J Comput Virol (2008) 4:235–250
DOI 10.1007/s11416-008-0085-1

ORIGINAL PAPER

Malware as interaction machines: a new framework for behavior modelling

Grégoire Jacob · Eric Filiol · Hervé Debar

Received: 15 June 2007 / Revised: 17 November 2007 / Accepted: 2 February 2008 / Published online: 21 February 2008

Abstract Several semantic-based malware analyzers have recently been put forward, each one defining its own model to capture the code behavior. All these semantic models, and abstract virology models likewise, fundamentally rely on formalisms equivalent to Turing Machines. However, as stated by recent advances in computer theory, these same formalisms do not capture appropriately interactions and concurrency. Unfortunately, malware, adaptable and resilient by essence, are likely to use these mechanisms intensively. In this paper, we thus extend the malware models to the specifically designed Interaction Machines. We first introduce two formal definitions for the interactive and the distributed viruses. According to different classes of interactions, their detection complexity is strongly impacted. Based on interactive languages, we then design an operational framework to describe malicious behaviors. Descriptions for some representative behaviors are given to complete and assess this framework.

1 Introduction

This article relies on a very simple observation. A survey we draw up over the techniques of behavioral detection, revealed a multitude of systems, each one redefining its own behavior model, in particular semantic-based ones. The underlying idea was then to provide a reference language to express malicious behaviors. In a first place, Turing complete languages seemed an adequate starting point, since most of abstract virology models rely on Turing Machine equivalent formalisms. Yet, it appeared gradually that some dynamic aspects, such as interactivity, were fundamentally missing to apprehend certain recent malicious trends. This article tries to explore new directions in computer virology to cope with the problem.

First, we recall briefly the known lacks of Turing Machines and equivalent models (Sect. 2). Next, we introduce the extended model chosen as solution: Interaction Machines. According to this model, we provide new definitions and complexity results (Sect. 3). The remaining of the paper is more operational and provides a modelling framework keeping interactions as leading thread (Sect. 4). Application cases are finally considered in order to assess the relevance of the model (Sect. 5).

G. Jacob (✉) · H. Debar
France Télécom R&D, Caen, France
e-mail: gregoire.jacob@orange-ftgroup.com

H. Debar
e-mail: herve.debar@orange-ftgroup.com

G. Jacob · E. Filiol
French Army Signals Academy,
Virology and Cryptology Lab., Rennes, France
e-mail: eric.filiol@esat.terre.defense.gouv.fr

2 Shortcomings of the Turing Machine models

2.1 Existing models in abstract virology

Surprisingly, very few formal models have actually been published in abstract virology. Since the release of the original concepts in the eighties, about only ten publications can be listed. As we are going to base our discourse on these models, it may be interesting to remind the most significant ones briefly. For those who would like to delve deeper into the subject, references are given for each model, otherwise a detailed survey is given in [1].

Based on self replicating cellular automata introduced by 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 tape.

Cohen's thesis supervisor, L. Adleman came up two years later with a more abstract formalization [4]. 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. He defined a virus as a function associating to each program, an infected form exhibiting 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.

Years later, Z. Zuo and M. Zhou extended the formalization to introduce the mutation process and additional aspects such as residency or stealth [5].

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

(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) $(\forall p \in N)\phi_{v(p)}(x) = \phi_p(x)$.

Recently, G. Bonfante, M. Kaczmarek and J.-Y. Marion have provided a last formalism based on the existence of fixed points which not only matches up with the previous models but also offers a greater flexibility [6]. As a direct consequence of Kleene's recursion theorem, a virus is built as the solution of a fixed point equation. The virus is no longer considered 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 which, 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, according to the Church-Turing thesis, the three previous models finally rely on common foundations: computability, the recursion theorem of Kleene and self-reproduction theory.

2.2 Known limitations

Basically, current abstract models captures effectively duplication, propagation and mutation concepts, but most importantly they provide fundamental results on the detection complexity. However, P. Wegner rightly underlines the fact that, if Turing Machines remain sufficient to model closed system wholly determined by their input, they fail to model open systems [7]. Extending the formalism of replicating virus to more complex malware will eventually fail because of missing dynamic concepts:

Interactions: Dynamic interactions with the external world, seen as ways to import and export data, are missing in simple Turing Machines. As mentioned in E. Filiol's recent paper on k-ary malware [8,9], even k-Turing Machines using multiple tapes cannot wholly apprehend interactions since they are limited by a quadratic enhancement in the complexity of the computed algorithm. As a matter of fact, the set of possible interaction histories is undecidable because it cannot be diagonalized (see Sect. 4.3). Unfortunately, interactions are crucial to model malware since they can perform tasks that are entirely determined by stimuli or observations of their environment (emulation detection, triggering through user actions, random execution).

Concurrency: The limitations appearing for interactions obviously impact concurrency likewise. According to R. Milner [10], Turing Machines, and generally sequential models, are no longer sufficient to model concurrent processes. Moreover Z. Manna and A. Pnueli have shown that non-terminating reactive processes, such as operating systems, cannot be captured either [11]. This could be a major drawback, since malware are highly adaptable by nature. They often use the system in a complex way in order to make its facilities work to their own benefit (file system, mail or P2P clients). In parallel, concurrency can also be seen in the scope of the malware itself: its code can be distributed over several components just as E. Filiol stated in its paper on k-ary malware [8,9].

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 account. F. Leitold first has introduced a new mathematical formalism based on Random Access Stored Program Machines with Attached Background Storages [12]. These storage facilities are additional bands with concurrent access in reading and writing modes, shared by all the processes. He has proved that this type of machine could capture communicating processes and operating systems. Unfortunately, only the interactions constrained by the band access are considered and most non-deterministic behaviors are simply ignored since the executed program is fixed. In parallel, M. Webster has introduced another model based on Distributed Abstract State Machines to capture the virus's environment but only few details are 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. To sum up our contribution:

- Interactions are divided into several classes according to the nature of the considered adversary and the communication channel.
- Formal definitions are introduced for recent malicious strains. These definitions have led to new detection complexity results.
- An operational framework is put forward to model behaviors. Its coverage is assessed in terms of soundness and completeness.
- Several existing behaviors are described within the framework, to give hints of its expressive power.

3 Interaction machine based models

3.1 Theory of interactive machines

The shortcomings of the Turing model are not really new, even A. Turing himself was aware 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, synchronous or asynchronous communications, and differences 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/or infinite input [15]. Leaving aside the infinite input, an Oracle Machine also denoted O-Machine has one or several oracles represented as immediate responses stored on additional bands [16]. The main interest is that an oracle can hypothetically solve problems of any complexity class, even undecidable. In addition, an unbounded input streams would have been required to model reactive processes by infinite computation [17]. However, we would have lost any possibility to use the computability foundations on which we base our results. As a consequence, if the operating system functioning can be hidden behind oracles, on the other hand, reactive viruses (resident in memory) cannot be modelled.

Definition 5 An Oracle Machine is a Turing machine connected to an oracle Θ through an additional tape. The Turing machine can write on this new tape an input for the oracle and then signal its request thanks to a particular state $q_?$. In a single step, the oracle computes its function in a black box way, writes its output to the tape and signal the result is ready by a second state q_r.

In Interaction Machines, the executed program is placed into an open environment with possible adversaries. An adversary is basically any object (concurrent process, operating system, network, hardware with computing facilities…) able to interact with the given program. With regards to Interaction Machines, the behavior of any concurrent adversary can be modelled as follows. According to the adversary's internal mechanism, the result of an interaction between an object O and an adversary A: I_O^A, is function of three main factors: the transmitted data, the interaction history (string made by concatenation of the data previously sent and received) and time. No assumption is made about the nature of the exchanged data. These data may be seen as simple values, implying that I_O^A is a simple function. But considering transmission of executable code for example, these data can also be functions, and I_O^A will consequently have a higher order.

$I_O^A(data\ transmitted, time, interaction\ history) = data\ received$

The oracle is used to simplify the computation acting as a black box: the time and dynamic aspects hard to capture in Turing Machines are hidden behind the oracle. Basically, the argument taken by the oracle represents the data sent by an object O to trigger the interaction with an adversary A, whereas the result represents the data returned.

$\Theta_O^A(data\ transmitted) = data\ received$

In case of unilateral interactions, either the input or the output can be null. To simplify the notations in the coming definitions, we denote the string describing the whole interaction history between the object O and an adversary A: $\Theta_O^A()$.

3.2 Abstract models for new classes of viruses

Based on a formalism equivalent to O-machines, we 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. using the Definition 3 [6]. Their definition is extended to the concept of interactive virus. 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 the results of these interactions as parameters.

Definition 6 Let $C_1, \dots, C_k$ be k semi-computable disjoint subsets of a computation domain D, $\Theta^1, \dots, \Theta^n$ be the n oracles associated to n interactive adversaries and $V_{1,1}, \dots, V_{n,k}$ be a set of semi-computable functions. An interactive virus v is defined such that, for all p and x:

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

Proposition 1 *An interactive virus v satisfying the Definition 6 exists.*

Proof The proof is similar to the one developed for the implicit virus by G. Bonfante et al. [6], except that it relies on relativized computability [18]. Let us consider the viral set V according to Definition 3 and a set of possible adversaries A. We can define on $V \times A$ a function f as follows: $f(v,a) = \Theta^a_v()$ if the oracle $\Theta^a_v()$ is defined and $f(v,a)\uparrow$[1] otherwise.

The set I of known interaction schemes is built as: $I = \{\Theta^a_v() | v \in V, a \in A, \Theta^a_v()\downarrow\}$. f is said I-semi-computable because f becomes semi-computable as soon as we can compute elements of I.

Let us consider now the case of an interactive virus with a single adversary a (the result can be extended easily to n adversaries). Let us now define two functions F' and F such as:

$$F'(y,p,i,x) = \begin{cases} V_1(y,p,x,i)) & \text{if} \quad (p,x,i) \in C_1 \\ \dots \\ V_k(y,p,x,i) & \text{if} \quad (p,x,i) \in C_k. \end{cases}$$

$F(y,p,x) = F'(y,p,f(y,a),x)$ if $f(y,a)\downarrow$, otherwise $F(y,p,x)\uparrow$.

F being I-semi-computable, by application of the relativized recursion theorem, we obtain a program v satisfying $\varphi^I_v(p,x) = F(v,p,x)$. Let e be a program computing F, e' a program computing F' and consider $\beta_1(v,p) = S(e,v,p)$, $\beta_2(v,p,f(v)) = S(e',v,p,f(v,a))$ where S is the specialization function.

$$\begin{aligned} \varphi^I_{\beta_1(v,p)}(x) &= \varphi^I_{S(e,v,p)}(x) \\ &= \varphi^I_e(v,p,x) \text{ by the relativized s-m-n theorem} \\ &= F(v,p,x) \\ &= \varphi^I_v(p,x). \end{aligned}$$

Similarly:

$$\begin{aligned} \varphi^I_{\beta_2(v,p,f(v,a))}(x) &= \varphi^I_{S(e',v,p,f(v))}(x) \\ &= \varphi^I_{e'}(v,p,f(v),x) \text{ by the relativized s-m-n theorem} \\ &= F'(v,p,f(v),x) \\ &= F(v,p,x) \\ &= \varphi^I_v(p,x). \end{aligned}$$

This second construction proves that **the result of the interaction can also be used as a parameter in the propagation function**. □

Example 1 The contradictory virus was introduced by Cohen to illustrate the detection undecidability [19]. 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 where the conditions C_k symbolize the different types of supported requests (DDos, Spam relay, Remote execution). Let us consider a remote command channel r represented by the oracle Θ^r. The oracle result is a couple of the form (c,p) where c is the request type and p the additional parameters (each component can be accessed separately using the projections π_1 and π_2). A definition for a botnet could be:

$$\varphi_v(p,x) = \begin{cases} \varphi_{\beta(v,p)}(x) & \text{if } \pi_1(\Theta^r_v()) \in install \\ \varphi_q(p,x) & \text{if } \pi_1(\Theta^r_v()) \in exec \\ & \text{with } q = \pi_2(\Theta^r_v()) \\ \varphi_{mailer}(m) & \text{if } \pi_1(\Theta^r_v()) \in relay \\ & \text{with } m = \pi_2(\Theta^r_v()) \\ <\varphi_{connect}(t), \dots, & \\ \quad \varphi_{connect}(t)> & \text{if } \pi_1(\Theta^r_v()) \in denial \\ & \text{with } t = \pi_2(\Theta^r_v()) \end{cases} \quad (1)$$

Following the same formalism, we 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]. To put things simply, we can consider distributivity over two processes as the decomposition $Behavior(P|Q) = Behavior(P) + Behavior(Q) + Interaction(P,Q)$. For the purpose of this definition, we introduce a new notation $\varphi_{p|q}$ that refers to the parallel computation of two programs p and q.

Definition 7 Let $\Theta^w_v()$ and $\Theta^v_w()$ be the oracles reflecting the interactions of two programs v and w. v and w are components of a distributed virus $v|w$ if there is a combination function f, semi computable, such as:

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

[1] According to the standard notation: f is defined at x will be denoted $f(x)\downarrow$ whereas f is undefined (divergent) at x will be denoted $f(x)\uparrow$.

Proposition 2 *Components v and w for a distributed virus satisfying the Definition 7 exist.*

Proof The proof is almost identical to the previous one, using relativized computability. By a similar reasoning, it can be proven that a propagation function β exists such that $\varphi_{v|w}(p, x, y) = \varphi_{\beta(v,w,p)}(x, y)$. □

A definition of distributed malware has been given for two components. Let us now extend our formalization to n components. Before to go any further, it shall prove useful to make a parallel with E. Filiol's work on k-ary malware [8,9]. According to his definition, k-ary malware are made up of several files which can be either active (executable) or inert (only used as data repository). By convention, we denote active components v_i and inert ones d_j. As stated by E. filiol, component interactions can be seen as graphs where the vertices symbolize the components and the edges symbolize interactions between the connected extremities. In other words, if two components v_i and v_j interact, the edge (v_i, v_j) is included in the edge set of the graph denoted E_G.

In his work, he makes the distinction between two classes of k-ary malware. The first class I gathers the sequential k-ary codes whose components are executed consecutively using the results of the previous ones. By choice, we do consider it as real concurrency but rather as a simple composition denoted $v \cdot w$: $\varphi_{v \cdot w}(p, x) = \varphi_v(\varphi_w(p, x))$. On the opposite, the second class II of parallel k-ary codes dynamically interacting is typically the notion we want to capture in the following proposition.

Definition 8 Let G be an interaction graph made up of n active components v_i and m inert components d_j. We introduce $\Theta^V_{v_i}()$ (resp. $\Theta^D_{v_i}()$) which correspond to the concatenation of all the interaction histories with its connected active (resp. inert) components: $\Theta^{v_j}_{v_i}()$ where $\{v_j | (v_i, v_j) \in E_G\}$ (resp. $\Theta^{d_k}_{v_i}()$ where $\{d_k | (v_i, d_k) \in E_G\}$). The components of the graph G constitute a distributed malware if there is a semi-computable functions g satisfying the system:

$$\varphi_G(p, x) = g(\varphi_{v_1}(p, x, \Theta^V_{v_1}(), \Theta^D_{v_1}()), \ldots, \varphi_{v_n}(p, x, \Theta^V_{v_n}(), \Theta^D_{v_n}()))$$

The complexity of the combination is dramatically increasing with the number of components. A solution to simplify the approach would be to partition the original graph into biconnected subgraphs in order to pinpoint the articulation vertices. As a consequence, it should reduce the complexity of the interaction network as pictured in Fig. 1. Therefore, we would have, instead of a massive combination function, a system of $n + 1$ more simple equations, where n is the number of biconnected subgraphs, plus one for the combination of the subgraphs. The idea is to study interactions locally before to enlarge the scope.

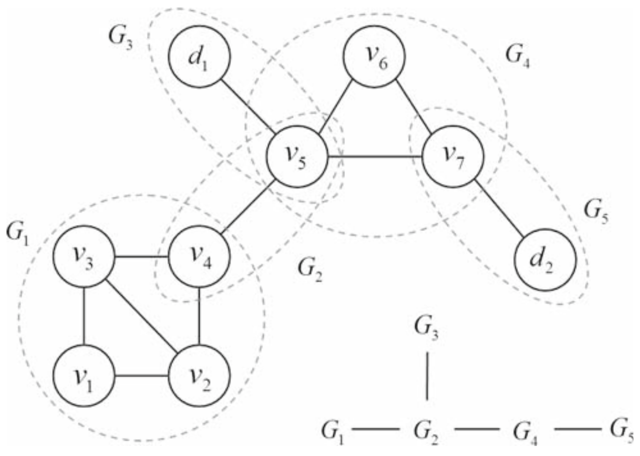

Fig. 1 Distributed virus made up of nine components. This graph of interaction is given as an example and pictures a quite complex distribution. We can see that by searching for biconnected subgraphs we can decrease the complexity of interaction to a condensed graph

3.3 Complexity of the detection problem

3.3.1 Classes of interaction and their time complexity

Interactions may be different according to the entities put into relation. By considering the different classes of interactions, we can associate a time complexity to the oracles modelling them and measure their impact on the performance of a detector.

(Class 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, 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 and thus linear.

Proposition 3 *The complexity of interactions with inert objects is in P.*

(Class 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 communication interfaces. Kernel objects such as synchronization objects are typical example. These interactions remain initiated by the observed program. Even when waiting for a remote activation signal, it cannot be achieved without an explicit request from the program.

Proposition 4 *The complexity of interactions with active objects through defined interfaces is NP-Complete.*

Proof These active objects have limited internal mechanisms able to process a given input if it complies with its interface. We can choose to describe the constraints weighting on the input as a context-free grammar (CFG). The active object can thus be described as a pushdown automaton recognizing the language described by the CFG. Let us define the automaton as the following 7-tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$:

- Q is the finite set of states associated to the different combinations of values taken by the internal attributes,
- Σ is the alphabet of input symbols corresponding to the range of values authorized by the interface signature. Likewise Γ is the stack alphabet which can basically be the same one,
- δ is the transition function modelling the internal mechanism processing the inputs,
- $q_0 \in Q$ is the initial state taken and Z_0 the stack initialization,
- $F \subset Q$ is the set of accepting states corresponding to the different possible return values that will be sent back including errors.

Determining the interaction result is equivalent to determining which accepting state is reached by the automaton. The complexity is thus equivalent to the accepting problem of a word over a language described by a CFG. This particular problem is known to be NP-complete [20]. Nevertheless, in some particular cases the grammar can be proven regular. The problem becomes P-hard since it can be solved by a deterministic finite automaton. □

Example 3 Contrary to common sense, network communications are a practical example from I_2. Even if the resulting value of the interaction seems unpredictable, partly because the remote system is out of our control, the interaction remain constrained by a protocol defining the structure of the exchanged data packets. These structures such as an IP packet for example, can finally be described by means of context-free grammars [21]. They are then interpreted by a dedicated parser using its own internal stack.

(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. An obvious case would be concurrent processes rewriting memory locations or their own code. Typically, a rootkit modify dynamically the system API addresses. Once loaded, it has repercussions on the behavior of any program using system services.

Proposition 5 *The complexity of unconstrained interactions with active objects is undecidable.*

Proof Let P be the observed program, and Q a concurrent process. P uses the value (data or instruction indifferently) stored in a memory space M without being aware that Q can modify it. M is left unmodified by Q until the end of its execution. At termination, Q writes a different value in M. Guessing which value will be used by P is equivalent to determine the termination of Q. The complexity of such interactions is thus equivalent to the halting problem which is undecidable.
Going back to our parallel with formal grammars, unconstrained interactions can relevantly be described by Turing-complete languages. □

The complexity of interactions is not only determined by their nature but also by their combination. 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 linear factor n. In the case of distributed malware, we consider multiple interactions between the adversaries. The complexity increases polynomially with the complexity of the interaction graph. The worst case is reached whenever the malware is a complete graph which cannot 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.

3.3.2 Impact of the interactions on detection

By extending the existing models with interactions, we can show that the detection complexity, previously bound by Turing Machine expressiveness, is increased by their introduction. According to Bonfante et al. the set of viruses for a given propagation function is Π_2 [6]. To introduce the new result, let us define the two following sets:

$$V_i = \{v \mid v \; is \; an \; interactive \; virus\}$$
$$V_d = \{(v, w) \mid v|w \; is \; a \; distributed \; virus\}$$

Proposition 6 *The set of interactive viruses V_i (resp. distributed viruses V_d) for a given propagation function is at least* Σ_3.

Proof Proof is given for distribution over two components but can be generalized to any arbitrary n. The proof for the set of interactive viruses is almost identical and is not detailed. Let us consider globally possible interaction schemes as a set I. From the detector perspective, detection is only possible if the set of possible interaction schemes can be explored. *We* *can thus consider the reductive hypothesis that I is computably enumerable in order to express a lower bound for the detection complexity.*

Let q be a program computing the distributed propagation function f from the definition. The set of distributed viruses over two components is then given by:

$$\exists \Theta_v^w(), \Theta_w^v()\, \forall x, y, p\, \exists y_1, \ldots, y_8 \begin{cases} [I \text{ is a computably enumerable set}] \wedge \\ [\Theta_v^w() \in I \wedge \Theta_w^v() \in I] \wedge \\ (p, x, \Theta_v^w()) = y_1 \wedge (p, y, \Theta_w^v()) = y_2 \wedge \\ (p, x, y) = y_3 \wedge \varphi_v(y_1) = y_4 \wedge \\ \varphi_w(y_2) = y_5 \wedge (y_4, y_5) = y_6 \wedge \\ \varphi_q(y_6) = y_7 \wedge \varphi_{v,w}(y_3) = y_7 \end{cases}$$

We know that $\Theta_v^w() \in I$ and $\Theta_w^v() \in I$ are Σ_1 predicates whose complexity is added to the set complexity which was originally Π_2, thus V_d is Σ_3. □

4 A formal semantic based on interactive machines for malware behaviors

The previous theoretical background justifies the importance of interactions by studying their impact on modelization and detection. Based on the Interaction Machine formalism, we want now to establish a language to model malware and more particularly malicious behaviors. The formal grammars have the advantage of providing a better understanding of the malware effects and great manipulation facilities, while remaining formal enough for a high level representation. The key idea is the migration from abstract virology to a more operational context. The purpose is similar to the recent imperative programming language introduced by Bonfante et al. in order to study the implementation of their theoretical results [22]. Unfortunately, they do not consider interactions and parallelism, which leaves some place for possible improvements.

The behavioral approach to model malware is not really new, several attempts to provide a semantic description of malicious behaviors have already been made. In 2002, Markus Schmall put forward a metalanguage which finally proved itself insufficiently formal to establish proofs about the language properties [23]. Other semantics based on simplified programming language were introduced afterwards [13], but they remained too close to the assembly level. On the opposite, recent works in the domain of intrusion detection put forward a semantic traducing the intrinsic properties of the vulnerabilities rather than the exploits themselves [24]. Our guiding principle is similar, we focus on describing the final purpose of a behavior rather than the technical solutions used to achieve it. That is why we have decided to establish a new high level dedicated formalism which can then be declined into more concrete models or instantiations by refinement.

4.1 Introduced framework

By choice, the problem was addressed from an object oriented perspective. 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 defining the Malicious Behavior Language (MBL). This grammar is wholly described on the next page. The associated operational semantics is not truly necessary to understand what follows, but the interested reader can refer to Appendix A.

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 mainly within the rules (5) and (6). These operations are then combined into blocks and structures according to the rules (10) to (14).

Proposition 7 *The MBL language is Turing-complete.*

Proof An obvious proof can be given by describing a Turing Machine in the MBL. □

Even if the proof of Turing Completeness states that our language is sound and complete with regards to internal mechanisms, Sect. 2 has shown that it remained insufficient to capture interactions. Notice that Turing Machine equivalent languages are the richest languages known to be both complete and sound.

4.3 Interaction extension

Our main improvement compared to other program grammars [25], lies in this extension. The difference with the previous approaches is that system calls and more generally interactions are treated in a generic way by semantic interpretation. Different service calls can basically have the same effect. Most of these calls can finally be reduced to simple control or input/output operations offering classification possibilities. Dealing with this equivalence is critical since the used services often betray the final purpose of the behavior.

As a matter of fact, interaction grammars require additional dynamic features. In interactive languages, some terminal grammar units may be impacted by concurrent objects. As a consequence, listening and transmitting operators are required to dynamically affect values to these units. The rules (7) to (9) define dynamic interactive commands for listening and transmitting operations. The future possible values,

```
(1)  <Attribute>    ::= var | const
(2)  <Adversary>    ::= obj_perm | obj_temp | obj_com
                      | obj_boot | obj_exec | obj_sec
                      | env_var | this
(3)  <Op1>          ::= ¬|&
(4)  <Op2>          ::= ∨| ∧ | ⊕ | < | ≤ | = | ≥ | >
                      | + | − | × | ÷ | ≡ | << | >>
(5)  <Term>         ::= <Attribute> | [<Term>]
                      | <Op1> (<Term>)
                      | <Op2> (<Term>, <Term>)
                      | <Operation> | <Interaction>
(6)  <Operation>    ::= var := (<Expression>)
                      | [<Expression>] := (<Expression>)
                      | goto <Expression>
                      | stop
(7)  <Interaction>  ::= <Control><Adversary> | <I/O>
(8)  <Control>      ::= open | create | close | delete
(9)  <I/O>          ::= receive var ←<Adversary>
                      | receive [<Expression>] ←<Adversary>
                      | send <Expression>→<Adversary>
                      | wait <Adversary>
                      | signal <Adversary>
(10) <Block>        ::= <Term>; <Block>
                      | <Term>;
(11) <Structure>    ::= <Block>
                      | if(<Expression>)then{
                            <Sequence>
                        }else{
                            <Sequence>
                        }
                      | if(<Term>)then{
                            <Sequence>
                        }
                      | while(<Term>){
                            <Sequence>
                        }
                      | [<Sequence>‖<Alternatives>]
(12) <Alternatives> ::= <Sequence>‖<Alternatives>
                      | <Sequence>
(13) <Sequence>     ::= <Structure><Sequence>
                      | <Structure>
(14) <Behavior>     ::= <Sequence>
```

taken by the variables storing the results of these interactions, are incrementally transformed into a sequential past at each computational step (see Appendix A for operational semantic). These operators prove themselves sufficient for modelling the interactions 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 distinguish synchronous and asynchronous communications. On the other hand, the interactions of class I_3 can only be modelled by non-deterministic choices requiring a third additional operator. In our grammar, the respective operator is introduced in rule (11) 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 Sect. 3.2) and thus can hardly be integrated to models in first place.

Proposition 8 *Soundness of the MBL with regards to interactions is quite intuitive considering the fact that the concept of object-oriented modelling is directly inspired from the reality. On the opposite, completeness is impossible to achieve.*

Proof Interactive systems have an inherent incompleteness [7]. Dynamically generated streams can be mathematically modelled by the set of infinite sequences which cannot be diagonalized. Similarly to the Godël incompleteness result for the integers, any domain whose set of true assertions cannot be diagonalized, cannot be complete. Moreover, the results of interactions are not necessarily strings: in case of code rewriting, the interaction can be seen as a function passing (class I_3). Nevertheless, a partial completeness can be guaranteed empirically showing that we are able to express sufficiently interactions to capture the actions of malware. □

4.4 Adversaries classification

To extend the framework, object types are defined to classify the adversaries. Using the object-oriented approach, an inheritance scheme has been developed based on the final purpose of each object. Only classes with relevant effects on the malware lifecycle have been specified in detail. This classification suggests a certain approach but remain open for discussion.

Basically, objects are separated into two classes according to their persistence as pictured in Fig. 2. The first class 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, the second class gathers the temporary objects (*obj_temp*) existing only for a given time, as long as the system remains active. Mutex or events are simple examples already used by malware writers. Particular objects inheriting of these two classes are defined more specifically. The more specific class always prevails on the generic one:

- A first permanent subclass gathers the boot objects (*obj_boot*). These objects provide the malware facilities to execute its code automatically. The run registry keys, the win.ini file for Windows, or the master boot record make execution possible during the boot sequence. Automatic execution is also possible at runtime by overwriting the global system service descriptor table, the import tables or entry points in executables. Such locations are also considered as members of the boot class.
- A second permanent subclass is made up of the environment variables (*env_var*). These objects store important information about the platform. Configuration files, registered path but also hardware fixed data structures available through particular instructions (cpuid) are just a few examples.

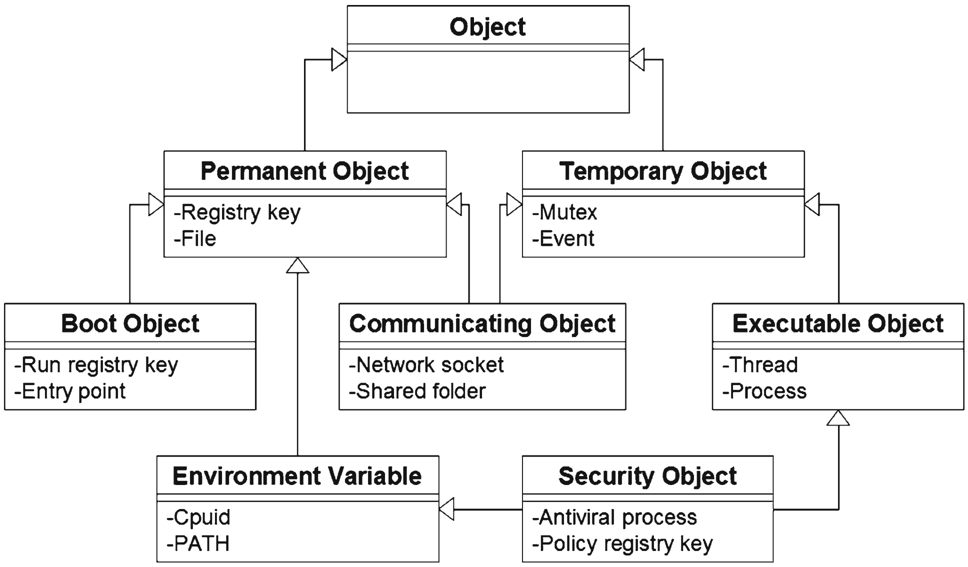

Fig. 2 Adversary inheritance scheme. Any system object is either permanent or temporary making the classification complete. They have been derived into several specific classes according to the malware perspective

- The subclass made up of the communicating object inherits from both the permanent and temporary classes (obj_com). These objects constitute communication channels to remote locations or systems. The definition of a communicating object is very large. Obviously network drives, shared directories (intranet, P2P), or removable devices are permanent members of this class whereas network connexions, or processes pipes are particular temporary members.
- 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) 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, the definition of an autoreference ($this$) shall prove itself useful 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"

As stated in part 4.3, theoretical completeness cannot be proven. To assess partial completeness, the framework has been confronted to a pool of real world cases. To do so, we have considered a pool of a dozen of representative malicious strains. Similarly to the analysis led in a previous paper [26], we have identified different techniques used to achieve several classes of typical malicious behaviors. Global information about these behaviors are partly available on observatory websites [27]. When deeper information were required, they were found in detailed analyses of malware (wild examples: Bagle [28], MyDoom [29], or significant zoo examples: Magistr [30], Zellome [31]).

Through the results of this survey, several behavior classes have been successfully described. The obtained descriptions have proven themselves generic enough to apply to all our example. Even if some differences were observed in the different instantiations of a given behavior, the generic principle remained quite similar and was successfully traduced by the descriptions we introduce in the next part.

5.2 Specific behavior definitions

Based on this survey, we 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 (see example in Appendix B for illustration).

5.2.1 Replication mechanisms

Self replication is a key mechanism with viruses or 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:

Duplication: Simple duplication requires no target to host the code. The viral code is first stored in a local buffer symbolized by the generic variable V_{code}. This code is then stored in a newly created permanent object O_{clone}. During the duplication, mutations can occur but these mechanisms shall only be described in appendix for the interested reader (see Appendix C).

```
$V_{code} \in var$
$O_{clone} \in obj\_perm$
  $(i)$   $<Duplication>$ ::= $<Creation><Reading>$
                             $<Mutation><Writing>$
                           | $<Reading><Creation>$
                             $<Mutation><Writing>$
  $(ii)$  $<Creation>$    ::= $create\ O_{clone};$
  $(iii)$ $<Reading>$     ::= $receive\ V_{code} \leftarrow this;$
  $(iv)$  $<Writing>$     ::= $send\ V_{code} \rightarrow O_{clone};$
```

Infection: Contrary to duplication, infection requires an existing entity to host the viral code. As a consequence, the first phase of the replication always consists in crawling into the system to look for a potential target. Conditions modelled by C_{valid} are defined to check the validity of the target, one of them being the absence of previous infection. An example could be the absence in the file of an infection marker, a "magic constant" for example. 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.

```
$V_{target}, V_{code}, V_{save}, V_{comparison} \in var$
$C_{valid} \in const$
$O_{target} \in obj\_perm$
  $(i)$   $<Infection>$ ::= $<Searching><Opening><Relocating>$
                          $<Reading><Mutation><Writing>$
                        | $<Searching><Opening><Reading>$
                          $<Relocating><Mutation><Writing>$
  $(ii)$  $<Searching>$ ::= $while(V_{comparison} := (\neg(= (V_{target}, C_{valid})))) \{$
                              $open\ O_{target};$
                              $receive\ V_{target} \leftarrow O_{target};$
                          $\}$
  $(iii)$ $<Opening>$    ::= $open\ O_{target};$
  $(iv)$  $<Relocating>$ ::= $receive\ V_{save} \leftarrow O_{target};$
                           $send\ V_{save} \rightarrow O_{target};$
                         | $\epsilon$
  $(v)$   $<Reading>$    ::= $receive\ V_{code} \leftarrow this;$
  $(vi)$  $<Writing>$    ::= $send\ V_{code} \rightarrow O_{target};$
```

Propagation: Propagation is a third way of replicating, 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 into 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 encoding of the attached code of the malware in a base 64 format. Notice that encoding may take several steps. Like any other replication mechanism, mutations are likely to occur.

```
$V_{code}, V_{formatted}, V_{parameter}, V_{position} \in var$
$C_{header}, C_{hsize} \in const$
$O_{channel} \in obj\_com$
  $(i)$   $<Propagation>$  ::= $<Opening><Reading>$
                              $<Mutation><Transmitting>$
                            | $<Reading><Opening>$
                              $<Mutation><Transmitting>$
  $(ii)$  $<Opening>$      ::= $open\ O_{channel};$
  $(iii)$ $<Reading>$      ::= $receive\ V_{code} \leftarrow this;$
  $(iv)$  $<Transmitting>$ ::= $send\ V_{code} \rightarrow O_{channel};$
                            | $<Formatting>$
                              $send\ V_{formatted} \rightarrow O_{channel};$
  $(v)$   $<Formatting>$   ::= $V_{position} := (\&(V_{formatted}));$
                              $[V_{position}] := (C_{header});$
                              $V_{position} := (+(V_{position}, C_{hsize}))$
                              $<Encoding>$
                              $[V_{position}] := (V_{code});$
  $(vi)$  $<Encoding>$     ::= $V_{code} := (<Op2>\ (V_{code}, V_{parameter}));$
                              $<Encoding>$
                            | $\epsilon$
```

5.2.2 *Overinfection and activity tests*

Overinfection test: The overinfection test detects if any instance of the malware is present on the system. The detection is done by checking the existence of a permanent marker O_{marker}. This test can be achieved through at least three different methods. In the case of file infection, the overinfection test for a target is already integrated in the searching routine which search for an healthy target. It does not need to be redefined here.

```
$O_{marker} \in obj\_perm$
  $(i)$   $<Overinfection>$ ::= $<Test1> \mid <Test2> \mid <Test3>$
  $(ii)$  $<Test1>$         ::= $if(create\ O_{marker})then\{$
                                   $stop;$
                               $\}$
  $(iii)$ $<Test2>$         ::= $if(open\ O_{marker})then\{$
                                   $stop;$
                               $\}else\{$
                                   $create\ O_{marker};$
                               $\}$
  $(iv)$  $<Test3>$         ::= $if(\neg open\ O_{marker})then\{$
                                   $create\ O_{marker};$
                               $\}else\{$
                                   $stop;$
                               $\}$
```

Activity test: If overinfection test addresses the static problem, the activity test deals with the dynamic aspect. The activity test detects if an instance of the malware is already running in memory. It proves itself really useful for worms whose code is never written down on the disk. The execution is betrayed by the presence of a particular temporary object O_{active}. Otherwise the structure is quite similar to the previous one.

```
$O_{active} \in obj\_temp$
  $(i)$   $<Activity> ::= <Test1> \mid <Test2> \mid <Test3>$
  $(ii)$  $<Test1>$   $::= if(create\ O_{active})then\{$
                  $stop;$
              $\}$
  $(iii)$ $<Test2>$   $::= if(open\ O_{active})then\{$
                  $stop;$
              $\}else\{$
                  $create\ O_{active};$
              $\}$
  $(iv)$ $<Test3> ::= if(\neg open\ O_{active})then\{$
                  $create\ O_{active};$
          $\}else\{$
                  $stop;$
          $\}$
```

5.2.3 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.

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

6 Conclusion and perspectives

Throughout this paper, we have introduced a new framework based on interactions 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 malware modelling since we have managed to describe most of the identified malicious behaviors. In order to achieve a greater completeness, the scope of the survey should be increased to a wider range of malware. Anyhow, the generative grammar proves itself sufficiently generic to define additional behaviors or refine existing descriptions. Additional behaviors such as data gathering or typical final payload could have been described but we had to limit ourselves not to bury 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. In fact, this semantic brings into light functional similarities more evolved than simple instruction equivalence which is the major drawback of most current detection systems. This is particularly due to the generic interpretation of interactions. Eventually, it could be worth considering integrating our framework to existing semantic analysis systems for malware detection as in [25,32,33].

The inheritance scheme for adversaries is also an interesting feature since it helps to understand the relation between a malware and its environment. Studying this classification further would help to refine the scheme and bring additional information about the data flow. Another perspective would be to explore deeper the existing interaction semantics for a more proper theoretical formalism than oracles [10].

Acknowledgements We would like to thank the reviewers, their interesting comments have helped much to improve this paper. Another special thanks in particular to Jean-Yves Marion for his valuable remarks and advices when correcting this paper.

Appendix A: MBL operational semantic

The MBL operational semantic requires important data structures to manage interactions between the different objects. Let us define a configuration of n concurrent objects. First an array $\sigma = \sigma_1 \dots \sigma_n$ is required to store the immediate results during the evaluation of the different objects executions. A second array $V = V_1 \dots V_n$ is required to store the values manipulated by these objects. At last, to model the exchanges between the objects three matrices of size $n \times n$ must be defined:

- A matrix L symbolizing the links between objects. A link exists between the ith object and the jth object if $L_{ij} = 1$.
- A matrix S symbolizing the signalization between objects. A signal is sent from the ith object to the jth object if $S_{ij} = true$.
- A matrix D of lists storing the values transmitted between the objects.

Let ϵ^i denote the evaluation function for the ith object, $[x/y]$ denote the substitution of y by x, and the operator $\cdot$ denote the concatenation of two lists.

Evaluation of interactions:

Link creation and destruction

$$\epsilon^i_{open\ o_j}(<\sigma, V, L, S, D>)$$
$$= \begin{cases} <\sigma[\sigma_i/1], V, L[L_{ij}/1], S, D> & \text{if } L_{ij} = 0 \\ <\sigma[\sigma_i/0], V, L, S, D> & \text{otherwise} \end{cases}$$

$$\epsilon^i_{close\ o_j}(<\sigma, V, L, S, D>)$$
$$= \begin{cases} <\sigma[\sigma_i/1], V, L[L_{ij}/0], S, D> & \text{if } L_{ij} = 1 \\ <\sigma[\sigma_i/0], V, L, S, D> & \text{otherwise} \end{cases}$$

Signalization

$$\epsilon^i_{signal\ o_j}(<\sigma, V, L, S, D>)$$
$$= \begin{cases} <\sigma[\sigma_i/1], V, L, S[S_{ij}/true], D> & \text{if } L_{ij} = 1 \\ <\sigma[\sigma_i/0], V, L, S, D> & \text{otherwise} \end{cases}$$

$$\epsilon^i_{wait\ o_j}(<\sigma, V, L, S, D>)$$
$$= \begin{cases} <\sigma[\sigma_i/1], V, L, S[S_{ji}/false], D> & \text{if } L_{ij} = 1 \text{ and } S_{ji} = true \\ \epsilon^i_{wait\ o_j}(<\sigma, V, L, S, D>) & \text{otherwise} \end{cases}$$

Data transmission and reception

$$\epsilon^i_{send\ v \to o_j}(<\sigma, V, L, S, D>)$$
$$= \begin{cases} <\sigma[\sigma_i/1], V, L, S, D[D_{ij}/D_{ij} \cdot [v]> & \text{if } L_{ij} = 1 \\ <\sigma[\sigma_i/0], V, L, S, D> & \text{otherwise} \end{cases}$$

$$\epsilon^i_{receive\ v \leftarrow o_j}(<\sigma, V, L, S, D>)$$
$$= \begin{cases} <\sigma[\sigma_i/1], V[v/v'], L, S, D[D_{ji}/T]> & \text{if } L_{ij}{=}1 \text{ and } D_{ji} = [v] \cdot T \\ <\sigma[\sigma_i/0], V, L, S, D> & \text{otherwise} \end{cases}$$

The evaluation of the other expressions from the grammar is quite similar to any other programming language and shall not be described here. The reader can refer to the semantic described by Bonfante et al. to have a complementary overview (Appendix [22]).

Appendix B: From instanciation to semantic description

This appendix gives more details about our survey of existing behaviors. As said in Sect. 5.1, we have deployed our analysis on several malware strains from different types: File Infectors (Flip, Lewor, Magistr, Metaphor, Rile, Zelly), Worms (Slammer, CodeRed), E-mail Worms (Bagle, Chir, Feebs, LoveLetter, MyDoom, Sober, Zellome), Peer-to-peer Worms (Supova, Winur), Trojans (Puper), Rootkits (Vanti). A list of characteristic behaviours has been established which is detailed through the different description from Sect. 5.2 and the next appendix.

Let us now focus on the translation approach that has been used during the different analyses. How can some instruction blocks be interpreted into grammar units. Let us consider a practical example with the propagation function which is quite complete. The propagation behavior can be implemented through different technical solutions which remain nonetheless quite similar as the table below underlines it.

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
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 known 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

To move still closer to intantiation, let us chose a specific example with the e-mail worm MyDoom (already analyzed in [26]) and illustrate the translation with quotes from its source code. Obviously, the same explanation stands for the other samples. Let us remind briefly the generative rule of the description:

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

$$\begin{aligned}(i)\ <Propagation> ::= & <Opening><Reading> \\ & <Mutation><Transmitting> \\ | & <Reading><Opening> \\ & <Mutation><Transmitting>\end{aligned}$$

This rule does not contain any final unit, let us move forward to the opening rule:

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

```
/* Open socket */
struct hostent *h = gethostbyname(hostname);
struct sockaddr_in addr = *(h->h_addr_list[0]);
sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
connect(sock, addr, sizeof(struct sockaddr_in));
```

A simple call to the socket function is sometimes insufficient. Combined with the connect call, we can interpret precious information about the kind of socket created. Anyhow, a socket is basically a communicating object. Similarly, if the communicating object had been a shared directory, an interpretation would have been required on the path given as parameters to the CreateFile in order to recognize a P2P associated folder.

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

```
/* Open currently executing file */
GetModuleFileName(NULL, selfpath, MAX_PATH);
HANDLE hFile = CreateFile(selfpath, GENERIC_READ,
   FILE_SHARE_READ|FILE_SHARE_WRITE, NULL,
   OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
/* Reading file content in buffer */
DWORD dwSize = GetFileSize(hFile, &dwUp);
ReadFile(hFile, pBufferCode, dwSize, &dwRead, NULL);
```

Once again, the call to GetModuleHandle allow us to analyze the parameters passed to CreateFile. This preliminary call makes the distinction between a simple permanent object opening and the access to the auto-reference.

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

```
/* Sending information */
send(sock, pBufferCode, lstrlen(pBufferCode), 0);
```

This translation is quite trivial. Nevertheless in our case, a formatting preliminary phase is required since the propagation is done by e-mail. It would not have been the case for example in basic worms like Slammer who send their code like raw data.

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

```
/* Concatenate header */
char header[] = ''From: myadresse@domaine.ext\r\n
   To: target adresse@domaine.ext\r\n
   Subject mail subject\r\nDate\r\n
   MIME-Version\r\nContent-Type: multipart/mixed\r\n'';
lstrcat(pFormatted, header);
/* Optional encoding */
...
/* Concatenate code */
lstrcat(pFormatted, pBufferCode);
```

The concatenated header is a constant which is directly determined by the exchange protocol used by the communicating object. In our particular case, the constant is a SMTP header predefined in the character table. In addition to the header, an encoding step is required in MyDoom because attached files must be encoded in a base 64 to comply with the SMTP protocol.

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

```
/* Base 64 table */
BYTE alpha[] = ''ABCDEFGHIJKLMNOPQRSTUVWXYZ
   abcdefghijklmnopqrstuvwxyz0123456789+/'';
/* Base 64 encoding */
q[0] = alpha[t[0] >> 2];
q[1] = alpha[((t[0] & 03) << 4) | (t[1] >> 4)];
q[2] = alpha[((t[1] & 017) << 2) | (t[2] >> 6)];
q[3] = alpha[t[2] & 077];
```

Appendix C: Additional behavior descriptions

In this appendix, we have chosen to detail some additional behaviors which are not necessarily relevant to our main focus: interactions. Nevertheless, they constitute a proof of the expressiveness of the semantic approaches.

Mutation mechanisms

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

(i) $<Mutation>$ ::= $<Polymorphism><Metamorphism>$
| $<Polymorphism>$
| $<Metamorphism>$
| ϵ

Polymorphism: Polymorphism is historically the first type of engine and thus the simpler. In polymorphism, the plain code is encrypted during copy. As a matter of fact most of the actual encryption 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 [25]. In particular, key variation and chaining like CBC have been encountered in some of the analyzed malware and have thus been added. If this model only describes simple encryption algorithms, it can be extended to more complex ones as our grammar is Turing complete. It is also possible to define a particular progression during the process. Certain algorithms such as in PRIDE (Pseudo-Random Index DEcryption [34]) have complex or even random memory accesses instead of sequential ones in order to delude emulators.

$V_{code}, V_{position}, V_{temp}, V_{key}, V_{posprev}, V_{previous}, V_{comparison} \in var$
$C_{limit}, C_{variation}, C_{progress} \in const$

```
(i)   <Polymorphism> ::= $V_{position} := (\&(V_{code}));$
                         $while(V_{comparison} := (< (V_{position}, C_{limit}))) \{$
                             <Ciphering>
                             <KeyVariation>
                             <Next>
                         }
(ii)  <Ciphering>    ::= $V_{temp} := ([V_{position}]);$
                         <Chaining>
                         $V_{temp} := (<Op2> (V_{temp}, V_{key}));$
                         $[V_{position}] := (V_{temp});$
(iii) <Chaining>     ::= $V_{posprev} := (-(V_{position}, 1));$
                         $V_{previous} := ([V_{posprev}]);$
                         $V_{temp} := (<Op2> (V_{temp}, V_{previous}));$
                       | $\epsilon$
(iv)  <KeyVariation> ::= $V_{key} := (<Op2> (V_{key}, C_{variation}));$
                       | $\epsilon$
(v)   <Next>         ::= $V_{position} := (<Op2> (V_{position}, C_{progress}));$
```

The associated decrypting routine structure is almost identical to the mutation process as encryption and decryption 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) $<DecryptRoutine> ::= <Polymorphism>$ $goto\ V_{position};$

Metamorphism: 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 [9,35]. We 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 formalization, the use of different system services with varying parameters can be reduced to their basic interpretation as interactions bringing equivalences into light.

The first technique is garbage insertion. Existing works already define the insertion of dead code as a grammar production rule [36]. This model considers only the insertion of nop equivalent instructions. In our model, we 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 $s_1, \ldots, s_n$ 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.

$s_1 \ldots s_n \Rightarrow_R <Garbage>\ s_1\ <Garbage> \cdots <Garbage>\ s_n\ <Garbage>$
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 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 $s_i \ldots s_j$. Jump are then introduced in order to maintain the correct control flow.

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

As we are working at a semantic level, the problem of register reassignement is already addressed using generic variables. But we 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 malware whose form-based detection is undecidable [9,35].

Anti-antiviral mechanisms

Proactive defense: 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) $<Proactive> ::= delete\ O_{protect};$

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) $<Policy> ::= open\ O_{policy};$
$\qquad send\ V_{weak} \rightarrow O_{policy};$

These two techniques are quite aggressive and they are toughly monitored by antivirus and HIPS. There are more subtle ways to avoid detection, such as preventing the capture of any information betraying the malicious activity. In order to analyze malware, 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 [37]. In case of detection, the malware can execute a legitimate sequence or simply stop.

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

Stealth: A malware 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. Most of these environment structures are accessed thanks to services, so the references to the malware should be deleted at this level. In order to achieve this, we define ways for a malware to be stealthy relatively to services and in particular 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 like in the preprocessing case. Or the function returns data likely to contain this reference like in the postprocessing case. In this last case, the data sent back can be an explicit value but also an address toward a complex structure requiring analysis through a loop.

```
$V_{comparison}, V_{parameter}, V_{return}, V_{benign}, V_{position}, V_{size}, V_{value} \in var$
$C_{this}, C_{progress} \in const$
  $(i)$   $<StealthFuntion>$ $::=$ $<Preprocessing>$
                                 $<SysCall>$
                                 $<Postprocessing>$
  $(ii)$  $<Preprocessing>$  $::=$ $if(V_{comparison} := (= (V_{parameter}, C_{this})))then\{$
                                     $V_{parameter} := (V_{benign})$;
                                 $\}$
                               $|\ \epsilon$
  $(iii)$ $<Postprocessing>$ $::=$ $if(V_{comparison} := (= (V_{return}, C_{this})))then\{$
                                     $V_{return} := (V_{benign})$;
                                 $\}$
                               $|\ V_{position} := (V_{return})$;
                                 $while(V_{comparison} := (< (V_{position}, V_{limit}))) \{$
                                     $V_{value} := ([V_{position}])$;
                                     $if(V_{comparison} := (= (V_{value}, C_{this})))then\{$
                                         $[V_{position}] := (V_{benign})$;
                                     $\}$
                                     $V_{position} := (<Op2> (V_{position}, C_{progress}))$
                                 $\}$
                               $|\ \epsilon$
```

References

1. Filiol, E.: Computer Viruses: From Theory to Applications. Springer, Berlin, IRIS Collection, ISBN:2-287-23939-1 (2005)
2. von Neumann, J.: Theory of Self-Reproducing Automata. University of Illinois Press, ISBN:0-598-37798-0 (1966)
3. Cohen, F.: Computer Viruses. Ph.D. Thesis, University of South California (1986)
4. Adleman, L.M.: An abstract theory of computer viruses. In: CRYPTO '88: Proceedings on Advances in cryptology, pp. 354–374 (1990)
5. Zuo, Z., Zhou, M.: Some further theoretical results about computer viruses. Comput. J. **47**(6), 627–633 (2004)
6. Bonfante, G., Kaczmarek, M., Marion, J.-Y.: On abstract computer virology from a recursion-theoretic perspective. J. Comput. Virol. **1**(3–4), 45–54 (2006)
7. Wegner, P.: Why interaction is more powerful than algorithms. Commun. ACM **40**(5), 80–91 (1997)
8. Filiol, E.: Formalisation and implementation aspects of k-ary (malicious) codes. J. Comput. Virol., vol. 3, no. 3, EICAR 2007 Special Issue. Broucek, V., Turner, P. (eds) (2007)
9. Filiol, E.: Techniques Virales avancTes. Springer, Berlin, IRIS Collection, ISBN:2-287-33887-8 (2007)
10. Milner, R.: Elements of interaction: Turing award lecture. Commun. ACM **36**(1), 78–89 (1993)
11. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, New York, ISBN:0-387-97664-7 (1992)
12. Leitold, F.: Mathematical model of computer viruses. In: Best Paper Proceedings of EICAR, pp. 194–217 (2000)
13. Webster, M.: 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, Mosses, P., Power, J., Seisenberger, M. (eds) pp. 99–113 (2005)
14. Wegner, P.: Interactive foundations of computing. Theor. Comput. Sci. **192**(2), 315–351 (1998)
15. Wegner, P.: Interaction as a basis for empirical computer science. ACM Comput. Surv. **27**(1), 45–48 (1995)
16. Atallah, M.J.: Algorithms and Theory of Computation Handbook. CRC Press LLC, West Palm Beach, FL (2000)
17. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer. Elsevier, Amsterdam (1990)
18. Rogers, H.: Theory of Recursive Functions and Effective Computability. MIT Press, Cambridge, MA, ISBN:0-262-68052-1 (1987)
19. Cohen, F.B.: Computer viruses: theory and experiments. Comput. Secur. **6**(1), 22–35 (1987)
20. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages and Computation, 2nd edn. Addison-Wesley, Reading, MA, ISBN:0-201-44124-1 (1995)
21. Manual reference pages—ipsend. http://www.gsp.com/cgi-bin/man.cgi?section=5&topic=ipsend
22. Bonfante, G., Kaczmarek, M., Marion, J.-Y.: A classification of viruses through recursion theorems In: Computation and Logic in

the Real World, CIE'07, vol. 4497 of Lecture Notes in Computer Science, pp. 73–82. Springer, Berlin (2007)
23. Schmall, M.: Classification and Identification of Malicious Code Based on Heuristic Techniques Utilizing Meta-languages. Ph.D. Thesis, University of Hamburg (2002)
24. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 2–16 (2006)
25. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantic-aware malware detection. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 32–46 (2005)
26. Filiol, E., Jacob, G., Liard, M.L.: Evaluation methodology and theoretical model for antiviral behavioural detection strategies. J. Comput. Virol., vol. 3, no. 1, WTCV'06 Special Issue, G. Bonfante and J-Y. Marion Eds., pp. 23–37 (2007)
27. Fortinet observatory. http://www.fortinet.com/FortiGuardCenter/
28. Rozinov, K.: 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)
29. Filiol, E.: Le ver mydoom. MISC—Le magazine de la sTcuritT informatique, vol. 13 (2004)
30. Ferrie, P.: Magisterium abraxas. In: Proceedings of Virus Bulletin, pp. 6–7 (2001)
31. Ferrie, P., Shannon, H.: It's zell(d)ome the one you expect—w32/zellome. In: Proceedings of Virus Bulletin, pp. 7–11 (2005)
32. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. Lect. Notes Comput. Sci. **3548**, 74–187 (2005)
33. Shin, J., Spears, D.: The Basic Building Blocks of Malware. Technical Report, University of Wyoming (2006)
34. Driller, T.M.: Advanced polymorphic engine construction. 29A E-zine, vol. 5 (2003)
35. Filiol, E.: Metamorphism, formal grammars and undecidable code mutation. In: Proceedings of the International Conference on Computational Intelligence (ICCI), Published in the International Journal in Computer Science, vol. 2, issue 1, pp. 70–75 (2007)
36. Qozah, Polymorphism and grammars, 29A E-zine, vol. 4 (1999)
37. Rutkowska, J.: Red pill…or how to detect vmm using (almost) one cpu instruction (2005). http://invisiblethings.org/papers/redpill.html