Malwares as Interactive Machines: A New Framework for Behavior Modelling

Grégoire Jacob[1/2], Hervé Debar[1], Eric Filiol[2]

[1] France Télécom R&D, Caen, France
{gregoire.jacob|herve.debar}@orange-ftgroup.com

[2] French Army Signal Academy,
Virology and Cryptology Lab., Rennes, France
eric.filiol@esat.terre.defense.gouv.fr

Abstract

Several semantic-based malware analyzers have recently been proposed but each one redefining its own model. All these systems and abstract virology models likewise, rely on Turing Machine equivalent formalisms. Recent works in the computer theory field have though shown that they prove insufficient in capturing wholly interactions and concurrency. By essence, malwares, as adaptable and resilient agents, are likely to use these mechanisms intensively. In this paper, we extend the malware models to the Interaction Machine formalism. This theoretical model is particularly adapted to apprehend these missing notions. We describe more precisely different classes of interactions and study their impact on the detection complexity. In a second part, we introduce an operational framework based on interactive languages, specifically designed for describing malicious behaviors. To complete our study and assess this framework, we provide descriptions for several behaviors usually used by current malware strains.

Keywords: Malware models – Behavior models – Interaction machines – Language theory – Interactive grammars – Detection complexity.

1 Introduction

This article relies on a very simple observation. By making a survey on the different techniques of behavioral detection, we have quickly noticed that a multitude of systems exist each one redefining its own behavior model. The underlying idea was then to provide a reference language for expressing these malicious behaviors. In a first place, we thought that Turing Machine languages were a good starting point, since most of abstract virology models and semantic detectors likewise, rely on Turing Machine equivalent formalisms. But along our work, we have gradually become aware that some dynamic notions were fundamentally missing in order to apprehend certain recent malicious trends. In a first part, we are thus going to state the known lacks of Turing Machines and equivalent models whereas in the following one, we will introduce the extended

model we have chosen as solution: Interactions Machines. According to this model, we will provide new definitions and complexity results. The second part of the paper is less theoretical and aim to provide a model framework still based on interactions. Application cases will finally be considered in order to assess the relevance of the model.

2 Shortcomings of the Turing Machine models

2.1 Actual models in abstract virology

It may be surprising, but as a matter of fact very few formal works have actually been published concerning models in abstract virology. Since the eighties, the release period of the first original concepts, about ten publications only can be listed. As we are going to base our speech on these models, we think that it is important to remind the most significant ones briefly. For those who would like to delve deeper into these questions, references are given for each model, otherwise a detailed survey is given in [1].

Based on self replicating cellular automaton introduced by J. von Neumann [2], F. Cohen was the first to establish a formal definition of a computer virus using Turing Machines [3]:

Definition 1 *According to Cohen, a symbol sequence is a virus with regards to a Turing Machine if, as a consequence of its execution, a possibly evolved copy of itself is written further on the band.*

Cohen's thesis supervisor, L. Adleman came up two years later with a more abstract formalization. He transposed the problem from a Turing Machine point of view, which is by nature linked to physical computers, to the more abstract theory of recursive functions [4]. He defined a virus as a function associating an infected form to each program. This infected form exhibits one of the following capabilities:
(1) **Injuring** where a malicious task is run instead of the intended one.
(2) **Infecting** where a malicious task is run once the intended one has halted.
(3) **Imitating** where only the intended program is run for stealth reasons.
This formalism has recently been extended by Z. Zuo and M. Zhou to introduce the mutation process and additional aspects such as stealth with regards to system calls [5].

Definition 2 *According to Adleman, a total recursive function v is a virus with respect to all Godel numberings of the partial recursive functions $\{\phi_i\}$ if and only if for all possible input x either:*

$$\begin{array}{llcl} (1) & (\forall p, q \in N) & \phi_{v(p)}(x) & = & \phi_{v(q)}(x), \\ (2) & (\forall p \in N) & \phi_{v(p)}(x) & = & v(\phi_p(x)), \\ (3) & (\forall p \in N) & \phi_{v(p)}(x) & = & \phi_p(x). \end{array}$$

G. Bonfante, M. Kaczmarek and J.-Y. Marion have provided a last formalism based on the foundation of computability which matches up with the previous models [6]. Based on the Kleene's recursion theorem, this model no longer considers the virus as a function but as a program making the notions of programming environment and program specialization available.

Definition 3 *According to Bonfante, Kaczmarek and Marion, a virus v is a program who, for all values of p and x over the computation domain D, satisfies the equation $\varphi_v(p,x) = \varphi_{\beta(v,p)}(x)$ where β denotes the propagation method.*

Even if their potential expression capabilities may differ, the three previous models in fact rely on equivalent formalisms which can be reduced to Turing Machines. It is commonly acknowledged that Turing-computability is equivalent to alternative notions of computability such as the foundation of recursive functions. This property of the Turing Machine fundamentalism is asserted indemonstrably by the Church-Turing thesis.

Thesis 1 *Every computable function can be computed by a Turing Machine.*

2.2 Known limitations

On the one hand, current abstract models have proved effective in capturing duplication, propagation and mutation concepts, but most important of all they have provided fundamental results on detection complexity. On the other hand, P. Wegner rightly underlines the fact that, if Turing Machines remain sufficient to model close system wholly determined by their input, they fail to model open systems [7]. Extending the formalism of replicating virus to more complex malwares will eventually fail because of important dynamic concepts missing as we will now state.

Interactions: Dynamic interactions with the external world, seen as ways to import and export data, are missing in simple Turing Machines. Interactions could be modelled using a shared band but this would lead to inconsistent accesses without additional management. Moreover interaction history remain unbounded to be strored on a band. Unfortunately, considering malwares, certain tasks can be entirely determined by stimuli or observations of their environment. An easy example concerning anti-antiviral techniques is the detection of an emulated environment which can prevent any malicious mechanism from being run. Other tasks can be manually controlled and triggered by inputs from the user or a remote attacker. Most of the non deterministic behaviors of malwares can be expressed by interactions with the external world, even tasks randomly executed which are mainly conditioned by an external random number generator. As an illustration, it could be interesting to introduce the concept of a malware with minimal code which makes the facilities provided by the environment work to its own benefit. Without embedding these functionalities, it could use any mail client present on the system to propagate or use any ciphering system to encrypt. Such a malware maximize interactions for the sake of adaptability.

Parallelism and distributivity: Turing Machines can model parallelism between programs as long as interactions are not considered. As we have just stated, interactions remain a central point in malware conception and thus can not be ignored. Unfortunately, if these processes are no longer independent but concurrent, according to R. Milner, sequential models such as Turing Machines are no longer sufficient [8]. Moreover Z. Manna

and A. Pnueli have shown that non-terminating reactive processes, such as operating systems, cannot be captured by these models [9]. This could be a major drawback, since malwares are highly adaptable programs making complex uses of the system facilities. Concurrency or distributivity in malware is achieved by its splitting into several modules. Typically, a main executable could be responsible for the infection while using a rootkit to execute particular actions requiring kernel privileges. Distributivity is not necessarily bound to the system. It can be performed over different connected systems (botnets networks) or even over physical devices of the local machine as long as they provide computing facilities (processors in graphic cards). Some malwares relying on distributivity to evade detection are already operational as mentioned in E. Filiol's recent paper on k-ary malwares [10, 11]. Such considerations are far too complex for a single Turing Machine to apprehend easily, and he proved that even Turing k-machines enable a partial generalisation of concurrent viruses since it is limited to a quadratic enhancement.

2.3 Related works and contribution

To our knowledge, only two related works have already tried to extend the viral models to take interactions into accounts. The first to achieve this was F. Leitold who has introduced a new mathematical formalism based on Random Access Stored Program Machines with Attached Background Storages [12]. These storage facilities are in fact additional bands with concurrent access in reading and writing modes, shared by all the processes. He proved that this model could capture communicating processes and operating systems. Unfortunately, this paper only considers particular interactions constrained by the band access and simply ignore most non-deterministic behaviors as the executed program is fixed. More recently, M. Webster has introduced another model based on Distributed Abstract State Machines to capture the virus's environment but only few details are briefly given since interactions are not the central point of the paper [13].

This paper intends to introduce a new formal model in order to describe malicious behaviors more completely with regards to interactions. This model is based on the established domains of language theory and interaction mechanisms. Notice that the descriptions generated by the language can then be applied either in static semantic detectors or dynamic monitors. To sum up our contribution:

- We divide interactions into several classes according to the nature of the considered adversary and the communication channel.
- We introduce a formal definition for recent malicious strains. These definitions have led to new detection complexity results.
- We provide an operational framework to model behaviors and assess its coverage in terms of soundness and completeness.
- We identify and describe several real behaviors in order to give hints of the language expressive power.

3 Interaction Machine based models

3.1 Theory of interactive machines

The shortcomings of the Turing model previously evoked are not really new, even A. Turing himself was conscious of certain gaps. Fortunately, several alternative extensions of Turing Machines have been put forward and in particular the Interaction Machines. According to the definition advanced by P. Wegner [14], an interaction machine can be described as a Turing Machine with dynamic input and output facilities.

Definition 4 *According to Wegner, Interaction Machines (IMs) extend Turing Machine (TMs) by adding dynamic input/output (read/write) actions. Interaction Machines may have single or multiple input streams and synchronous or asynchronous communications, and can differ along many other dimensions, but all Interaction Machines are open systems that express dynamic external behaviors beyond that computable by algorithms.*

Basically, an Interaction Machine has the same expressive power than a Turing Machine with oracles and infinite input [15]. Such a machine is called an Oracle Turing Machine or O-Machine and may have several oracles represented as immediate responses stored on additional bands [16]. The main interest is that an oracle can hypothetically solve any problem even undecidable and manipulate data of infinite size. With regards to Interaction Machines, the oracle can model the behavior of any adversaries of the machine taking the time and interaction history into account. Like pictured in the two formulae below, the input of the oracle model the data sent during interactions and the output, the data received. In case of unilateral interactions, either the input or the output can be null:

$I(data\ transmitted, time, interaction\ history) = data\ received$
$\Theta(data\ transmitted) = data\ received$

3.2 Abstract models for new classes of viruses

Based on this theory we will provide a model for two new classes of viruses. The first definition is based on the one of an implicit virus introduced by G. Bonfante et al. [6]. We will extend their definition to the concept of interactive virus using a formalism equivalent to O-Machine. Basically, the designated virus performs several actions depending on some conditions not only on its arguments but on its interactions with adversaries. In particular, these actions can take as parameters the results of these interactions.

Definition 5 *Let $C_1, ..., C_k$ be k semi-computable disjoint subsets of a computation domain D, $\Theta^1, ..., \Theta^n$ be the n oracles associated to n interactive adversaries and $V_{1,1}, ..., V_{n,k}$ be a set of semi-computable functions. An interactive virus v exists such that, for all p and x, the equation is satisfied:*

$$\varphi_v(p,x) = \begin{cases} V_{1,1}(v,p,x,\Theta^1(v)) & \text{if} \quad (p,x,\Theta^1(v)) \in C_1 \\ ... & \\ V_{n,k}(v,p,x,\Theta^n(v)) & \text{if} \quad (p,x,\Theta^n(v)) \in C_k. \end{cases}$$

Proof.
The proof is almost identical to the one developed for the implicit virus by G. Bonfante et al. [6]. Simply, using oracles allow us to consider the result of the interaction as computable and thus the same reasoning can be followed. $\forall i, \Theta^i(y)$ is considered as computable and thus semi computable. Let us define F such as:

$$F(y,p,x) = \begin{cases} V_{1,1}(y,p,x,\Theta^1(y)) & \text{if} \quad (p,x,\Theta^1(y)) \in C_1 \\ ... & \\ V_{n,k}(y,p,x,\Theta^n(y)) & \text{if} \quad (p,x,\Theta^n(y)) \in C_k. \end{cases}$$

As a composed of semi computable functions, F is semi-computable. By application of the recursion theorem, we obtain a program v satisfying $\varphi_v(p,x) = F(v,p,x)$. Let e be a program computing F and $\beta(v,p) = S(e,v,p)$ where S is the specialization function.

$$\begin{array}{rcl} \varphi_{\beta(v,p)}(x) & = & \varphi_{S(e,v,p)}(x) \\ & = & F(v,p,x) \text{ by the iteration theorem} \\ & = & \varphi_v(p,x). \end{array}$$

□

Example 1 *This definition makes it possible to define formally the contradictory virus introduced by Cohen to illustrate the detection undecidability [17]. Let us assume that the procedure D determining if a program is a virus is an interaction. We thus can describe the contradictory virus as follows:*

$$\varphi_v(p,x) = \begin{cases} \varphi_p(x) & \text{if} \quad \Theta^D(v) \in true \\ \varphi_{\beta(v,p)}(x) & \text{if} \quad \Theta^D(v) \in false \end{cases}$$

Example 2 *An other typical example would be a botnet. Each oracle Θ^n represents a possible command channel whereas the conditions C_k symbolize the different types of supported requests (DDos, Spam relay, Remote execution).*

Following the same formalism, we will now suggest the definition of a distributed malware. A distributed malware is made up of two or more programs executing and interacting. Distributivity can be seen as the interactive composition of several processes as suggested by P. Wegner [14]. For the sake of simplicity, if we consider the distributivity over two processes we have $Behavior(P|Q) = Behavior(P) + Behavior(Q) + Interaction(P,Q)$. This notion of distributed malware must be compared to the k-ary malwares introduced by E. Filiol [10, 11]. According to his definition, k-ary malware are made up of k files, executable or not, whose union constitute a malware and perform malicious actions.

Definition 6 *Let Θ be an oracle reflecting the interaction of two programs. The programs v and w are components of a distributed virus if there is a semi computable function f satisfying the equation:*

$$\varphi_{v,w}(p,x,y) = f(\varphi_v(p,x,\Theta^w(v)), \varphi_w(p,y,\Theta^v(w))).$$

Proof.
The proof is identical to the previous one as f is once again the composed of several semi computable functions. □

Remark 1 *In his work, E. Filiol defines two classes of k-ary malware. The first class I gathers together the sequential k-ary codes where the different components are executed sequentially, the actions of the first being used by the following. These are not considered as distributed malwares according to our definition as they do not interact dynamically. From the recursion, point of view, they can be seen as a simple composition:* $\varphi_{v,w}(p,x) = \varphi_v(p,x,\varphi_w(p,x))$. *On the other hand, the second class II of parallel k-ary codes is typically included in our definition.*

A definition of distributed malwares has been given for two components. Let us now extend our formalisation to n components. As stated by E. filiol, component interactions can be seen as graphs where vertices model the components and edges between two of them symbolize interactions. To simplify our definition, we will partition the graph in biconnected subgraphs in order to pinpoint the articulation vertices. As a consequence, it should reduce the complexity of the interaction network as pictured in Figure 1.

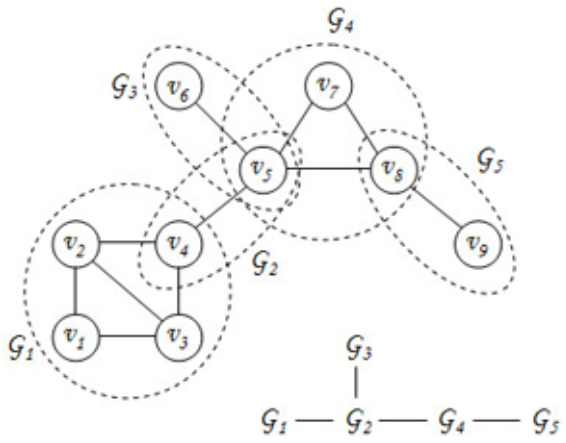

Figure 1: Distributed virus made up of nine components. This graph of interaction is given as an example and pictures a quite complex distribution. We can see that by searching for biconnected subgraphs we can decrease the complexity of interaction to a condensed graph.

Definition 7 *Let* G *be an interaction graph made up of* n *biconnected subgraphs* $G_1, ..., G_n$. *Locally to a subgraph, a simple active component* v_i *is connected to active components* $v_{i+1}, ..., v_{i+a}$ *and inert components* $d_j, ..., d_{j+b}$. *In the case of an articulation point, the component can also be connected to other subgraphs. Let* Θ^G,Θ^V *and* Θ^D *be respectively the oracles modelling the interactions between the different subgraphs, active and inert components. For the sake of notation simplicity,* $\Theta^V(v_i)$ *will denote the interaction history of* v_i *with all its connected active components* $v_{i+1}, ..., v_{i+a}$. *Equivalent notations will be used for the different oracles. The components of the graph* G *constitute a distributed malware if there are* $n+1$ *semi computable functions* $f, g_1, ..., g_n$ *satisfying the system:*

$$
\begin{cases}
\varphi_{G_1}(p,x) = g_1(\varphi_{v_1}(p,x,\Theta^V(v_1),\Theta^D(v_1)), \\
\qquad\qquad\qquad ..., \\
\qquad\qquad\qquad \varphi_{v_i}(p,x,\Theta^V(v_i),\Theta^D(v_i),\Theta^G(v_i))) \\
... \\
\varphi_{G_n}(p,x) = g_n(\varphi_{v_{z-j}}(p,x,\Theta^V(v_{z-j}),\Theta^D(v_{z-j}),\Theta^G(v_{z-j})), \\
\qquad\qquad\qquad ..., \\
\qquad\qquad\qquad \varphi_{v_z}(p,x,\Theta^V(v_z),\Theta^D(v_z))) \\
\varphi_G(p,x) = f(\varphi_{G_1}(p,x),...,\varphi_{G_n}(p,x))
\end{cases}
$$

3.3 Complexity of the detection problem

3.3.1 Classes of interaction

Interactions may be different according to the entities put into relation. By considering the different classes of interactions, we will be able to associate an equivalent complexity to the oracles modelling them.

(Class $\mathbf{I}_1$) Interactions with inert objects: This class gathers the interactions made with inert objects which have no internal mechanisms. Data files, registry entries and more generally storage memories and data repositories are typical examples. These interactions are always initiated by the observed program. In this case, the complexity is proportional to the size of the requested data.

Proposition 1 *The complexity of interactions with inert objects is linear or Σ_0. Notice that this type of interaction could be integrated by writing the object content on the tape of the Turing Machine without requiring an oracle.*

(Class $\mathbf{I}_2$) Interactions with active objects through interfaces: This second class gathers the interactions made with active objects which exhibit internal mechanisms constrained by defined interfaces. Kernel objects such as synchronisation objects are typical example. These interactions remain initiated by the observed program. Even when waiting for a remote activation signal, it can not be achieved without an explicit request from the program.

Proposition 2 *The complexity of interactions with active objects through defined interfaces is NP-Complete or Σ_1.*

Proof.
These active objects have limited internal mechanisms that can be seen as deterministic finite automaton. The object receives inputs and process them according to its internal state. According to the input it finishes in an accepting or error state. In these particular states, an output is sent back. The complexity is thus equivalent to the accepting problem of a word over a language. This particular problem is known to be NP-complete [18]. □

Example 3 *Contrary to what could be thought, network communications, thread communications through pipes likely, are practical examples from*

I_2. Even if the resulting value of the interaction is unpredictable, because these adversaries may not be controlled by the system for example, the interactions are constrained by a protocol defining the data conditions of transmission and consumption. Consequently, the manipulated data is size-bound by a factor s in bytes. It is thus possible to enumerate the possible results of the interaction. By neglecting the statistical biais due to the data nature, the set of possible results has at most 256^s entries which is exponential.

(Class I_3) Unconstrained interactions with adversaries: This last class gathers the unconstrained interactions with any active objects including human interventions. Contrary to the three previous ones, these interactions are not necessarily requested by the observed program. The typical case would be concurrent processes rewriting memory locations.

Proposition 3 *The complexity of free interactions with active objects is Undecidable.*

Proof.
Let P be the observed program, and Q a concurrent process. P uses the value stored in a memory space M without being aware that Q can modify it. M is left untouched by Q until the end of its process. When terminating, Q writes a different value in M. Knowing which value will be used by P is equivalent to know if Q terminates whatever its inputs are. The complexity of such interactions is thus equivalent to the halting problem which is undecidable. An example could be a rootkit modifying the system API addresses. Once loaded, it will have repercussions on the behavior of any program using system services. □

3.3.2 Impact of the interactions on detection

The complexity of interactions is not only determined by their nature but also by their combining. Their complexity are multiplied by a factor depending on the structure and perimeter of the observed system.

In the case of an interactive virus, we simply consider one to one interactions with the target of the observation. The factor is then directly proportional to the number of adversaries. The complexity of the oracle according to the interaction class is thus multiplied by a factor n.

In the case of distributed malwares, we consider multiple interactions between the adversaries. The complexity increases polynomially with the complexity of the interaction graph. The worst case is reached when the malware is a complete graph which can not be divided into biconnected subgraphs. The complexity is then multiplied by a factor $(n \times (n-1))/2$ corresponding to the maximum possible interactions. In both cases the increase induced by the combining factor is polynomial.

By extending the existing model with interactions, we can show that the detection complexity limited to the simple Turing machine is increased by the introduction of interactions. According to Bonfante et al. the set of the viruses

for a given propagation function is Π_2 [6]. Without taking the interactions into account (the oracle result is considered as a fixed input), this proposition can be directly applied to the set of the interactive viruses. We will now demonstrate that, without considering interactions, the sets of distributed viruses is also Π_2.

Proof.
Proof will be given for distribution over two components but can be generalized to n. Let q be a program computing the distributed propagation function f of the definition. The set of distributed viruses over two components is given by:

$$\forall x, y, p \exists y_1, ..., y_8 \begin{cases} (p, x, \Theta(v, w)) = y_1 \wedge (p, y, \Theta(v, w)) = y_2 \wedge \\ (p, x, y) = y_3 \wedge \varphi_v(y_1) = y_4 \wedge \\ \varphi_w(y_2) = y_5 \wedge (y_4, y_5) = y_6 \wedge \\ \varphi_q(y_6) = y_7 \wedge \varphi_{v,w}(y_3) = y_7 \end{cases}$$

□

Proposition 4 *By introducing the oracle complexity we obtain the following results. The set of interactive (resp. distributed) viruses for a given propagation function is respectively* Π_2, Π_3 *and undecidable according to the class of interaction considered.*

4 A formal semantic based on interactive machines for malware behaviors

The goal of this previous theoretical background was mainly to justify the importance of interactions and their impact on the detection. Based on the Interaction Machine formalism, it could now be interesting to establish a language in order to model malwares and in particular their behaviors. The formal grammars have the advantage of providing a better understanding of the malware effects, with great manipulation facilities, while remaining enough formal for a high level representation. This way, we migrate from abstract virology to a more operational context. With regards to intrusion detection, recent works underline the importance of generating a semantic traducing the intrinsic properties of the vulnerabilities rather than the exploits themselves [19]. Our guiding principle is similar, we think that it is important to describe the final purpose of a behavior rather than the technical solutions used to achieve it.

Several attempts to provide a semantic description of malwares' behaviors have already been made like the metalanguage introduced by Markus Schmall [20]. Finally, this first description was not enough formal to establish proofs about the language properties. Other semantics were introduced later based on simplified programming language [13] but they were not really intended to traduce the final purpose remaining at the assembly level. That is why we have decided to establish our own high level dedicated formalism which could then be declined into more concrete models or instanctiations by refinement.

4.1 Introduced framework

By choice, we have adopted an object oriented vision of the problem. The malware is thus considered as an object with internal attributes and mechanisms.

Additional interfaces are then provided for interaction with external objects. Before getting any further, let us begin with introducing our grammar describing the Malicious Behavior Language (MBL).

---

(1) $< Attribute >$ ::= $var|const$

(2) $< Op1 >$ ::= $\neg|\&$

(3) $< Op2 >$ ::= $\vee| \wedge | \oplus | < | \leq | = | \geq | >$
$| + | - | \times | \div | \equiv | << | >>$

(4) $< Value >$ ::= $< Attribute >$
$| [< Attribute >]$
$| < Op1 > (< Value >)$
$| < Op2 > (< Value >, < Value >)$

(5) $< Operation >$ ::= $var := (< Value >)$
$| [< Attribute >] := (< Value >)$
$| goto \quad < Attribute >$
$| stop$

(6) $< Adversary >$ ::= obj_perm
$| obj\_temp$
$| obj\_com$
$| obj\_boot$
$| obj\_exec$
$| obj\_sec$
$| env\_var$
$| this$

(7) $< Control >$ ::= $open|create|close|delete$

(8) $< I/O >$ ::= $receive \quad var \leftarrow < Adversary >$
$| receive \quad [< Attribute >] \leftarrow < Adversary >$
$| send \quad < Value > \rightarrow < Adversary >$
$| wait \quad < Adversary >$
$| signal \quad < Adversary >$

(9) $< Interaction >$ ::= $< Control >< Adversary > | < I/O >$

(10) $< Command >$ ::= $< Operation > | < Interaction >$

(11) $< Block >$ ::= $< Command >; < Block >$
$| < Command >;$

(12) $< Structure >$ ::= $< Block >$
$| if(< Command >)then\{$
$< Sequence >$
$\}else\{$
$< Sequence >$
$\}$
$| if(< Command >)then\{$
$< Sequence >$
$\}$
$| while(< Command >)\{$
$< Sequence >$
$\}$
$| [< Sequence > \| < Alternatives >]$

(13) $< Alternatives >$ ::= $< Sequence > \| < Alternatives >$
$| < Sequence >$

$(14)\ < \mathit{Sequence} > ::= < \mathit{Structure} >< \mathit{Sequence} >$
$\qquad\qquad\qquad\qquad | < \mathit{Structure} >$
$(15)\ < \mathit{Behavior} > ::= < \mathit{Sequence} >$

4.2 Internal mechanisms

Internal mechanisms are operations performed by the malware without requiring external interventions assuming that the processed data is available. Even if the data is originally supplied by an adversary, the data processing on its own is considered internal. With regards to the grammar, atomic internal operations are defined within the rules (1) to (5). These operations are then combined in blocks and structures according to the rules (10) to (15).

Proposition 5 *The MBL language is Turing-complete.*

Proof.
Proof is given in appendix by describing a Turing Machine with the MBL. □

Even if the proof of Turing Completeness states that our language is sound and complete with regards to internal mechanisms, we have shown in the part 2. that it remained insufficient to model the behavior of malwares. Notice that Turing Machine equivalent languages are the richest languages known to be both complete and sound.

4.3 Interaction extension

In fact, Interaction Machines extend the Chomsky hierarchy to the non computable domain as pictured in Table 1 [14]. As a consequence, Interaction grammars require additional dynamic features. In interactive languages, a notion of polarity is introduced for terminal grammar units [21]. A negative polarity indicates that the associated unit is in fact the result of an interaction and inversely for a positive value which is used as an input. Otherwise, the unit is said neutral. As a consequence, dynamic listening and transmitting operators are required to affect them dynamic values. The rules (7) and (8) defines dynamic interactive commands for listening and transmitting operations. The future possible values taken by the variables storing the results of these interactions are incrementally transformed into a sequential past at each computational step. These operators prove sufficient for modelling interaction of classes I_1 and I_2. In effect, they describe cases where the malware is set in a listening or transmitting state willingly. Notice that the *wait* and *signal* commands make it possible to distinguish between synchronous and asynchronous communications. On the other hand, interactions of class I_3 can only be modelled by non-deterministic choices requring a third additional operator. In our grammar, the respective operator is introduced in rule (12) with the notation $[s \parallel s']$. The choice between the different alternative sequences can be indirectly committed according to such unforeseen interactions. By nature, these behaviors are almost impossible to predict (see the undecadibility result in part 3.2) and thus can hardly be integrated to models in first place.

Proposition 6 *Soundness of the MBL with regards to interactions is quite intuitive considering the fact that the concept of object-oriented modelling is di-*

rectly inspired from the reality. On the other hand, completeness is impossible to achieve.

Proof.
We know that possible stream histories making up interactions are not recursively enumerable. Similarly to the Godël incompleteness for the integers, any domain whose set of true assertions is not recursively enumerable, can not be complete. Nevertheless, the partial completeness can be assessed through experiments. To do so, we will confront our model to a pool of real world cases. □

	Generative Models	Machine Models
Computable functions	Regular languages Context-free grammars Context-sensitive grammars Unrestricted rewriting rules	Finite automata Pushdown automata Linear-bounded automata Turing machines
Non-computable functions	Sequential interaction grammars Non-serializable grammars	Sequential interaction machines Non-serializable interaction machines

Table 1: Extended Chomsky Hierarchy. Interaction Machines extend the Chomsky hierarchy to the domain of non-computable functions by introducing dynamic operators and non determinism.

4.4 Adversaries classification

In order to extend our model we have chosen to affect object types to the adversaries. Using the object-oriented approach, we have put forward an inheritance scheme based on the final purpose of each object. Only classes with relevant effects on the malware lifecycle have been specified particularly. Every other object will be classified in two generic classes as pictured in Figure 2. This classification suggests a certain approach but remain open for discussion.

Basically, objects have been separated into two classes according to their persistence. The first one gathers the permanent objects (*obj_perm*) which remain present after a complete reboot of the system. Files, directories or registry keys are members of this class. At the opposite, temporary objects (*obj_temp*) exist only for a determined time as long as the system remains active. Mutex, events or critical sections are simple examples. Particular objects inheriting of these two classes are defined more specifically. In our grammar, the more specific class always prevail on the generic:

- The first subclass of the permanent objects is made up of the communicating object (*obj_com*). These objects are in fact communication channels to remote locations or systems. The definition of a communicating object is very large. Obviously network connexions, drivers are members of this class but also network drives, shared directories (intranet, P2P) or removable devices. In very particular cases such as pipes between processes they can also derived from temporary objects.

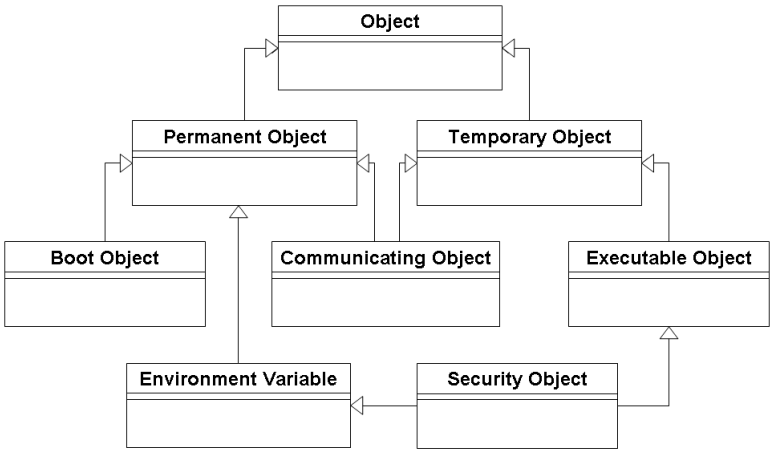

Figure 2: Adversary inheritance scheme. Any system object is either permanent or temporary making the classification complete. They have been derived into several specific classes according to the malware perspective. This classification is not fixed and can be the subject of enhancements.

- The second subclass gathers the boot objects (*obj_boot*). These objects provide the malware facilities to execute automatically its code. The run registry keys, the win.ini file for Windows or the master boot record make execution possible during the boot sequence. But automatic execution is also possible during runtime by overwriting the gobal system service descriptor table, the import tables or entry points in executables with the malware address. Such locations are also considered as members of the class.

- An other permanent subclass is made up of the environment variables (*env_var*). These objects store important information on the platform. Configuration files, registered path but also hardware fixed data structures available through particular instructions (cpuid) are just a few examples.

- Executable objects (*obj_exe*) constitute a fourth subclass inheriting from the temporary object. Process and threads in particular are appealing target for corruption by the malware.

- Security objects (*obj_sec*) is quite particular with regards to the others subclasses. These objects can be either environment variables or executables making this subclass hybrid. They play an important role in the protection of the system. They can be respectively antiviral processes or registry keys storing the security configuration for certain web or P2P clients.

- Ultimately, it shall prove useful to define an autoreference (*this*) as in object programming. This element has no particular type as it can be either the drive image of the malware, its associated process in memory. Such a reference can be obtained under Windows thanks to functions like GetCurrentProcess() or GetModuleHandle() called with a null value. It corresponds more simply to the $0 in a shell script.

5 Behavior modelling through interactions

5.1 Behaviors identified "in the wild"

In order to assess our model, we have chosen to confront it to existing malwares. To do so, we have proceeded to a behavior survey for several representative malicious strains. Thus, we have identified different techniques used to achieve several classes of typical malicious behaviors. Such information are partly available on observatory websites [22]. When deeper information were required, we have referred to detailed analysis of malwares in the wild [23, 24] or significant zoo examples [25, 26]. The results of this survey are given synthetically in appendix.

5.2 Specific behavior definitions

Based on this survey, we will now describe several malicious behaviors as subgrammars of the generative one. This means that any language generated by one of them is included in the language defined by our framework. Each of the used grammar unit can then be translated into several possible instruction metastructures by refinement from the abstraction to the implementation.

5.2.1 Replication mechanisms

Self replication is a key mechanism with viruses and worms. Most of the definitions put forward for these agents are based on this principle. In our description we have split replication according to three modes. The first one is a simple duplication where no target is required to host the code. The code is first stored in a local buffer symbolized by the generic variable V_{code}. It is then stored in a newly created permanent object O_{clone}. During the duplication, mutations can occur but these mechanisms shall not be described before the next section.

$V_{code} \in var$
$O_{clone} \in obj\_perm$

$$
\begin{array}{lll}
(i) & < Duplication > ::= & < Creation >< Reading > \\
 & & \quad < Mutation >< Writing > \\
 & & | < Reading >< Creation > \\
 & & \quad < Mutation >< Writing > \\
(ii) & < Creation > ::= & create \quad O_{clone}; \\
(iii) & < Reading > ::= & receive \quad V_{code} \leftarrow this; \\
(iv) & < Writing > ::= & send \quad V_{code} \rightarrow O_{clone};
\end{array}
$$

Contrary to duplication, infection requires an existing entity to host its code. As a consequence, the first phase of the replication always consists in crawling in the system to look for a potential target. In order to describe a valid target, conditions modelled by C_{valid} are defined on the nature of the target, one of them being to be not previously infected. An example could be the absence of an infected marker in a file such as a "magic constant". In our model we have integrated append and prepend modes of infections, whether destructive or not. In particular, the variable V_{save} is used as a buffer during the optional recopy of the original data. Once again mutations may intervene.

$V_{target}, V_{code}, V_{save}, V_{comparison} \in var$
$C_{valid} \in const$
$O_{target} \in obj\_perm$

(i) $< Infection >$ $::=$ $< Searching >< Opening >< Relocating >$
$< Reading >< Mutation >< Writing >$
$| < Searching >< Opening >< Reading >$
$< Relocating >< Mutation >< Writing >$

(ii) $< Searching >$ $::=$ $while(V_{comparison} := (\neg(= (V_{target}, C_{valid}))))\{$
$open \quad O_{target};$
$receive \quad V_{target} \leftarrow O_{target};$
$\}$

(iii) $< Opening >$ $::=$ $open \quad O_{target};$

(iv) $< Relocating >$ $::=$ $receive \quad V_{save} \leftarrow O_{target};$
$send \quad V_{save} \rightarrow O_{target};$
$| \epsilon$

(v) $< Reading >$ $::=$ $receive \quad V_{code} \leftarrow this;$

(vi) $< Writing >$ $::=$ $send \quad V_{code} \rightarrow O_{target};$

Propagation is a third way of replicating more specific to worm. Contrary to the two previous cases of local replication, propagation is the capacity to replicate over remote systems. The code is no longer copied in a permanent object but rather sent to a communicating object. According to the nature of the channel used, a formatting phase may be required. For example, mail propagation requires the construction of a mail structure with valid headers and the code of the malware attached encoded in a base 64 format. Notice that encoding the malware code may take several steps. Like any other replication mechanism, mutations are likely to occur.

$V_{code}, V_{formatted}, V_{parameter}, V_{position} \in var$
$C_{header}, C_{hsize} \in const$
$O_{channel} \in obj\_com$

(i) $< Propagation >$ $::=$ $< Opening >< Reading >$
$< Mutation >< Transmitting >$
$| < Reading >< Opening >$
$< Mutation >< Transmitting >$

(ii) $< Opening >$ $::=$ $open \quad O_{channel};$

(iii) $< Reading >$ $::=$ $receive \quad V_{code} \leftarrow this;$

(iv) $< Transmitting >$ $::=$ $send \quad V_{code} \rightarrow O_{channel};$
$| < Formatting >$
$send \quad V_{formatted} \rightarrow O_{channel};$

(v) $< Formatting >$ $::=$ $V_{position} := (\&(V_{formatted}));$
$[V_{position}] := (C_{header});$
$V_{position} := (+(V_{position}, C_{hsize}))$
$< Encoding >$
$[V_{position}] := (V_{code});$

(vi) $< Encoding >$ $::=$ $V_{code} := (< Op2 > (V_{code}, V_{parameter}));$
$< Encoding >$
$| \epsilon$

5.2.2 Mutation mechanisms

Up until now, mutations mechanisms have been mentioned without definition. We will now fill this gap. Mutations are mainly separated into two types of engine: polymorphic and metamorphic. Any one of them, both or none can be applied at the same time.

$$
\begin{array}{ll}
(i) < Mutation > ::= & < Polymorphism >< Metamorphism > \\
& \mid < Polymorphism > \\
& \mid < Metamorphism > \\
& \mid \epsilon
\end{array}
$$

Let us begin with polymorphism which is historically the first type of engine and thus the simpler. In polymorphism, the plain code is ciphered during its copy. As a matter of fact most of actual ciphering functions used by malware writers are simple binary operations like XOR applied with a constant key value V_{key}. Basically, our subgrammar is really similar to the behavior template described by Christodorescu et al. but with several extensions [27]. In particular, key variation and chaining like CBC have been added on the purpose of genericity. If this model only describes simple ciphering algorithms, it can be extended to more complex one as our grammar is Turing complete. A last point is the possibility to define a particular progression during the process. Certain algorithms such as in PRIDE (Pseudo-Random Index DEcryption [28]) have complex or even random memory accesses instead of sequential ones in order to delude emulators.

$$
\begin{array}{l}
V_{code}, V_{position}, V_{temp}, V_{key}, V_{posprev}, V_{previous}, V_{comparison} \in var \\
C_{limit}, C_{variation}, C_{progress} \in const \\
\begin{array}{lll}
(i) & < Polymorphism > ::= & V_{position} := (\&(V_{code})); \\
& & while(V_{comparison} := (< (V_{position}, C_{limit}))) \{ \\
& & \quad < Ciphering > \\
& & \quad < KeyVariation > \\
& & \quad < Next > \\
& & \} \\
(ii) & < Ciphering > ::= & V_{temp} := ([V_{position}]); \\
& & < Chaining > \\
& & V_{temp} := (< Op2 > (V_{temp}, V_{key})); \\
& & [V_{position}] := (V_{temp}); \\
(iii) & < Chaining > ::= & V_{posprev} := (-(V_{position}, 1)); \\
& & V_{previous} := ([V_{posprev}]); \\
& & V_{temp} := (< Op2 > (V_{temp}, V_{previous})); \\
& \mid & \epsilon \\
(iv) & < KeyVariation > ::= & V_{key} := (< Op2 > (V_{key}, C_{variation})); \\
& \mid & \epsilon \\
(v) & < Next > ::= & V_{position} := (< Op2 > (V_{position}, C_{progress}));
\end{array}
\end{array}
$$

The associated decrypting routine structure is almost identical to the mutation process as ciphering and deciphering algorithms are almost the same. According to the algorithms the key may vary or the arithmetic operations implied. The main difference relies in an additional jump for the malicious code

to gain control.

$(i) < DecryptRoutine > ::= < Polymorphism > \quad goto \quad V_{position};$

Metamorphism is much more complex to describe with formal grammar. In recent works, E. Filiol has given a definition of the metamorphism as a rewriting system transforming a grammar into an other [29, 11]. We will thus base our model on this definition establishing rewriting rules for our grammar. Metamorphic engines use four main types of techniques: reordering, register reassignement, garbage insertion and substitution with equivalent instructions. This last technique is partially addressed by working at the semantic level and thus shall not be described formally. In particular, in our formalisation, the use of different system services with varying parameters can be reduced to their basic interpretation as interactions bringing equivalences into light.

First technique is garbage insertion. Existing works already define the insertion of dead code as a grammar production rule [30]. This model considers only the insertion of nop equivalent instructions. In our model, we will extend the notion of garbage code to any sequence that once inserted does not modify any variable or interaction history of the original code. In order to define our rewriting rule, let us define a sequence S generated by our framework. Let $s_1, ..., s_n$ be any possible partition of S into n subsequences. Such a partition is always possible as soon as the sequence is not made up of a single command or a single structure.

$s_1...s_n \Rightarrow_R < Garbage > s_1 < Garbage > ... < Garbage > s_n < Garbage >$
with
$< Garbage > ::= < Sequence' >$
where $< Sequence' >$ has the same syntax than $< Sequence >$ but for all variable v and object o of S, we have $v \notin L(Sequence')$ and $o \notin L(Sequence')$. The sequence is thus defined on a restraint spaces for variables $var \setminus \{v \in var | \exists i, v \in s_i\}$ and objects $L(< Object >) \setminus \{o \in L(< Object >) | \exists i, o \in s_i\}$.

We will use the same notation in order to define code reordering. The sequence is once again partitioned and then recombined according to any possible permutation of the subsequence $s_i...s_j$. Jump are then introduced in order to maintain the correct control flow.

$s_1...s_n \Rightarrow_R goto \quad V_{address_1}; s_i; goto \quad V_{address_{i+1}}; ...; s_1; goto \quad V_{address_2}; ...; s_n$

As we are working at a semantic level, the problem of register reassignement is already addressed using generic variables. But we will once again extend the notion of register reassignement to the more generic principle of variable reassignement.

$S \Rightarrow_R V_{new} := (V_{old}); S[V_{old}/V_{new}]$
where $S[V_{old}/V_{new}]$ is equal to S where all occurrences of V_{old} are replaced by V_{new}.

These rules describe the techniques usually used by malware writer but E. Filiol has shown that by choosing more thoughtfully these rewriting rules it is

possible to generate mutating malwares whose form-based detection is undecidable [29, 11]. Based on the word problem stated by Emile Post, PBMOT is an engine he developed as a proof of concept.

5.2.3 Overinfection and activity tests

The overinfection test proves useful to detect if any instance of the malware is present on the system. It is done by checking the existence of a permanent marker O_{marker}. This test can be achieved through at least three different methods. Notice that in the case of file infection, the test to know if a target is healthy, is already integrated in the searching routine and thus does not need to be redefined here.

$O_{marker} \in obj\_perm$

(i) $< Overinfection > ::= < Test1 > | < Test2 > | < Test3 >$

(ii) $< Test1 > ::= if(create \quad O_{marker})then\{$
$stop;$
$\}$

(iii) $< Test2 > ::= if(open \quad O_{marker})then\{$
$stop;$
$\}else\{$
$create \quad O_{marker};$
$\}$

(iv) $< Test3 > ::= if(open \quad O_{marker})then\{$
$create \quad O_{marker};$
$\}else\{$
$stop;$
$\}$

If overinfection test addresses the static problem, the activity test deals with the dynamic aspect making it possible to detect if an instance of the malware is already running in memory. The execution is betrayed by the presence of a particular temporary object O_{active}. Otherwise the structure is quite similar to the previous one.

$O_{active} \in obj\_temp$

(i) $< Activity > ::= < Test1 > | < Test2 > | < Test3 >$

(ii) $< Test1 > ::= if(create \quad O_{active})then\{$
$stop;$
$\}$

(iii) $< Test2 > ::= if(open \quad O_{active})then\{$
$stop;$
$\}else\{$
$create \quad O_{active};$
$\}$

(iv) $< Test3 > ::= if(open \quad O_{active})then\{$
$create \quad O_{active};$
$\}else\{$
$stop;$
$\}$

5.2.4 Residency mechanism

Residency is a way for the malware to trigger its execution automatically. It is achieved by writing its reference $V_{reference}$ in a boot object O_{run}. According to the object used, the nature of the reference will be different. For a run registry key, it will be its path in the file system whereas for import tables or entry points, it will be its address in memory.

$V_{reference} \in var\ O_{run} \in obj\_boot$
$(i) < Residency > ::= send \quad V_{reference} \rightarrow O_{run};$

5.2.5 Anti-antiviral mechanisms

According to the principle, the best defense is attack, malware sometimes deploy proactive protections. The malware will try to delete security files or terminate antivirus processes in order to execute freely.

$O_{protect} \in obj\_sec$
$(i) < Proactive > ::= delete \quad O_{protect};$

An other form of proactive protection is the modification of the security policy. Most of programs, even the operating system store this information in policy objects O_{policy} like registry keys or configuration files. The current configuration is thus replaced by the weaker possible.

$V_{weaker} \in var$
$O_{policy} \in obj\_sec$
$(i) < Policy > ::= open \quad O_{policy};$
$send \quad V_{weak} \rightarrow O_{policy};$

These two techniques are quite aggressive and they are toughly monitored by antivirus and HIPS. There are other ways more subtle to avoid detection such as preventing the capture of any information betraying the malicious activity. In order to analyse malwares, they are often primarily run in an emulated environment. Such a virtual system can be detected because it does not match up entirely with a real one. Typically, the redpill technique is based on this kind of comparison by reading the CPU structure thanks to the cpuid instruction [31]. In case of detection, the malware can execute a legitimate sequence or simply stop.

$V_{read}, V_{comparison} \in var$
$C_{expected} \in const$
$O_{infostructure} \in env\_var$
$(i) < DetectEmulator > ::= receive \quad V_{read} \leftarrow O_{infostructure};$
$if(V_{comparison} := (= (V_{read}, C_{expected})))then\{$
$< Sequence >$
$\}else\{$
$< Sequence >$
$\}$

A second technique is stealth. A virus is said stealthy with regards to its environment if no reference is made to it in the information structures controlled by the system. In the terms of our grammar, it could be translated by the following result: $env\_var \cap this = \oslash$. For example no reference to the malware should be clearly visible in the file system tables or the process list. In order to achieve this we will define means for a malware to be stealthy relatively to system calls by replacing them with altered functions. There are two basic cases. Either the malware is specifically targeted by the call through the parameters and then its reference should be locally replaced by a benign parameter. Or the function returns data likely to contain this reference knowing that this data can be an explicit value or an address toward a complex structure requiring analysis.

$V_{parameter}, V_{return}, V_{benign}, V_{position}, V_{size}, V_{value} \in var$
$C_{this} \in const$

(i) $< StealthFuntion > ::= < Preprocessing >$
$< SysCall >$
$< Postprocessing >$

(ii) $< Preprocessing > ::= if(V_{comparison} := (= (V_{parameter}, C_{this})))then\{$
$V_{parameter} := (V_{benign});$
$\}$
$| \epsilon$

(iii) $< Postprocessing > ::= if(V_{comparison} := (= (V_{return}, C_{this})))then\{$
$V_{return} := (V_{benign});$
$\}$
$| V_{position} := (V_{return});$
$while(V_{comparison} := (< (V_{position}, V_{limit})))\{$
$V_{value} := ([V_{position}]);$
$if(V_{comparison} := (= (V_{value}, C_{this})))then\{$
$[V_{position}] := (V_{benign});$
$\}$
$V_{position} := (< Op2 > (V_{position}, C_{progression}))$
$\}$
$| \epsilon$

---

6 Conclusion and perspectives

Through this paper, we have introduced a new framework based on interactions in order to describe malicious behaviors. The first theoretical approach has given results measuring the heavy impact of interactions on the detection complexity, thereby justifying their consideration. We have then provided a semantic that seems relevant with respect to our survey since we have managed to describe most of the identified behaviors. In order to achieve a greater completeness, the scope of the survey should be increased to a wider range of malwares. Anyhow, the generative grammar proves to be sufficiently generic to define additional behavior or refine exisitng descriptions. Additional behaviors such as data gathering or typical final payload could have been described but we had to limit ourselves not to drown the important facts among examples. Eventually this grammar is proposed as a base and can be extended for specific purposes.

Working at a higher level of representation has several advantages. It proves really useful in expressing the final aim of behaviors rather than the techniques used to achieve it. Moreover this semantic brings into light functional similarities more evolved than simple instruction equivalence which is the major drawback of most current detection systems. Eventually, it could be worth considering integrating our framework to existing semantic analysis systems for malware detection as in [27, 32, 33].

The inheritance scheme for adversaries is also an intearesting features since it helps to understand the relation between a malware and its environement. Studying this classification further would help to refine the scheme and bring additional information about the data flow. An other way to characterize this flow would be by exploring deeper the existing interaction semantics and in particular π-calculus for a more proper theoretical formalism than oracles [8, 21]. As recently stated by J-Y. Marion, using this formalism should bring into light possible access restrictions to hinder the malware propagation [34].

References

[1] E. Filiol, *Computer Viruses: From Theory to Applications*. Springer, IRIS Collection, 2005, ISBN:2-287-23939-1.

[2] J. von Neumann, *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966, ISBN:0-598-37798-0.

[3] F. Cohen, *Computer Viruses*. PhD thesis, University of South California, 1986.

[4] L. M. Adleman, "An abstract theory of computer viruses," in *CRYPTO '88: Proceedings on Advances in cryptology*, pp. 354–374, 1990.

[5] Z. Zhihong and M. Zhou, "Some further theoretical results about computer viruses," *The Computer Journal*, vol. 47, no. 6, pp. 627–633, 2004.

[6] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, "On abstract computer virology from a recursion theoretic perspective," *Journal in Computer Virology*, vol. 1, no. 3-4, pp. 45–54, 2006.

[7] P. Wegner, "Why interaction is more powerful than algorithms," *Communications of the ACM*, vol. 40, no. 5, pp. 80–91, 1997.

[8] R. Milner, "Elements of interaction: Turing award lecture," *Communications of the ACM*, vol. 36, no. 1, pp. 78–89, 1993.

[9] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., 1992, ISBN:0-387-97664-7.

[10] E. Filiol, "Formalisation and implementation aspects of k-ary (malicious) codes," *Journal in Computer Virology*, vol. 3, no. 3, EICAR 2007 Special Issue, V. Broucek Ed., 2007.

[11] E. Filiol, *Techniques Virales avancées*. Springer, IRIS Collection, 2007, ISBN:2-287-33887-8.

[12] F. Leitold, "Mathematical model of computer viruses," in *Best Paper Proceedings of EICAR*, pp. 194–217, 2000.

[13] M. Webster, "Algebraic specification of computer viruses and their environments," in *Selected Papers from the First Conference on Algebra and Coalgebra in Computer Science, Young Researchers Workshop (CALCO-jnr), University of Wales Swansea Computer Science Report Series CSR 18-2005, P. Mosses, J. Power, and M. Seisenberger Eds.*, pp. 99–113, 2005.

[14] P. Wegner, "Interactive foundations of computing," *Theoretical Computer Science*, vol. 192, no. 2, pp. 315–351, 1998.

[15] P. Wegner, "Interaction as a basis for empirical computer science," *ACM Computing Surveys*, vol. 27, no. 1, pp. 45–48, 1995.

[16] M. J. Atallah, *Algorithms and Theory of Computation Handbook*. CRC Press LLC, 2000.

[17] F. B. Cohen, "Computer viruses: Theory and experiments," *Computers & Security*, vol. 6, no. 1, pp. 22–35, 1987.

[18] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages and Computation, Second Edition*. Addison Wesley, 1995, ISBN:0-201-44124-1.

[19] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 2–16, 2006.

[20] M. Schmall, *Classification and Identification of Malicious Code Based on Heuristic Techniques Utilizing Meta-languages*. PhD thesis, University of Hamburg, 2002.

[21] G. Perrier, "Interaction grammars," in *Proceedings of the 18th conference on Computational linguistics - Volume 2*, pp. 600–606, 2000.

[22] "Fortinet observatory." url=`www.fortinet.com/FortiGuardCenter/`.

[23] K. Rozinov, "Reverse code engineering: An in-depth analysis of the bagle virus," in *Proceedings of the 2005 IEEE Workshop on Information Assurance*, pp. 178–184, 2005.

[24] E. Filiol, "Le ver mydoom," *MISC - Le magazine de la sécurité informatique*, vol. 13, 2004.

[25] P. Ferrie, "Magisterium abraxas," in *Proceedings of Virus Bulletin*, pp. 6–7, 2001.

[26] P. Ferrie and H. Shannon, "It's zell(d)ome the one you expect - w32/zellome," in *Proceedings of Virus Bulletin*, pp. 7–11, 2005.

[27] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantic-aware malware detection," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 32–46, 2005.

[28] T. M. Driller, "Advanced polymorphic engine construction," *29A E-zine*, vol. 5, 2003.

[29] E. Filiol, "Metamorphism, formal grammars and undecidable code mutation," in *Proceedings of the International Conference on Computational Intelligence (ICCI)*, 2007.

[30] Qozah, "Polymorphism and grammars," *29A E-zine*, vol. 4, 1999.

[31] J. Rutkowska, "Red pill... or how to detect vmm using (almost) one cpu instruction," 2005, url=`http://invisiblethings.org/papers/redpill.html`.

[32] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," *Lecture Notes in Computer Science*, vol. 3548, pp. 74–187, 2005.

[33] J. Shin and D. Spears, "The basic building blocks of malware," tech. rep., University of Wyoming, 2006.

[34] J.-Y. Marion, G. Bonfante, and M. Kaczmarek, "Experiments with recursion theory," in *2nd Workshop on the Theory of Computer Viruses (TCV)*, 2007.

[35] E. Filiol, G. Jacob, and M. L. Liard, "Evaluation methodology and theoretical model for antiviral behavioural detection strategies," *Journal in Computer Virology*, vol. 3, no. 1, WTCV'06 Special Issue, G. Bonfante and J-Y. Marion Eds., pp. 23–37, 2007.

A Proof of Turing completeness

Basically, there are three means to prove that a language is Turing-complete. One of them is to exhibit a program in this language emulating a Turing Machine. We will base our demonstration on this principle. Before writing down this program, we need to define a certain number of elements in our language:

- the alphabet manipulated by the machine, simply 0 and 1 $\in const$,
- the different internal states of the machine defined as constants $S_0, ..., S_n \in const$, S_n being the acceptance state. These states will be associated for simplicity,
- the transitions will be described in three tables T_{state},T_{symbol},$T_{move} \in const$ representing respectively the next state, the symbol to write and the associated move. The current state determines the row to read whereas the current symbol determines the column,
- a variable V_{tape} is also needed to describe the working tape as well as variables storing the current state V_{state}, the current position V_{head} and the current symbol V_{symbol}. The equivalent transitional variables $V_{nextstate}$, V_{move} and $V_{nextsymbol}$ shall prove useful to store the next machine state during each computational step.

Proof.

#Machine initialization
$V_{head} := (\&(V_{tape}))$;
$V_{state} := (S_0)$;
$V_{symbol} := (0)$;
$while(V_{comparison} := (\neg(= (V_{state}, S_n))))then\{$
  #Reading symbol under the head
  $V_{symbol} := [V_{head}]$;
  #Transition operations
  $V_{position} := (\&(T_{state}))$;
  $V_{position} := (+(V_{position}, *(V_{state}, C_{rowsize})))$;
  $V_{position} := (+(V_{position}, V_{symbol}))$;
  $V_{nextstate} := ([V_{position}])$;
  $V_{position} := (\&(T_{symbol}))$;
  $V_{position} := (+(V_{position}, *(V_{state}, C_{rowsize})))$;
  $V_{position} := (+(V_{position}, V_{symbol}))$;
  $V_{nextsymbol} := ([V_{position}])$;
  $V_{position} := (\&(T_{move}))$;
  $V_{position} := (+(V_{position}, *(V_{state}, C_{rowsize})))$;
  $V_{position} := (+(V_{position}, V_{symbol}))$;
  $V_{move} := ([V_{position}])$;
  $V_{state} := (V_{nextstate})$;
  #Writing symbol on the tape
  $[V_{head}] := (V_{nextsymbol})$;
  #Moving head towards right if move is 1 towards left otherwise
  $if(V_{comparison} := (= (V_{move}, 1)))then\{$

$V_{head} := (+(V_{head}, 1));$
}*else*{
$V_{head} := (-(V_{head}, 1));$
}
}
stop; □

B Behavior survey

The table 2 sums up most of the results of our survey on the existing behaviors. This work is the continuation of the malicious behavior classification begun with the worm MyDoom in a previous paper [35]. We can see that few different behaviors actually exist whereas the techniques used to achieve them are multiple.

Replication	
V/FI	
Flip	Infect of COM and executable files
Lewor	Prepend infection of an executable file
Rile	Prepend infection of an executable file with original code relocation
Zelly	Infect adding new sections of the PE file
	or merging the program in a unique section and infection
V/EmW	
Bagle	Copy the running virus in the system directory
Chir	Copy the running virus in the system directory
	Copy in a companion file associated to a web page
Feebs	Copy the running virus in the system directory
Loveletter	Copy the running virus in the system directory as several executables
	Copy the running virus in the system directory as several web pages
	Replace every picture file or with a specific extension on the hard drive
Magistr	Infection of the last section in the executables of the Windows directory
MyDoom	Copy the running virus in the system directory
Sober	Copy the running virus in the system directory as several executables
	Copy the running virus in the system directory as several mails
Zellome	Copy the running virus in the system directory and destroy the original one
V/P2PW	
Supova	Copy the running virus in the system directory
Winur	Copy the running virus in the root directory
Propagation to other systems	
V/FI	
Lewor	Copy on removable devices
	Copy on connected network drives
V/EmW	
Bagle	Massmailing with the virus as attached file
Chir	Massmailing with the virus as attached file
	Copy on connected network drives
Feebs	Massmailing with the virus as attached file
	Copy in directories whose name evoked shared folders through P2P
Loveletter	Massmailing with the virus as attached file
	Using IRC channels
Magistr	Massmailing with the virus as attached file
MyDoom	Massmailing with the virus as attached file
	Copy in the KaZaA default shared directory

Table 2: Identified behaviors during the survey (First part). Abbreviation: Virus (V), File Infector (FI), Worm (W), E-mail Worm (EmW), Peer-to-Peer Worm (P2PW), Trojan (T), Rootkit (R)

Sober	Massmailing with the virus as attached file
V/P2PW	
Supova	Copy in the Windows media folder and share it by configuring KaZaA
	Automatic sending to the MSN Messenger contact list
Winur	Copy in a new hidden directory and configure certain P2P clients to share it
	Copy on a floppy disk if present
W	
Slammer	Transmission by UDP packets with a fixed port to a random IP address
CodeRed	Transmission by TCP/IP packets on port 80
Polymorphism and metamorphism	
V/FI	
Zelly	Ciphering of the virus body according to a random quadratic function
	Mutation of the decryptor by random combination of arithmetic expressions
Magistr	Ciphering the injected code by simple XOR with a shifting key value
Metaphor	Ciphering using the pseudo-random index decryption
	Garbage insertion
	Substitution of equivalent instructions
	Code permutation
V/EmW	
MyDoom	Simple permutations of the strings
	Ciphering the embedded code by simple XOR with a shifting key value
Zellome	Ciphering of the virus body according to a random quadratic function
	Mutation of the decryptor by random combination of arithmetic expressions
Overinfection test	
V/EmW	
Bagle	Test the presence of a particular registry key
Magistr	Test the presence of constant values in PE file headers
MyDoom	Test the presence of a particular registry key
W	
CodeRed	Test the presence of a particular file under a precise path
Test of activity in memory	
V/EmW	
Bagle	Test the presence of a particular mutex
MyDoom	Test the presence of a particular mutex
Residency	
V/FI	
Flip	Alter the Master Boot Record and the boot sector
Lewor	Create an autorun file or modify the existing one
	Attempt to run as a remote task with NetBIOS
Zelly	Redirection of the program entry point towards the new sections
	or interception of a particular function call of the import table
V/EmW	
Bagle	Write the virus whole path and name in a Windows run registry key
Chir	Write the virus whole path and name in a Windows run registry key
	Add the necessary script to be launched by the infected webpage
Feebs	Register by the manager as a service executed when loading the system
Loveletter	Write the virus whole path and name in a Windows run registry key
Magistr	Write the virus whole path and name in a Windows run registry key
	Write the virus path in the file win.ini
	Overwrite an entry of the import table with the viral code address
MyDoom	Write the virus whole path and name in a Windows run registry key
Sober	Write the virus whole path and name in a Windows run registry key
Zellome	Write the virus whole path and name in a Windows run registry key
	Register the virus as a debug program for the Windows Taskmanager

Table 2: Identified behaviors during the survey (Middle). Abbreviation: Virus (V), File Infector (FI), Worm (W), E-mail Worm (EmW), Peer-to-Peer Worm (P2PW), Trojan (T), Rootkit (R)

V/P2PW	
Supova	Write the virus whole path in a Windows run registry key
Winur	Write the virus whole path in a Windows run registry key
T	
Puper	Write the virus path in the registry key storing the Windows Explorer policy
Proactive defence	
V/FI	
Lewor	Terminate processes with names characteristic of protection softs (AV,IDS)
V/EmW	
Bagle	Terminate processes with names characteristic of protection softs (AV,IDS)
Sober	Terminate processes with names characteristic of protection softs (AV,IDS)
V/P2PW	
Winur	Deactivate the KaZaA analysis and protections through registry keys
T	
Puper	Separated instances mutually monitoring their respective execution
Stealth and anti-analysis measures	
V/EmW	
Bagle	Redefine its own SMTP message builder
Chir	Redefine its own SMTP message builder
Feebs	Hide registry keys and files by intercepting system calls
	in the processes memory space
	Redefine its own SMTP message builder
Magistr	Execution of the original legitimate code until the hooked function is called
	Antidebugging by injection of structures for error handling
MyDoom	Redefining its own DNS cache
	Redefining its own SMTP message builder
R	
Vanti	Hide processes and files by intercepting system calls from the SSDT
T	
Puper	Hide registry keys by intercepting system calls

Table 2: Identified behaviors during the survey (Last part). Abbreviation: Virus (V), File Infector (FI), Worm (W), E-mail Worm (EmW), Peer-to-Peer Worm (P2PW), Trojan (T), Rootkit (R)