Formalization of viruses and malware through process algebras

Grégoire Jacob[(1/2)], Eric Filiol[(1)], Hervé Debar[(2)]
[1] ESIEA, Operational Cryptology and Virology Lab., Laval, France
filiol@esiea-ouest.fr
[2] Orange Labs, MAPS/STT, Caen, France
{gregoire.jacob|herve.debar}@orange-ftgroup.com

***Abstract*—Abstract virology has seen the apparition of successive viral models, all based on Turing-equivalent formalisms. Considering recent malware (rootkits, k-ary codes), these threats are only partially covered because functional formalisms do not support interactive computations. New models have thus appeared to support these evolutions, but loosing the unified approach in the way. This article provides a basis for a unified malware model, founded on the Join-Calculus. In terms of expressiveness, the process-based model supports the fundamental definitions based on self-replication and adds support for interactions, concurrency and non-termination to cover evolved malware. In terms of detection and prevention, detection undecidability and prevention by isolation still hold. Additional results are also established: identification of calculus fragments where detection becomes decidable, formal definition of the non-infection property, potential solutions to restrict malware propagation.**

***Index Terms*—Malware, Process Algebra, Detection, Prevention**

I. Introduction and related works

Considering malware, interactions with the execution environment, concurrency and non-termination are important functionalities [1]. In effect, resilient and adaptive by nature, malware intensively use these to survive and infect systems. Referring to abstract virology, existing models focus on the self-replication capacity which is defined in functional terms [2], [3], [4]. Unfortunately, they rely on Turing-equivalent formalisms [5] which hardly support interactive computations. With the apparition of interaction-based viral techniques, new models have been introduced to cope with this limitation, but loosing any unified approach in the way. K-ary malware introduce concurrency with a distribution of the malicious code over several executing parts. [6] provides a model based on Boolean functions to capture their evolving interdependence. Rootkits introduce stealth techniques requiring a reactive and non-terminating execution. Different models have been provided based either on steganography [7] or graph theory [8].

By evolving towards interaction-dedicated formalisms such as process algebras, a unified model for malware can be defined to support these innovative techniques [1]. To maintain the expressiveness of former models, the chosen algebra has to support both functional and interactive aspects. The Join-Calculus was found adequate for building the model [9], [10]. The model offers a greater expressiveness while being closer to the current vision of computer systems. Still, it provides reasoning and proof facilities because it relies on an established theoretical formalism. Process algebras also increase the visibility over computation locations and information flows. Consequently, the identification of potential detection methods and control points become proportionally eased. The article contribution can be summed-up to the following points:

- A process-based viral model superseding functional ones by support of interactive computations.
- A parametrization of the model to cover evolved malware.
- A verification of the formalism impact on fundamental results for detection and prevention.

The article is articulated as follows. Section II presents the Join-Calculus. Section III presents self-replication inside functional models. Section IV introduces the process-based model, allowing distributed self-replication. Section V extends it with a parametrization for Rootkits. Within the model, Sections VI and VII addresses the decidability of malware detection as well as solutions to prevent their propagation.

II. Introducing the Join-Calculus

This overview guarantees minimal self-containment but the reader is invited to refer to the relative literature [9], [10]. At the basis of the Join-Calculus, an infinite set of names $x, y, z...$ is defined. Names are compound into vectors $\overrightarrow{x}$ equivalent to $x_0, ..., x_n$. Names constitute the basic blocks for message emissions of the form $x{<}v{>}$ where x is called the *channel* and v the transmitted *message*. Given in Figure 1, the syntax of the Join-Calculus defines three elements to handle message passing: processes (P) being the communicating entities, definitions (D) describing the system evolution resulting of the interprocess communications, and the join-patterns (J) defining the channels and messages involved in communications [9, pp.57-60]. For ease of modeling, the support of expressions (E) has been introduced to provide the synchronous channels necessary to concurrent functional languages [9, pp.91-92]. Expressions can eventually be encoded into the minimal core of the Join-Calculus.

Based on the syntax, names are divided between different sets: 1) the channels defined through a join definition (dv), 2) the names received by a join-pattern (rv), 3) the free names (fv) and conversely bound names (bv) of a process. Their inductive construction can be found in [9, p.47]. In addition to the syntax, operational semantics are required to complete the computational model. These semantics are defined by Reflexive Chemical Abstract Machines (RCHAM), specified by the rules of Figure 2 [9, pp.56-62]. In particular, reductions

$P ::= v{<}E_1;...;E_n{>}$	asynchronous message
$\mid\ def\ D\ in\ P$	local definition
$\mid\ P \mid P$	parallel composition
$\mid\ 0$	null process
$\mid\ E; P$	sequence
$\mid\ let\ x_1, ..., x_m = E\ in\ P$	expression computation
$\mid\ return\ E_1, ..., E_n\ to\ x$	synchronous return
$E ::= v(E_1;...;E_n)$	synchronous call
$\mid\ def\ D\ in\ E$	local definition
$D ::= J \triangleright P$	reaction rule
$\mid\ D \wedge D$	conjunction
$\mid\ \top$	null definition
$J ::= x{<}y_1, ..., y_n{>}$	message pattern
$\mid\ x(y_1;...;y_n)$	call pattern
$\mid\ J \mid J$	join of patterns

Figure 1. Enriched syntax for the Join-Calculus.

STR-JOIN	$\vdash P_1 \mid P_2$	$\rightleftharpoons$	$\vdash P_1; P_2$
STR-NULL	$\vdash 0$	$\rightleftharpoons$	$\vdash$
STR-AND	$D_1 \wedge D_2 \vdash$	$\rightleftharpoons$	$D_1, D_2 \vdash$
STR-NODEF	$T \vdash$	$\rightleftharpoons$	$\vdash$
STR-DEF	$\vdash def\ D\ in\ P$	$\rightleftharpoons$	$D\sigma_{dv} \vdash P\sigma_{dv}$
(σ_{dv} substitutes fresh names to channels from $dv[D]$)			
RED	$J \triangleright P \vdash J\sigma_{rv}$	$\longrightarrow$	$J \triangleright P \vdash P\sigma_{rv}$
(σ_{rv} substitutes messages to parameters from $rv[J]$)			

Figure 2. Join-Calculus operational semantics.

$$C[\,.\,]_S ::= [\,.\,]_S \quad \mid \ P \mid C[\,.\,]_S \quad \mid \ def\ D\ in\ C[\,.\,]_S$$

Figure 3. Syntax rules for building evaluation contexts.

make the system evolve after resolution of message emissions:
$$def\ x(\overrightarrow{z}) \triangleright P\ in\ x(\overrightarrow{y}) \longrightarrow P\{\overrightarrow{y}/\overrightarrow{z}\}.$$

For observation, the join-calculus processes may be imbricated inside evaluation contexts. These contexts, whose syntax is given in Figure 3, define a set of captured names S. When a process is placed inside this context, its bound names are preserved if captured; otherwise, they are alpha-converted.

III. Autonomous self-replication in virology

Self-replication is at the heart of computer virology since it is the common denominator between viruses and worms. Referring to early works from [11], two fundamental concepts are mandatory for self-replication: a replication mechanism and the existence of a self-description also called self-reference.

As corroborated by [2], [3], [4], self-replication is linked to the concept of recursion, present in the different computation paradigms. In the provided definitions, both the self-reference and the replication mechanism can be identified. Definition 1 is the most flexible definition, compatible with former ones. By application of Kleene's recursion theorem [5], viruses are built as solutions of fixed point equations. In this definition, the replication mechanism is defined by the propagation function β. As for the self-reference, it is denoted by the variable v which is considered both as an executed program and a parameter according to the side of the equation. The program p constitutes the replication target and β implicitly contains a research routine for selecting valid targets for next replications.

Definition 1: Programs being indexed by a Gödel numbering, $\varphi_p(x)$ denotes the computation of the program p over x. According to [4], a virus v is a program which, for all values of p and x over the computation domain, satisfies $\varphi_v(p, x) = \varphi_{\beta(v,p)}(x)$ where β is the propagation function.

IV. Distributed self-replication

As stated by [12], self-replicating systems do not necessarily contain their own self-reference access or their own replication mechanism. They may rely on external services for these fundamental elements. Therefore, the advantages offered by process algebras become undeniable: exchanges between processes and their environment, distribution of the computations.

As seen in Section III, the self-reference notion is required to functionally express self-replication; so it is for process modeling. To reference themselves, programs are built as process abstractions (definition with a single pattern): $D_p = def\ p(\overrightarrow{arg}) \triangleright P$ where P is defined in function of the arguments $\overrightarrow{arg}$. The program execution then corresponds to an instantiating process: $E_p = def\ D_p\ in\ p(\overrightarrow{val})$. **This hypothesis will be kept all along the article**. Based on it, Definition 2 describes self-replication as the emission of this definition, or an equivalent, on an external channel.

Definition 2: (SELF-REPLICATION) A program is self-replicating over an external channel c if it can be expressed as a Join-Calculus definition capable to access or reconstruct itself before propagating on c (i.e. to extrude itself beyond its scope). The statement is translated as follows: $def\ s(c, \overrightarrow{x}) \ \triangleright\ P$ where $P \longrightarrow^* Q[def\ s'(\overrightarrow{x}) \triangleright P'\ in\ R[c(s')]]$ and $P' \approx P$. s denotes the self-reference, s' the equivalent program whereas R specifies the replication mechanism over c.

This first definition of self-replication is generic and covers several types of replicating codes, even mutating codes or codes reconstructed from environment pieces. To ease the remaining of the article, we will mainly focus on syntactic duplication which is a particular case of the definition where replication identically reproduces the code: $P \longrightarrow^* R[c(s)]$.

A. Modeling the environment

Before speaking of distribution, the execution environment in which processes evolve must be thoroughly defined. Execution environments share a global structure that can be specified by a generic evaluation context. Generally speaking, operating systems, just like any other execution environment, provide services (system calls) and resources (memory, files, registry). A system context denoted $C_{sys}[.]_{S \cup R}$ is thus built on service and resource bricks, formalized by channel definitions:

Services: The set of services S has a behavior similar to an execution server waiting for queries. Services computations are represented by a function f_{sv}. When a service is called, f_{sv} is computed over the arguments and sent back.

- $def\ S_{sv}(\overrightarrow{arg}) \triangleright return\ f_{sv}(\overrightarrow{arg})\ in\ ....$

Resources: The set of resources R provides storing facilities. Resources can be modeled by parametric processes storing information inside internal channels. Resources can be either static providing reading and writing accesses (data files) or executable possibly triggered on command (programs).

- For executables, let us consider f, f_0, f_n being functions:

$def\ R_{exec}(f_0) \triangleright def\ (write(f_n) | content{<}f{>}) \triangleright (content{<}f_n{>})$
$\wedge\ (read() | content{<}f{>}) \triangleright (return\ f\ to\ read | content{<}f{>})$
$\wedge\ (exec(\overrightarrow{a}) | content{<}f{>}) \triangleright (return\ f(\overrightarrow{a})\ to\ exec | content{<}f{>})$
$in\ content{<}f_0{>} | return\ read, write, exec\ to\ R_{exec}\ in\ ...$

B. Construction of the viral sets

Replication being formalized by extrusion of the process definition on an external channel, a process alone can not be infectious without access to the necessary services and resources. To observe these exchanges, the labeled transition system open-RCHAM will be used to make explicit the interactions with an abstract environment [10, pp.45-47]. Abstract environments are specified by a set of definitions and their defined name: here the services and resources.

Using this transition system, viruses can be defined according to the principle of viable replication. Viable replication guarantees that replicated intsances are still capable of self-replication. The programs satisfying viable self-replication constitute the viral sets [13]. Definition 3 redefines viral sets relatively to a system context conditioning the consumption of replicated definitions and the activation of intermediate infected forms. The sets are built by iteration starting with an original infection where the virus infects a first resource, followed by successive infections from resource to resource.

Definition 3: (VIRAL SET) Let us consider a system defining services S and resources R. Its set of defined names N is divided between services Sv, resource accesses in reading mode Rd, writing mode Wr, and execution mode Xc such as $N = Sv \cup Rd \cup Wr \cup Xc$. The current state of resources is represented by ΠR. The viral set E_v can be recursively constructed as follows:

$$E_v(C_{sys}[.]_N) = \{V \mid \exists \overrightarrow{w} \subset Wr,\ \overrightarrow{x} \subset Xc \; and \; n > 1 \; such \; as$$
$$S \wedge R \vdash_N V|\Pi R \xrightarrow{\mu_1;\{v\}\overline{w_0}<v>;\mu_2} S \wedge R \vdash_{N\cup\{v\}} V'|R_0|\Pi R$$
$$and \; for \; all \; 1 \leq i < n,$$
$$S \wedge R \vdash_N R_i|\Pi R \xrightarrow{x_i<\overrightarrow{a}>;\mu_1;\{v\}\overline{w_{i+1}}<v>;\mu_2} S \wedge R \vdash_{N\cup\{v\}} V'|R_{i+1}|\Pi R\}$$

The vector $\overrightarrow{w}$ constitutes writing accesses to infected resources and $\overrightarrow{x}$ activations of intermediate infected resources.

C. Distributed virus replication

1) Environment refinement: Specific services and resources must be defined because they may be externalized by the virus [12]: access to the self-reference, replication mechanisms and the necessarily external replication targets. The generically defined system must thus be refined to support these services and resources, concretely illustrated in Table I:

Self-reference access: Operating systems handle a list of executing processes for scheduling, with a pointer on the active process. A service is provided to access this list and the pointed process denoting the self-reference. To maintain the list, program executions are launched through a dedicated primitive $exec$. Scheduling being a service, sys_{upd} the updating primitive is private to avoid illegitimate modification, only reading access is made public through sys_{ref}.

- $D_{exe} \stackrel{\text{def}}{=} exec(p, \overrightarrow{args}) \; \triangleright \; sys_{upd}(p).return \; p(\overrightarrow{args}) \; to \; exec$
- $D_{ref} \stackrel{\text{def}}{=} sys_{upd}(r_n)|active<r> \triangleright active<r_n>$ $\wedge \; sys_{ref}()|active<r> \triangleright active<r>|return \; r \; to \; sys_{ref}$

Replication mechanism: The mechanism is represented by a function r copying data from an input channel towards and output channel. The function has been left parametric; however, it is strongly constrained to forward its input towards the output channel after an indefinite number of transformations.

- $D_{rep} \stackrel{\text{def}}{=} sys_{rep}(in, out) \; \triangleright \; return \; r(in, out) \; to \; sys_{rep}$

Table I
CHANNELS AND EQUIVALENT OS SERVICES AND RESOURCE ACCESSES.

Channels	Linux APIs	Windows APIs
$exec$	fork( ), exec( )	CreateProcess( )
sys_{ref}	getpid( ), readlink( )	GetModuleFileName( )...
sys_{rep}	sendfile( )	CopyFile( )
$\overrightarrow{sr}, \overrightarrow{sw}, \overrightarrow{se}$	fread( ), fwrite( )...	ReadFile( ), WriteFile( )...

Replication targets: A pool of executable resources constitutes the targets. Their definition D_{trg} is identical to the one in Section IV-A, allowing preexistence or dynamic creation.

A system context with n resources can now be defined to be used along the different definitions and proofs:
$C_{sys}[\,.\,]_{S\cup R} \stackrel{\text{def}}{=} def \; D_{exe} \wedge D_{ref} \wedge D_{rep} \wedge D_{trg} \; in$
$let \; sr_1, sw_1, se_1, ..., sr_n, sw_n, se_n =$
$R_{trg}(f_1), ..., R_{trg}(f_n) \; in \; (active<null> \,|\, [\,.\,])$
with $S = \{exec, sys_{ref}, sys_{rep}\}$ and $R = \{R_{trg}, \overrightarrow{sr}, \overrightarrow{sw}, \overrightarrow{se}\}$.

2) Classes of self-replicating viruses: Using this refined system context, the four classes of self-replicating viruses from [12] can be defined. Through these classes, the fundamental components for self-replication can be locally defined or exported: the access to the self-reference, the replication mechanism denoted by r, this function being constrained to reemit its input after a certain number of transformations. This function as well as the target research routine denoted by t are willingly left parameterizable in Definition 4.

Definition 4: Let V be a viral process. Let R and S be definitions responsible for the self-reference access and the replication mechanism. Additional definitions T and P are responsible for the target research and the payload:

- $R \stackrel{\text{def}}{=} loc_{rep}(in, out) \; \triangleright return \; r(in, out) \; to \; loc_{rep}$
- $S \stackrel{\text{def}}{=} loc_{ref}() \; \triangleright \; return \; v \; to \; loc_{ref}$
- $T \stackrel{\text{def}}{=} loc_{trg}() \; \triangleright return \; t() \; to \; loc_{trg}$
- $P \; is \; any \; process \; modeling \; a \; post\text{-}infection \; payload$

Viruses can be classified in four categories:

- **(Class I)** V is totally autonomous:
  $V_I \stackrel{\text{def}}{=} def \; v(\overrightarrow{x}) \triangleright (def \; S \wedge R \wedge T \; in$
  $loc_{rep}(loc_{ref}(), loc_{trg}()).P) \; in \; exec(v, \overrightarrow{a})$
- **(Class II)** V uses an external replication mechanism:
  $V_{II} \stackrel{\text{def}}{=} def \; v(\overrightarrow{x}) \triangleright (def \; S \wedge T \; in$
  $sys_{rep}(loc_{ref}(), loc_{trg}()).P) \; in \; exec(v, \overrightarrow{a})$
- **(Class III)** V uses external access to the self-reference:
  $V_{III} \stackrel{\text{def}}{=} def \; v(\overrightarrow{x}) \triangleright (def \; R \wedge T \; in$
  $loc_{rep}(sys_{ref}(), loc_{trg}()).P) \; in \; exec(v, \overrightarrow{a})$
- **(Class IV)** V uses only external services:
  $V_{IV} \stackrel{\text{def}}{=} def \; v(\overrightarrow{x}) \triangleright (def \; T \; in$
  $sys_{rep}(sys_{ref}(), loc_{trg}()).P) \; in \; exec(v, \overrightarrow{a})$

Through the parametrization, several types of replication mechanisms can be represented by refinement:
(1) overwriting infections: $def \; r(v, sw) \triangleright sw(v)$,
(2) append infections (respectively prepend): $def \, r(v, sw, sr) \triangleright$ $(let \, p = sr() \, in \, def \, p_1(\overrightarrow{arg}) \; \triangleright \; v().p(\overrightarrow{arg}) \, in \, sw(p_1))$,
Compared to Definition 2, viruses no longer take the target as parameter but uses a research routine that is also parametrized:
(1) hard-coded targets: $def \; t() \; \triangleright \; return \; n \; to \; t$,
(2) dynamically created targets:
$def \; t() \; \triangleright \; let \; sr, sw, se = R_{trg}(empty) \; in \; return \; sw \; to \; t$,

(3) discovered targets: $targets\ found\ by\ crawling\ through\ the\ file\ system\ searching\ for\ vulnerable\ resources.$
Independently of these parametrizations, these four classes of viruses achieve viable replication as stated by Proposition 1.

Proposition 1: If the system context provides the right services and valid targets, the virus classes I, II, III, IV achieve viable replication i.e. they appertain to its viral set.

Proof: Without modifying the proof core, let us consider a refined system and a simple case of parameterization:
$def\, r(x,w) \triangleright w(x)$ and $def\, t() \triangleright return\ sw_i\ to\ t\ at\ the\ i^{th}\ iteration$
Let us consider the virus class III knowing that an identical approach can provide proofs for the remaining classes:
$D_{V_{III}} \stackrel{\text{def}}{=} v() \triangleright def\ R \wedge T\ in\ loc_{rep}(sys_{ref}(), loc_{targ}()); P$
$D_{R_k} \stackrel{\text{def}}{=} sw_k(f_n)|content_k<f> \triangleright content_k<f_n>$
$\wedge\ sr_k()|content_k<f> \triangleright (content_k<f>|return\ f\ to\ sr_k$
$\wedge\ se_k(\overrightarrow{a})|content_k<f> \triangleright content_k<f>|return\ exec(f, \overrightarrow{a})\ to\ se_k$

Proof of initial infection: $\vdash C_{sys}[V_{III}]_{S \cup R}$
$\rightleftharpoons$ (str-def+str-and)

$D_{exe}, D_{ref}, D_{rep}, D_{trg} \vdash let\ sr_1, sw_1, se_1, ..., sr_n, sw_n, se_n = R_{targ}(f_1), ..., R_{trg}(f_n)\ in\ (active<null> \mid V_{III})$
$\longrightarrow$ (react+str-def+str-and+str-def)

$D_{exe}, D_{ref}, D_{rep}, D_{trg}, D_{R_1}, ..., D_{R_n}, D_{V_{III}} \vdash content_1<f_1> \mid \Pi_{i=2}^{n} content_i<f_i> \mid active<null> \mid exec(v, \overrightarrow{a})$
$\longrightarrow$ (react+react)

$D_{exe}, D_{ref}, D_{rep}, D_{trg}, D_{R_1}, ..., D_{R_n}, D_{V_{III}} \vdash content_1<f_1> \mid \Pi_{i=2}^{n} content_i<f_i> \mid active<v> \mid v(\overrightarrow{a})$
$\longrightarrow$ (react+str-def+str-and)

$D_{exe}, D_{ref}, D_{rep}, D_{trg}, D_{R_1}, ..., D_{R_n}, D_{V_{III}}, R, T \vdash content_1<f_1>| \Pi_{i=2}^{n} content_i<f_i> \mid active<v> \mid loc_{rep}(sys_{ref}(), loc_{trg}()).P$
$\longrightarrow$ (react+react)

$D_{exe}, D_{ref}, D_{rep}, D_{trg}, D_{R_1}, ..., D_{R_n}, D_{V_{III}}, R, T \vdash content_1<f_1>| \Pi_{i=2}^{n} content_i<f_i> \mid active<v> \mid loc_{rep}(v, sw_1).P$
$\longrightarrow$ (react+react)

$D_{exe}, D_{ref}, D_{rep}, D_{trg}, D_{R_1}, ..., D_{R_n}, D_{V_{III}}, R, T \vdash content_1<v>| \Pi_{i=2}^{n} content_i<f_i> \mid active<v> \mid P$

Proof of successive infections: Once initial replication is achieved, following replications are activated by execution requests $se_i(\overrightarrow{a})$. From there, reduction is identical to the previous one except for loc_{trg} which returns sw_{i+1}. ■

V. Complex behaviors: Rootkits and stealth

This section illustrates the expressiveness of the Join-Calculus by describing stealth techniques, hard to describe in functional models. Even if stealth is not malicious on its own, deployed in Rookits, it becomes a powerful tool for attackers. Few formal works exist on Rootkit modeling [7], [8], [14]; it thus constitutes an interesting concrete case for refinement of the payload process which has not been detailed yet. Let us consider the common case of Rootkits offering hooking functionalities. The definition in [14] of viruses resident relatively to a system call is the closest to our approach. But, the used recursive functions are not really adapted to model the required reactiveness and persistency. The Join-Calculus should offer far more flexibility.

System call hooking: Hooking mechanisms allow the interception of system calls. They rely on channel usurpation by alteration of the structures storing the access information to system calls. A new resource of the system must thus be

Table II
Parallel with Kernel Rootkits.

SuckIt (Linux kernel Rootkit, [15], 2001)	
Process	**Implementation**
R_{kit}	$core$, embedded kernel module containing the fake calls R_{fsc}.
D_{tsc}	Linux system call table.
D_{alloc}	memory device $/dev/kmem$.
Channel	**Implementation**
$alloc$	$kmalloc$.
$hook$	write function called with the address returned by $kmalloc$.
$publish$	$sysenter$ switching between user and kernel space.
$\overrightarrow{fsc}$	memory addresses of the fake system calls: $fork$, $open$, $kill$...
Agony (Windows kernel Rootkit by Intox7, [16], 2006)	
Process	**Implementation**
R_{kit}	$agony.sys$, embedded kernel module with the fake calls R_{fsc}.
D_{tsc}	SSDT ($System\ Service\ Descriptor\ Table$).
D_{alloc}	memory allocation services.
Channel	**Implementation**
$alloc$	$MmCreateMdl$ now replaced by $IoAllocateMdl$.
$hook$	writing operation to the space newly allocated.
$publish$	$sysenter$ instruction switching between user and kernel space.
$\overrightarrow{fsc}$	addresses of the fake system calls: $ZwQueryDirectoryFile$...

defined: the system call table. This entity publishes the list of available system calls on-demand. This list is modeled by a vector of channel $\overrightarrow{sc}$ which can only be modified by the kernel through a privileged writing access. This privileged access is provided by the $priv$ channel which from the malware perspective is private: only the $publish$ channel is made public:
$D_{tsc} \stackrel{\text{def}}{=} T_{sc}(\overrightarrow{t_0}) \triangleright def\ (priv(\overrightarrow{t_n}) \mid table<\overrightarrow{t}>) \triangleright table<\overrightarrow{t_n}>$
$\wedge\, (publish() \mid table<\overrightarrow{t}>) \triangleright return\ \overrightarrow{t}\ to\, publish \mid table<\overrightarrow{t}>$
The services of memory allocation can be diverted to gain access to this privileged channel. In fact, they can be used to modify the page protection of a memory space. In practice, they take as input a base address b and a size s and return an access to the allocated space. The $hook$ is only leaked if the base address is equal to the address of the system call table $scbase$. Otherwise, a simple access is returned:
$D_{alloc} \stackrel{\text{def}}{=} alloc(b, s) \triangleright$
$if\ [b = scbase]\ then\ return\ hook\ else\ return\ access$
The interest of hooking for the rootkit is to define a set of false system calls $R_{fsc1}, ..., R_{fscm}$, in order to hide files or processes, for example by filtering the original system calls. These malicious calls are registered in a new table being a vector of m entries $\overrightarrow{fsc} = fsc_1 ... fsc_m$:
$D_{fsc} \stackrel{\text{def}}{=} fsc_1(\overrightarrow{arg}) \triangleright R_{fsc1} \wedge\ ...\ \wedge fsc_m(\overrightarrow{arg}) \triangleright R_{fscm}$
$R_{kit} \stackrel{\text{def}}{=} def\ D_{fsc}\ in\ let\ hk = alloc(scbase, scsize)\ in\ hk(\overrightarrow{fsc})$
The system evolves along the following reduction where the privileged hook is leaked from the allocation mechanism:
$def\ D_{tsc} \wedge D_{alloc}\ in\ let\ pub = T_{sc}(\overrightarrow{sc})\ in\ R_{kit} \longrightarrow *$
$def\ D_{tsc} \wedge D_{alloc} \wedge D_{fsc}\ in\ table<\overrightarrow{fsc}>$
For validation, Table II draws a parallel between processes, definitions and their implementation in representative malware.

VI. Replication detection / System resilience

Since [17], it is well established that virus detection is an undecidable problem. However, thanks to this formalism, some fragments of the Join-Calculus can be identified for which the detection problem remains decidable up to a complexity factor. Let us consider an algorithm taking as input a system context and a process. The algorithm returns true if the process is able to self-replicate inside the context. Such an algorithm can be used either for detecting replication capabilities or assessing the context resilience to a viral class. An exhaustive procedure is described in Algorithm 1 whose purpose respectively changes whether the context or the process varies.

Algorithm 1 is not designed for operational deployment; it uses a brute-force approach for state exploration in order to study the decidability of detection. Without surprise, detection remains undecidable according to Proposition 2. However, according to this same proposition, it becomes decidable by restricting name generation. This restriction is not without impact on the system. Forbidding name generation induces a fixed number of resources without possibility to dynamically create new ones. But most importantly, without name generation, synchronous communication is no longer possible because fresh names can not be generated for return values.

Proposition 2: Detection of self-replication in the Join-Calculus is undecidable. Detection becomes decidable if the system context and the process are defined in the fragment of the Join-Calculus without name generation.

Proof: In algorithm 1, the set E_{succ} of states reached after reduction is finite because internal transitions τ are finite state branching [18]. The decidability thus depends on the bounded state exploration. To prove this decidability, detection is reduced to coverability in petri nets.

Let us consider the fragment of the join-calculus without name generation i.e. no nested definitions. This fragment can be encoded in the asynchronous π-calculus without external choice. Let us consider a similar encoding to [19] except that replication operators are replaced by recursive equations:

$$[[Q|R]]_j = [[Q]]_j \mid [[R]]_j$$
$$[[x{<}v{>}]]_j = \bar{x}v$$
$$[[def\ x{<}u{>} \mid y{<}v{>} \triangleright Q\ in\ R]]_j = \begin{cases} A = x(u).y(v).([[Q]]_j \mid A) \\ A \mid [[R]]_j \end{cases}$$

The process inside its system context can thus be encoded in the asynchronous π-calculus, resulting in a system of parametric equations. Name generation being excluded, scope restriction ν is absent from the encoding. The proof is then similar to [20]. The system is encoded into equations from the Calculus of Communicating Systems. CCS is parameterless, however, without name generation, channels σ and transmitted values a can be combined into parameterless channels $<\sigma, a>$. The encoding reintroduces external choices to handle their combination. Just like in [20], the obtained system contains a set of parallel processes guarded by channels:

$$A_i = \Sigma <\sigma, a> . <\sigma', a'> .(\Pi \overline{<\sigma, a>} \mid \Pi A_j)$$

In this equation system, replication is detected by the potential activation of a guarded processes A_i by a channel $<\sigma, p>$ with $\sigma \in R$ and p is the abstraction of P. This is a typical control reachability problem in CCS. As proven in [20], control reachability can be reduced to a coverability problem in petri nets and decidable algorithms exist to compute it [21]. ■

VII. Policies to prevent malware propagation

The facts that detection is only decidable under cumbersome constraints and that it is reactive instead of proactive, encourage the research of alternative solutions. Proactive approaches must be considered to prevent malware propagation.

Algorithm 1 Replication detection.

Require: P which is abstracted by p
Require: $C_{sys}[.]_{S \cup R}$ exporting services S and resources R
$E_{done} \leftarrow \oslash$, $E_{next} \leftarrow \oslash$, $C \leftarrow C_{sys}[P]_{S \cup R}$
repeat
  $E_{succ} \leftarrow \{C' | C \xrightarrow{\tau} C'\}$
  if $\exists C'$ reached by resource writing $w{<}p{>}$ **then**
    return *system is vulnerable to the replication of P*
  end if
  $E_{succ} \leftarrow E_{succ} \backslash \{C_d \in E_{succ} | \exists C_t \in E_{done} . C_d \equiv C_t\}$
  $E_{next} \leftarrow E_{next} \cup E_{succ}$, $E_{done} \leftarrow E_{done} \cup \{C\}$
  Choose a new $C \in E_{next}$
until $E_{next} = \oslash$ **or** infinite reaction without new transitions
return *system is not vulnerable to the replication of P*

A. Non-infection property and isolation

A different approach to fight back malware is to reason in terms of information flow as initiated in [17]. Addressing confidentiality, the formalization of the non-interference property specifies that the behavior of low-level processes must not be influenced by upper-level processes to avoid illicit data flows between different security levels [22]. Similarly, self-replication in malware can be compared to an illicit information flow of the viral code towards the system. Let us state the hypothesis that, contrary to malware, legitimate programs do not interfere with other processes implicitly through the system. This issue refers to integrity and requires a new property: non-infection introduced in Definition 5.

Definition 5: (NON-INFECTION). For a process P placed inside a stable system context (i.e. reactions to intrusions only), the property of non-infection is satisfied if the system evolves along the reaction $C_{sys}[P] \longrightarrow^* C'_{sys}[P']$, and for any non-infecting process T the equivalence $C_{sys}[T] \approx C'_{sys}[T]$ holds.

Non-infection guarantees the integrity of the system context. The consequent question is to find the mandatory constraints for a system context to satisfy non-infection. Proposition 3 states that there exist systems preventing replication through resource isolation. This generalizes the network partitioning principle advocated in [17] to fight virus propagation.

Proposition 3: In a system context made up of services and resources, the non-infection property can only be guaranteed by a strong isolation of resources, forbidding all transitions $C_{sys}[.] \xrightarrow{x(\overrightarrow{y})} C'_{sys}[.]$ where x is a writing channel to a resource.

Proof: Let us consider a system context defining services D_S and resources D_R. The isolation requirement is proven by showing that writing accesses, either direct or indirect, must be forbidden. The stable context only reacts to intrusions:

I. Intrusion towards resources: $J = x_1(\overrightarrow{y_1})|...|x_n(\overrightarrow{y_n}) \triangleright R'$

$$def\ D_S \wedge D_R \backslash \{J\} \wedge J\ in\ R_0|x_1(\overrightarrow{z_1}).R_1|...|x_m(\overrightarrow{z_m}).R_m|[\,.\,]$$
$$\xrightarrow{x_{m+1}(\overrightarrow{z_{m+1}})|...|x_n(\overrightarrow{z_n})}$$
$$def\ D_S \wedge D_R\ in\ R_0|R_1|...|R_m|R'[\overrightarrow{y}/\overrightarrow{z}]|[\,.\,].$$

The x_i only store the resource content meaning that all $R_i = 0$. After simplification, three cases remain for transition:

1) Reading case: $R' \equiv x_1(\overrightarrow{y_1})|...|x_m(\overrightarrow{y_m})\ |return\ \overrightarrow{y_1}, ..., \overrightarrow{y_m}\ to\ x_{m+1}$. Once the return consumed, the system recovers its initial state; the non-infection property is satisfied.

2) Writing case: $R' \equiv x_1(\overrightarrow{y_{m+1}})|...|x_m(\overrightarrow{y_n})|return\ to\ x_{m+1}$. Once the return consumed, the original values y_i with $1 \leq i \leq m$ are substituted by values y_j with $m+1 \leq j \leq n$. The system may not recover its original state before the intrusion: the non-infection property may not be satisfied.

3) Execution case: Equivalent to the service case II).

II. Intrusion towards services: $J = x_1(\overrightarrow{y_1})|...|x_n(\overrightarrow{y_n}) \triangleright S$

$$def\ D_S \setminus \{J\} \wedge J \wedge D_R\ in\ R \mid [\,.\,] \xrightarrow{x_1(\overrightarrow{z_1})|...|x_n(\overrightarrow{z_n})} def\ D_S \wedge D_R\ in\ S[\overrightarrow{y}/\overrightarrow{z}] \mid R' \mid [\,.\,]$$

S is of the form $return\ f(\overrightarrow{z_1}, ..., \overrightarrow{z_n})\ to\ x_1$ which reduces to the null process when the return is consumed. The system modification thus depends on the behavior of the function f:

1) Case of f reading resource or no access: Identical to I.1).

2) Case of f writing or creating resources: Identical to I.2).

3) Case of f executing resources: The solution depends on the content of the resource. The same test is applied recursively to this content until II.1) or II.2). ■

B. Policies to restrict infection scope

Non-infection is impossible to guarantee in practice. Complete isolation can not obviously be deployed in systems without loosing most of their use [17]. To maintain utility, solutions restricting the resource accesses case-by-case, can still contain malware by confining the scope of the propagation.

An access authority deploys such restriction by blocking unauthorized accesses to the resources and services of a system. A solution based on access tokens can be considered, either for spatial restriction (only programs and resources sharing the same token can access each others) or for time restriction (each token is valid a fixed number of executions). [23] specifies access authorities as two components: a Policy Decision Point which can be seen as the token distribution mechanism and a Policy Enforcement Point which checks the token validity and thus must not be bypassed. If security tokens are not forgeable and no distribution mechanism is responsible for their extrusion, the process must not be able to access any service and resource. In fact, access control mechanisms are already deployed in two well known security models for Java [24] and .Net [25]. In both, the managed code is run in a isolated runtime environment with a controlled access to resources. The problem in actual system is that these controls are restricted to managed language and not to native code. Extending access controls to native code could fight malware propagation with a proven security.

VIII. Conclusion and perspectives

This paper introduces the basis for a unified malware model based on the Join-Calculus. Moving from the functional models used in virology to process-based models do not result in a loss of expressiveness. The fundamental results are maintained: characterization of self-replication, undecidability of detection and isolation for prevention. In addition, the model offers increased expressiveness by support of interactions, concurrency and non-termination, which ease the modeling of evolved malware. Beyond computational aspects, new results and perspectives have been provided with respect to detection and prevention. A fragment of the Join-Calculus has been identified where detection becomes decidable. With regards to prevention, a property of non-infection has been defined with potential solutions to restrict malware propagation. If non-infection is too strong in concrete cases, future works can be led to reduce the strength of the property. Typing mechanisms based on security levels constitute an interesting lead to restrict accesses to critical resources and services [22].

Acknowledgement The authors would like to thank G. Bonfante and J-Y. Marion for their help and working leads during the exploration of the process algebras, in particular on the detection decidability and prevention solutions.

References

[1] G. Jacob, E. Filiol, and H. Debar, "Malwares as interactive machines: A new framework for behavior modelling," *Journal in Computer Virology*, vol. 4, no. 3, pp. 235–250, 2008.

[2] L. M. Adleman, "An abstract theory of computer viruses," in *Proc. Advances in Cryptology (CRYPTO)*, 1990, pp. 354–374.

[3] J. Kraus, "On self-reproducing programs, translated and edited by D. Bilar and E. Filiol," *Journal in Computer Virology*, vol. 5, no. 1, 2009.

[4] 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.

[5] H. Rogers, *Theory of Recursive Functions and Effective Computability*. The MIT Press, 1987.

[6] E. Filiol, "Formalisation and implementation aspects of k-ary (malicious) codes," *Journal in Computer Virology*, vol. 3, no. 3, 2007.

[7] ——, "Formal model proposal for (malware) program stealth," in *Proc. Conf Virus Bulletin (VB)*, 2007.

[8] A. Derock and P. Veron, "Another formal proposal for stealth," in *World Academy Science, Engineering & Technology*, 2008, pp. 158–164.

[9] C. Fournet, "The join-calculus: a calculus for distributed mobile programming," Ph.D. dissertation, Ecole Polytechnique, Palaiseau, 1998.

[10] C. Fournet and G. Gonthier, "Join-calculus: a language for distributed mobile programming," in *Applied Semantics Summer School*, 2000.

[11] J. von Neumann, *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[12] M. Webster and G. Malcolm, "Reproducer classification using the theory of affordances," in *Proc. IEEE Symp. Artificial Life*, 2007, pp. 115–122.

[13] F. B. Cohen, "Computational aspects of computer viruses," *Computers & Security*, vol. 8, no. 4, pp. 325–344, 1989.

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

[15] Sd and Devik, "0x07 - linux on-the-fly kernel patching without lkm," *Phrack*, vol. 58, 2001.

[16] G. Hoglund and J. Butler, *Rootkits, Subverting the Windows kernel*. Addison-Wesley Professional, 2006.

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

[18] C. Laneve, "May and must testing in the join-calculus: Ublcs-96-4," University of Bologna, Tech. Rep., 1996.

[19] C. Fournet and G. Gonthier, "The reflexive cham and the join-calculus," in *Proc. ACM Symp. Principles on Programming Languages*, 1996, pp. 372–385.

[20] R. M. Amadio and C. Meyssonnier, "On decidability of the control reachability problem in the asynchronous π-calculus," *Nordic Journal of Computing*, vol. 9, no. 2, pp. 70–101, 2002.

[21] P. Küngas, "Petri net reachability checking is polynomial with optimal abstraction hierarchies," in *Proc. Int. Symp. Abstraction, Reformulation and Approximation (SARA)*, 2005.

[22] M. Hennessy and J. Riely, "Information flow vs. resource access in the asynchronous π-calculus," *ACM Trans. Programming Languages and Systems*, vol. 25, no. 4, pp. 566–591, 2002.

[23] R. Yavatkar, D. Pendarakis, and R. Guerin, "A framework for policy-based admission control," RFC-2753, Tech. Rep., 2000.

[24] L. Gong, G. Ellison, and M. Dageforde, *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Prentice Hall, 2003.

[25] A. Freeman and A. Jones, *Programming .NET Security*. O'Reilly, 2003.