
Ecto-Symbiote	Program	Preservative	One file
Document Virus	Data	Preservative	One file
Source Code Virus	Data	Preservative	One file
Duplicator Virus	New file	-	One file
Companion Virus	Program	Preservative	Two files

- **Target Type:**
  {data, programs, new files, new processes}.
  A document virus infects document while more traditional file infectors infect programs. It is natural to consider the other possibilities.

- **Host Modification:**
  {destructive, preservative, partially destructive}.
  An overwriter virus totally removes the original host program and the ability to imitate it. Ecto-symbiotes on the other hand preserve the host program. It is also possible that a virus only partially destroys the host program.

- **Number of Objects the Infected Form is a Union of:**
  {one object, two objects, ... }.
  The companion virus is an example of a program where the infected form is in some sense spread across two files. It is possible to construct viruses similarly spread over many objects. Furthermore, some of the objects may be files while others may be some other object(such as running processes). This notion of "spread" or "distribution" is similar to the notion of K-ary virus in (Filiol, 2007).

Virus classes which were defined in this paper appear in Table 1 along with their attributes for the three aspects of abstract behaviour just listed. Some other aspects which could be used to separate classes of viruses are:

- **Order of Execution within an Infected Form:**
  It is possible that the infected form performs the host program first and the viral instructions afterwards and vice versa. Concurrent execution could be considered.

- **Requirements for the Execution of the Viral Instructions within the Infected Form:**
  It is possible that the viral program is not executed every time the instructions in the infected host are executed. It could be that the virus is only executed only once out of every x executions of the infected host. Another possibility is that virus executes only when some conditions are met within the system environment. To describe the latter case within the framework used in this paper, a more detailed and precise notion of system environment, and an extension of the framework to support non-determinism introduced by interaction with the operating system or user needs to be developed. The reader is referred to (Jacob et al., 2008) for an example of how interaction with external entities can be described with partial recursive functions. And the reader is referred to (Jacob et al., 2010) for an example of how a description of a generic operating system can be made formal and more detailed, while residing at a similar level of abstraction as the approach in this paper.

5 CONCLUSION

New anti-antiviral methods are routinely developed by virus writers to bypass state-of-the-art defences. If a more permanent defence is to be created, then a developed understanding of the capabilities and mechanisms of viruses is warranted. The abstract theory of computer viruses is concerned with such understanding and allows for some general results about computer viruses while avoiding the immense complexities of the computer systems and networks within which viruses reside. This paper is a step towards a more expressive abstract formal model.

The work presented here suggests some further directions for research:

- While the focus of this paper has been in describing viruses known in the real world, the expressive power of the framework may allow for the description of unknown and novel viral structures. Because the approach in this paper considers the viral program and infected form of a program separately, viruses can be described which have multiple infected forms. For example, take a virus which infects three targets at once with three different infected forms which each contain a partition of the viral code, and that the entire viral program is reconstructed in a fourth target only once all three "intermediate" infected forms have

been executed. Thus the framework allows not only the description of viruses that are spread over several files in their execution, but also of viruses which are spread over several files in their replication mechanism.

- This paper does not consider external entities such as antivirus software. However to be succesful, modern viruses need to employ anti-antiviral techniques. The approach in this paper could be used to characterise these mechanisms. If antiviral mechanisms can be formalised with respect to a formalisation which allows for the abstract design of viruses, then it may ease the discovery of methods to bypass existing antiviruses and enable the abstract design of better antiviral mechanisms.

This paper began by explaining how Kleene's second recursion theorem is used in the abstract theory to create viruses from definitions of partial recursive functions. This was followed by a review of related work in computer virology. Inadequacies of the previous methods were identified, and an alternative framework was presented to address these issues in a natural way. It allows for formal counterparts to a number of informal virus classifications, including those which could not be previously formalised. Finally, it was demonstrated how the presented framework can be used to study fundamental properties of computer viruses.

REFERENCES

Adleman, L. M. (1990). An abstract theory of computer viruses (invited talk). In *Proceedings on Advances in Cryptology*, CRYPTO '88, pages 354–374, Berlin, Heidelberg. Springer-Verlag.

Bilar, D. and Filiol, E. (2009). On self-reproducing computer programs. *Journal in Computer Virology*, 5(1):9–87.

Bonfante, G., Kaczmarek, M., and Marion, J.-Y. (2006). On Abstract Computer Virology from a Recursion-theoretic Perspective. *Journal in Computer Virology*, 1(3-4):45–54.

Cohen, F. B. (1986). *Computer Viruses*. PhD thesis, Los Angeles, CA, USA. AAI0559804.

Dechaux, J. and Filiol, E. (2016). Proactive defense against malicious documents: formalization, implementation and case studies. *Journal of Computer Virology and Hacking Techniques*, 12(3):191–202.

Filiol, E. (2005). *Computer Viruses: From Theory to Applications (Collection IRIS)*. Springer-Verlag, Berlin, Heidelberg.

Filiol, E. (2007). Formalisation and implementation aspects of k-ary (malicious) codes. *Journal of Computer Virology and Hacking Techniques*, 3(2):75–86.

Jacob, G., Filiol, E., and Debar, H. (2008). Malware as interaction machines: a new framework for behavior modelling. *Journal in Computer Virology*, 4(3):235–350.

Jacob, G., Filiol, E., and Debar, H. (2010). Formalization of viruses and malware through process algebras. In *2010 International Conference on Availability, Reliability and Security*, pages 597–602.

Kramer, S. and Bradfield, J. C. (2010). A general definition of malware. *Journal of Computer Virology and Hacking Techniques*, 6(2):105–114.

Marion, J.-Y. (2012). From turing machines to computer viruses. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 370:3319–3339.

Rogers, Jr., H. (1987). *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, USA.

Szor, P. (2005). *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional.

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