J Comput Virol
DOI 10.1007/s11416-009-0137-1

ORIGINAL PAPER

A general definition of malware

Simon Kramer · Julian C. Bradfield

Received: 1 July 2008 / Accepted: 9 September 2009
© The Author(s) 2009. This article is published with open access at Springerlink.com

Abstract We propose a general, formal definition of the concept of *malware* (*mal*icious soft*ware*) as a single sentence in the language of a certain *modal logic*. Our definition is general thanks to its abstract formulation, which, being abstract, is independent of—but nonetheless generally applicable to—the manifold concrete manifestations of malware. From our formulation of malware, we derive equally general and formal definitions of *benware* (*ben*ign soft*ware*), *anti-malware* ("antibodies" against malware), and *medware* (*med*ical soft*ware* or "medicine" for affected software). We provide theoretical tools and practical techniques for the *detection*, *comparison*, and *classification* of malware and its derivatives. Our general defining principle is *causation of (in)correctness*.

1 Introduction

In [1], the definition of the concept of *malware* (for *mal*icious soft*ware*) is posed as a difficult open problem in computer virology. This paper, which is an extended version of [2], proposes a simple, general, and formal solution to that problem as a single sentence in the language of a *modal fixpoint logic*. Such a single-sentence definition was opined to be difficult to come up with in [3, p. 19]. Our paper also proposes a formal definition of the concepts of *benware* (for *ben*ign soft*ware*), *anti-malware* ("antibodies" against malware), and *medware* (for *med*ical soft*ware*, or "medicine" for affected software) as derivatives of the concept of malware in the sense of being defined in terms of malware. Our formal definitions are related to the practical definitions of [4], which we compare to ours in Sect. 2.2.

Intuitively, malware is software that harmfully attacks other software[1] (cf. [5], [6], [7], and [8]), where *to harmfully attack* can be observed to mean *to cause the actual behaviour to differ from the intended behaviour*. Unfortunately, intended behaviour is rarely defined, i.e., stated explicitly as such. Rather, such behaviour is usually painfully discovered *a posteriori*, i.e., in the aftermath of an attack, as the lesson learnt from the damage caused by the attack. The difference between the actual and the intended behaviour of software systems being defined and experienced as incorrectness in verification[2] and validation,[3] respectively, the defining characteristic of malware must be (direct or indirect) *causation of incorrectness*. Hence, any (formal) definition of the concept of malware depends on the definition of the concept of software system correctness. Logically speaking, a harmful attack on a software system is nothing else than the falsification of a necessary condition for the correctness of that

Simon Kramer's contribution was initiated in the Comète group at Ecole Polytechnique and INRIA (France), and completed under Grant P 08742 from the Japan Society for the Promotion of Science in the Laboratory of Cryptography and Information Security at the University of Tsukuba (Japan). Guillaume Bonfante and Jean-Yves Marion, LORIA, Nancy, France have been invited as guest editors for this paper.

S. Kramer (✉)
Ecole Polytechnique and INRIA, Palaiseau, France
e-mail: simon.kramer@a3.epfl.ch

J. C. Bradfield
University of Edinburgh, Edinburgh, UK
e-mail: jcb@inf.ed.ac.uk

[1] and possibly also indirectly hardware to the extent to which hardware is controllable by (affected) driver software.

[2] The process of checking compliance with the system's specification. This process is possibly proof-based.

[3] The necessarily test-based process of checking compliance with the users' needs. (The problem of whether or not users' needs are captured by specifications is non-mathematical.)

Published online: 29 September 2009

system. Hence, pieces of malware are falsifiers of the correctness hypothesis made *de facto* by the shipment of the software system (cf. *Popper's critical rationalism* in philosophy and *duty of care* in law).

The correctness of a software system s can be expressed in at least three different styles:

1. *model*-theoretically, as the assertion that
   (a) $s \vDash \varphi$, pronounced "s satisfies φ", where φ is a logical formula (the *specification*) expressing the desired correctness criterion (possibly a finite conjunction of necessary sub-criteria)
   (b) $s \equiv s'$, pronounced "s is equivalent to s'", where s' is a *specification* term that realises more abstractly (by focusing on the *what*) what the *implementation* s is intended to realise more concretely (by focusing on the *how*)
2. *proof*-theoretically, as the assertion that $\mathrm{Ax}(s) \vdash \varphi$, pronounced "$\varphi$ (the *specification*) is deducible from the axiomatisation of s", where an axiomatisation of a software system s is a set of logical formulae adequately capturing the meaning of s.

Remark 1 Style 1.a (also known as *model-checking* [9]) and Style 1.b (also known as *equivalence-checking*[4] [10]) can sometimes be related by:

$$s \equiv s' \quad \text{iff} \quad \text{for all } \varphi, s \vDash \varphi \text{ if and only if } s' \vDash \varphi.$$

Style 1 and Style 2 (also known as *theorem-proving* [11,12]) can sometimes be related by the adequacy (soundness and completeness) of the axiomatisation:

$$\text{for all } \varphi, \mathrm{Ax}(s) \vdash \varphi \text{ if and only if } s \vDash \varphi.$$

Our formal definition of the concept of malware abstracts from—and thus is independent of—*how* correctness is established or violated; it is just decided by *that* either is the case—by means of software verification and/or validation. Our definition is therefore independent of the manifold concrete manifestations of malware, which only affect the way by which correctness is violated (e.g., via *polymorphic* and *metamorphic code* [3, Sects. 7.5 and 7.6, respectively]). Hence given a software system s, we shall designate by **correct**(s) the abstract fact that s is correct. Concretely, when there is a specification of some form (e.g., a formula ϕ, a term s', or an intuitive judgement in the engineer's mind) for s:

– for model-checking,

$$\mathbf{correct}(s) :\text{iff } s \vDash \varphi$$

– for equivalence-checking,

$$\mathbf{correct}(s) :\text{iff } s \equiv s'$$

– for theorem-proving,

$$\mathbf{correct}(s) :\text{iff } \mathrm{Ax}(s) \vdash \varphi$$

– and otherwise,

$$\mathbf{correct}(s) :\text{iff } s \text{ is intuitively considered correct,}$$

where ":iff" abbreviates "by definition, if and only if".

We stress that the logical strength and computational complexity of **correct** may vary with the chosen notion of correctness. Formally, **correct** may even be undecidable on a chosen domain of software systems, e.g., when already **correct**(s) is undecidable for some s in the domain. However, the pragmatic stance of deciding **correct**(s) by vigour of the intuition that security experts can acquire from gathering practical experience with s, is of course possible for any s. Hence empirically, **correct** can be decided on a given domain by harnessing the experience of security experts in that domain, e.g., by looking-up a table in which the experts record their decision about **correct**(s) for each s in the domain. Theoretically speaking, **correct** is decidable on a given (finite) domain (in the look-up time of the table) with an *oracle* for **correct**(s) for each s in the domain. In the ideal case of *proof-carrying code* [13], a software system comes even with its own, formal proof of correctness (w.r.t. a given specification), amenable to fully automated checking.

Correctness necessarily being formulated w.r.t. an underlying specification,[5] the formal definition of malware is necessarily *conditioned* on that specification. Consequently, a software system s' harming another software system s fails to qualify *formally* as malware if and only if the intention of the proper functioning of s is improperly, i.e., wrongly (*mal-specification*) or incompletely (*under-specification*), specified. From an engineer's perspective, mal-specification is an unsuccessful attempt, whereas under-specification is an unattempted success. However from a user's perspective, mal- and under-specification are indistinguishable in the sense of being equally bad failures of protection against harmful effects. (For example, the disjunction of both is a necessary condition for *Trojanisation*, i.e., the intended addition of improper functionality to a software system by a piece of malware—a Trojan horse.) From a user's perspective, all that matters is effect, not intention. In particular, maliciousness is immaterial. Note that failures due to mal- and/or

[4] or also as *refinement-checking* when only the intrinsic pre-order of the equivalence is used.

[5] expressed by a formula or term in the formal case, and by a natural language proposition, or even an unexpressed intuition, whose truth values we assume recorded as a Boolean value, in the informal case.

under-specification are the fault (the psychological failure) of the human specifier and not a fault (a logical failure) of our approach. Any software specification that is improper in the sense of allowing for functionality that is both *orthogonal*[6] to the correctness criterion and *undesirable* (side-)effects the creation of a trapdoor for dysfunctionality. Hence, a major engineering challenge is to make exhaustively explicit the intention of proper software functionality.

Malware causing incorrectness to a software system s, malware must *transform* s (i.e., violate the integrity of the code of s) and/or the environment (e.g., the operating system, other programs and data, the memory and/or file system, etc.)[7] of s, which for the sake of faithful modelling we assume to be included in s. Hence, pieces of malware are higher-order programs (i.e., source or binary mal*code*) or, more generally, higher-order software systems (e.g., *botnets* [14, Chap. 11]), in the sense of taking as input other programs or, more generally, other software systems, which they transform [15]. Thus, we shall designate the transformation (both the action and the result) of a software system s (but not of the corresponding predicate **correct**) by a piece of malware s' by the *functional application* $s'(s)$.[8]

2 Malware and its derivatives

2.1 Auxiliary definitions

Definition 1 (Damaging software) A software system s *damages* a correct software system s' by definition if and only if s (directly or indirectly) causes incorrectness to s'. Formally,

s damages s' :iff $\mathbf{correct}(s')$ and not $\mathbf{correct}(s(s'))$
s damages^0 s' :iff s damages s' s damages^{n+1} s' :iff there is s'' s.t. not s'' damages$^\circ$ s' and $s(s'')$ damages^n s' s damages$^\circ$ s' :iff $\bigcup_{n\in\mathbb{N}} s$ damages^n s'.

Observe that *causation* of software incorrectness is the damaging affection of *a priori* correct software with incorrectness via functional application; and that damaging software is defined to be so already due to its damaging *potential*. This potential can be practically tested in a sandbox of software systems (i.e., an isolated computer network) and/or in a sandbox within a software system (i.e., a virtual machine), as commonly done by anti-malware research laboratories and pieces of anti-malware, respectively (cf. [18] and [3, Sects. 11.4 and 11.13]).

Definition 2 (Repairing software) A software system s *repairs* an incorrect software system s' by definition if and only if s (directly or indirectly) causes correctness to s'. Formally,

s repairs s' :iff not $\mathbf{correct}(s')$ and $\mathbf{correct}(s(s'))$
s repairs^0 s' :iff s repairs s' s repairs^{n+1} s' :iff there is s'' s.t. not s'' repairs$^\circ$ s' and $s(s'')$ repairs^n s' s repairs$^\circ$ s' :iff $\bigcup_{n\in\mathbb{N}} s$ repairs^n s'.

Observe that repairing software is the *contrary*—not the contradictory (via negation)—of damaging software. Hence, analogous observations to those about Definition 1 apply here.

We stress that the properties of the relations damages$^\circ$ and repairs$^\circ$ depend on the properties of the predicate **correct**, which in turn depend on the underlying specifications. In particular, in specific cases it is possible to reduce indirect to direct damage by strengthening correctness to catch the specific indirection. We also stress that our (linear) definition of indirect causation is just an example, (e.g., non-linear) redefinitions are possible. The relations damages$^\circ$ and repairs$^\circ$ are plug-ins to our framework.

Definition 3 (Malware Logic) Let $\mathscr{M}$ designate a countable set of propositional variables M, and let

$$\Phi \ni \phi ::= M \mid \neg\phi \mid \phi \wedge \phi \mid \forall\mathsf{D}(\phi) \mid \forall\mathsf{R}(\phi) \mid \boldsymbol{\nu} M(\phi)$$

designate the language Φ of MalLog where all free occurrences of M in ϕ of $\boldsymbol{\nu} M(\phi)$ are assumed to occur within an even number of occurrences of $\neg$ to guarantee the existence of (greatest) fixpoints (expressed by $\boldsymbol{\nu} M(\phi)$) [19]. Then, given the (or only some finite or infinite sub-)class $\mathscr{S}$ of software systems s (not just pieces of software), and an interpretation $[\![\cdot]\!] : \mathscr{M} \to 2^{\mathscr{S}}$ of propositional variables, the interpretation $\|\cdot\|_{[\![\cdot]\!]} : \Phi \to 2^{\mathscr{S}}$ of MalLog-propositions is:

$$\begin{aligned}
\|M\|_{[\![\cdot]\!]} &:= [\![M]\!] \\
\|\neg\phi\|_{[\![\cdot]\!]} &:= \mathscr{S} \setminus \|\phi\|_{[\![\cdot]\!]} \\
\|\phi \wedge \phi'\|_{[\![\cdot]\!]} &:= \|\phi\|_{[\![\cdot]\!]} \cap \|\phi'\|_{[\![\cdot]\!]} \\
\|\forall\mathsf{D}(\phi)\|_{[\![\cdot]\!]} &:= \{s \in \mathscr{S} \mid \text{for all } s' \in \mathscr{S}, \text{if } s \text{ damages}^\circ\, s' \\
&\qquad \text{then } s' \in \|\phi\|_{[\![\cdot]\!]}\}
\end{aligned}$$

[6] i.e., logically incomparable (unrelated by logical implication).

[7] e.g. in [16], a virus is defined as a program that modifies other programs to include a possibly evolved copy of itself, and in [17], as a program that alters the operating system.

[8] Modelling transformation as function application is result-oriented. A process-oriented alternative is modelling transformation as parallel composition in the style of process algebra [10].

$$\|\forall\mathbf{R}(\phi)\|_{[\![\cdot]\!]} := \{s \in \mathscr{S} \mid \text{for all } s' \in \mathscr{S}, \text{if } s \text{ repairs}^\circ s' \text{ then } s' \in \|\phi\|_{[\![\cdot]\!]}\}$$

$$\|\boldsymbol{\nu} M(\phi)\|_{[\![\cdot]\!]} := \bigcup\{ S \subseteq \mathscr{S} \mid S \subseteq \|\phi\|_{[\![\cdot]\!]_{[M\mapsto S]}} \},$$

where $[\![\cdot]\!]_{[M\mapsto S]}$ maps M to S and otherwise agrees with $[\![\cdot]\!]$.

Further, $\phi \vee \phi' := \neg(\neg\phi \wedge \neg\phi')$, $\top := \phi \vee \neg\phi$, $\bot := \neg\top$, $\phi \rightarrow \phi' := \neg\phi \vee \phi'$, $\phi \leftrightarrow \phi' := (\phi \rightarrow \phi') \wedge (\phi' \rightarrow \phi)$, $\exists\mathbf{D}(\phi) := \neg\forall\mathbf{D}(\neg\phi)$, $\exists\mathbf{R}(\phi) := \neg\forall\mathbf{R}(\neg\phi)$, and, notably, $\boldsymbol{\mu} M(\phi(M)) := \neg\boldsymbol{\nu} M(\neg\phi(\neg M))$. Finally,

- for all $s \in \mathscr{S}$ and $\phi \in \Phi$, $s \models \phi$:iff $s \in \|\phi\|_{[\![\cdot]\!]}$[9]
- for all $\phi \in \Phi$, $\models \phi$:iff for all $s \in \mathscr{S}$, $s \models \phi$
- for all $\phi, \phi' \in \Phi$,
  - $\phi \Rightarrow \phi'$:iff for all $s \in \mathscr{S}$, if $s \models \phi$ then $s \models \phi'$
  - $\phi \Leftrightarrow \phi'$:iff $\phi \Rightarrow \phi'$ and $\phi' \Rightarrow \phi$.

Notice that MalLog lacks atomic propositions (but not propositional variables). Note also that in general, indirect damage and repair are not definable in terms of their direct counterparts within MalLog (e.g., with a fixpoint operator) because of the involved indirection introduced by the application operation in their definition.

We pronounce $\forall\mathbf{D}(\phi)$ as "necessarily through damage that ϕ", $\forall\mathbf{R}(\phi)$ as "necessarily through repair that ϕ", $\exists\mathbf{D}(\phi)$ as "possibly through damage that ϕ", $\exists\mathbf{R}(\phi)$ as "possibly through repair that ϕ", $\boldsymbol{\nu} M(\phi)$ as "the greatest fixpoint of the interpretation of the property M such that ϕ", $\boldsymbol{\mu} M(\phi)$ as "the least fixpoint of the interpretation of the property M such that ϕ", $s \models \phi$ as "s satisfies ϕ", $\models \phi$ as "ϕ is valid", $\phi \Rightarrow \phi'$ as "ϕ' is a logical consequence of ϕ", and $\phi \Leftrightarrow \phi'$ as "ϕ is logically equivalent to ϕ".

In addition to the present modalities, we could conceive corresponding *converse modalities* defined in terms of the converses of the present (active) accessibility relations: the active forms "damages" and "repairs" would be transformed into the passive forms "damaged by" and "repaired by", respectively.[10] This victim view, would focus on the (passive) susceptibility to damage and repair of software systems rather than on their (active) damaging or repairing potential.

Fact 1 *1.* $\models \phi \rightarrow \phi'$ *iff* $\phi \Rightarrow \phi'$ *(By expansion of the definitions.)*
2. $\models \phi \leftrightarrow \phi'$ *iff* $\phi \Leftrightarrow \phi'$
3. MalLog is a member of the family of μ*-calculi over the modal system* $\mathbf{K}_2$*, which is characterised by the laws of propositional logic and the modal laws* $\models \Box(\phi \rightarrow \phi') \rightarrow (\Box\phi \rightarrow \Box\phi')$ *and "if* $\models \phi$ *then* $\models \Box\phi$*", where* $\Box \in \{\forall\mathbf{D}, \forall\mathbf{R}\}$.
(As stressed before, the properties of the relations damages° *and* repairs°*, and thus those of the modalities* $\forall\mathbf{D}$ *and* $\forall\mathbf{R}$*, depend on the predicate* **correct***, whose properties in turn depend on the underlying specifications. Hence, no more constraints than those of the modal system* ***K****, i.e., none, can generally be assumed to hold for* $\forall\mathbf{D}$ *and* $\forall\mathbf{R}$*. Note also that since MalLog lacks atomic propositions, satisfiability for MalLog is definable with a mere* first-*order existential quantifier. Whereas satisfiability for a "full" member of the family of* μ*-calculi is definable only with a* second-*order quantifier, namely one over propositional valuations, i.e., functions from atomic propositions to those sets of points where these propositions are true. See [19] for details.)*

Corollary 1 *1. If* damages° *and* repairs° *are decidable on a given software systems domain then the satisfiability problem for MalLog, i.e., "Given* $\phi \in \Phi$*, is there* $s \in \mathscr{S}$ *s.t.* $s \models \phi$*?", (and thus also the model-checking problem, i.e., "Given* $\phi \in \Phi$ *and* $s \in \mathscr{S}$*, is it the case that* $s \models \phi$*?") is decidable.*
2. MalLog has the finite-model property, i.e., for all $\phi \in \Phi$*, if* ϕ *is satisfiable then* ϕ *is satisfiable in a finite model.*
3. MalLog is axiomatisable by the following Hilbert-style proof-system:
(a) the axioms and rules of the modal system ***K*** *for each* $\forall\mathbf{D}$ *and* $\forall\mathbf{R}$
(b) the axiom $\dfrac{}{\phi(\boldsymbol{\mu} M(\phi(M))) \rightarrow \boldsymbol{\mu} M(\phi(M))}$
(c) the rule $\dfrac{\phi(\phi') \rightarrow \phi'}{\boldsymbol{\mu} M(\phi(M)) \rightarrow \phi'}$.

Remark 2 1. In computer science, the most popular modal μ-calculus is one over $\mathbf{K}_n$ where $n \in \mathbb{N}$ and accessibility is given by a family of n action-labelled transition relations [19].
2. MalLog, being a modal μ-calculus, is related to monadic second-order logic, of which modal μ-calculi are bisimulation-invariant fragments [19, Sect. 8.1], and to the theory of non-well-founded sets [20].

2.2 Main definitions

Definition 4 (Malware) A software system s is *malware* by definition if and only if s damages non-damaging software systems (the civil population so to say) or[11] software

[9] The reader is invited not to confuse the satisfaction relation $\models$ used for the property-based definition of software correctness with the one $\models$ of MalLog used for the definition of malware-based concepts *on top of* software correctness.

[10] in analogy to temporal logic with future and past (converse future) modalities.

[11] This "or" is implicit in the second "if—then" of Condition 1 $\exists\mathbf{D}(\forall\mathbf{D}(M)) \Leftrightarrow \exists\mathbf{D}(\forall\mathbf{D}(\bot) \vee \forall\mathbf{D}(M))$.

Table 1 Enumeration of safe software systems (safe SWS)

(a)	non-damaging SWS (CP)
(b)	SWS that damage only SWS that damage CP (ATF1)
(c)	SWS that damage only SWS that damage ATF1 (ATF2)
(d)	SWS that damage only SWS that damage ATF2 (ATF3)
(e)	etc.

systems that damage malware (the anti-terror force so to say). Formally,

$$\boxed{\mathrm{mal}(s) \quad :\text{iff} \quad s \models \nu M(\exists\mathbf{D}(\forall\mathbf{D}(M))).}$$

Recall that $s \in \mathscr{S}$, i.e., the property of being a piece of malware is w.r.t. a certain context or environment $\mathscr{S}$ of software systems (possibly subject to certain closure conditions, e.g., closure under system composition).

The definition says that s is in the *greatest* fixpoint of the interpretation of *the property M such that*

if s satisfies M (i.e., s is in the interpretation of M) then s satisfies $\exists\mathbf{D}(\forall\mathbf{D}(M))$—which in turn says (the reader is invited to expand the operator definitions in case of doubt) that there is s' such that s damages s' and for all s'', if s' damages s'' then s'' satisfies M. (1)

Observe that all (=1) free occurrences of M in $\exists\mathbf{D}(\forall\mathbf{D}(M))$ of $\nu M(\exists\mathbf{D}(\forall\mathbf{D}(M)))$ occur within an even (=2) number of occurrences of $\neg$. Hence, our definition is formally well-defined.

This compact *co-inductive* definition has the following informal iterative paraphrase:

– Everything is malware (better be safe than sorry);
– except for (throw *out* what is clearly safe) the software systems enumerated in Table 1.

In contrast, and for a better understanding (especially of fixpoint operators), consider the opposite (in fact the contradictory) of malware:

Definition 5 (Benware) A software system s is *benware* by definition if and only if s is non-damaging or damages only software systems that damage benware. Formally,

$$\boxed{\mathrm{ben}(s) \quad :\text{iff} \quad s \models \mu M(\forall\mathbf{D}(\exists\mathbf{D}(M))).}$$

The definition says that s is in the *least* fixpoint of the interpretation of *the property M such that*

if s satisfies $\forall\mathbf{D}(\exists\mathbf{D}(M))$—which in turn says that for all s', if s damages s' then there is s'' such that s' damages s'' and s'' satisfies M—then s satisfies M.

This *inductive* definition has the following informal iterative paraphrase:

– Nothing is benware (again, better be safe than sorry);
– except for (throw *in* what is clearly safe) the software systems enumerated in Table 1.

Benware is (and intuitively must be) the contradictory of malware because

Fact 2 ben(s) *if and only if not* mal(s),

as can be seen by flipping negation symbols. The difference between the least and the greatest fixpoint is that the greatest—as opposed to the least—allows loops in the damage relation (mutual damage).

Figure 1 depicts the partition of the class of software systems into malware and benware, which fight each other via functional application. Shape nesting symbolises set inclusion. Solid (dashed) shapes symbolise disjoint (possibly overlapping) sets. Arrows symbolise potential damage. And grey-shading symbolises software that potentially damages itself (and thus potentially wastes resources, which is considered to be bad). Notice that the possibility for the malware sets MW0, MW1, MW2, MW3, ... of being overlapping implies the possibility of potential *mutual* damage.

Observe that we use a *greatest* fixpoint at the *object*-level of a *modal* language to define *malware* (including computer viruses), whereas [21] use *least* fixpoints at the *meta*-level of an *equational* language to define computer *viruses*. The framework of [21] is *bottom-up*: it focuses on *how* computer viruses can be constructed. In contrast, our framework is *top-down*: it focuses on *what* malware effects but abstracts from how malware actually does so. We believe that bottom-up and top-down approaches are complementary: anti-malware measures may well be implemented in the form of counter-Trojans, counter-viruses, and/or counter-worms [3, Sect. 14.6])! Finally, observe that our formal definition is, in contrast to [4]'s practical one, not formulated in terms of the maliciousness of the cause of its harmful effects (dysfunction due to harmful intention) but only in terms of the harmfulness of its effects (dysfunction due to formal

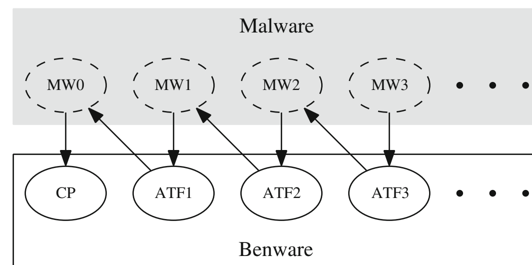

Fig. 1 Software systems

damage[12]). Malicious intention, as opposed to its harmful effect, is not generally directly observable. We believe this *causal indifference* to be an advantage of our definition due to the resulting gain in generality and thus applicability. For example, our definition should even be applicable to the *biological* domain with suitable definitions of correctness (cf. the predicate **correct**) for biological systems (and their chemical sub-systems) such as bacteria, biological viruses, prions, and the like. Possible biological notions of correctness could be liveness (the existence of a metabolism), healthiness (the existence of a normal metabolism), reproductiveness, etc. See [22] for a review of biological models that, due to their computational nature, are suitable for our approach.

Definition 6 (Anti-malware) A software system s is *anti-malware* by definition if and only if s damages no benware (safety)[13] and s neutralises[14] malware (effectiveness). Formally,

$$\text{antimal}(s) \quad :\text{iff} \quad s \models \neg\exists\mathbf{D}(\mathsf{BEN}) \text{ and there is } s' \text{ s.t. } \text{mal}(s') \text{ and not } \text{mal}(s(s')),$$

where $\mathsf{BEN} := \mu M(\forall\mathbf{D}(\exists\mathbf{D}(M)))$.

Observations analogous to those about software incorrectness under Definition 1 can be made about malware neutralisation here. Additionally, observe that the concept of anti-malware is defined *with* rather than *within* MalLog. Neutralisation transcends damage.

Definition 7 (Medware) A software system s is *medware* by definition if and only if s damages no benware (safety) and s repairs benware (effectiveness). Formally,

$$\text{med}(s) \quad :\text{iff} \quad s \models \neg\exists\mathbf{D}(\mathsf{BEN}) \land \exists\mathbf{R}(\mathsf{BEN}).$$

Fact 3 *1. Anti-malware is benware.*
(Anti-malware, being non-damaging to benware, is not malware.*)*
2. *Malware may damage anti-malware.*
(Damaging anti-malware is part of the anti-terror force. Non-damaging anti-malware is part of the civil population[15]*. See Figure 1.)*
3. *Anti-malware does not necessarily damage malware.*
(Neutralisation does not imply formal damage in case of malware mal- and/or under-specification.)
4. *Medware is benware.*
(Medware, being non-damaging to benware, is not malware.*)*
5. *Malware may damage medware.*[16]
(Damaging medware is the medical staff of the anti-terror force; and non-damaging medware is the medical staff of the civil population.)
6. *Medware repairs no malware.*[17]
(Otherwise medware would indirectly damage benware, yet by definition medware damages no benware—not directly nor indirectly.)
7. *Malware may cause benware to become malware.*[18]
(By definition, malware transforms software systems!)

3 Application

3.1 Scope

The manifold concrete manifestations of malware either are self-replicating, in which case affection is self-sufficient, or are in need of a host program for their replication, in which case affection needs infection. (See [23] for a classification of different degrees of hosted replication.) By definition, self-replicating manifestations of malware are called *worms* (cf. [24] for an informal taxonomy), manifestations of malware that need a host for their replication are called *viruses* (cf. [15] and [21] for formal classifications), and malware-carrying (e.g., a *rootkit* payload [3, Sect. 2.3.16]) pieces of malware are called *Trojan horses* [3, Sect. 2.3.4]. Since our definition of malware is indifferent to *how* affection takes place but is heedful to *that* it takes place to the extent that affection implies incorrectness, our definition exhausts all manifestations of malware on the condition that the specification of software correctness be exhaustive.

3.2 Tasks, tools, and techniques

MalLog is a theoretical tool for the practical *detection* of malware in software systems by means of model-checking. Two other theoretical tools related to MalLog are ordering relations for the *comparison* of these systems by means of equivalence- and refinement-checking, and characteristic formulae for their *classification* (including their *naming*).

When defining ordering relations on software systems, we can choose a *declarative* formulation in terms of *what* can be said about the systems in a given language (here Φ), or an *operational* formulation in terms of *how* the systems

[12] We stress that it is not the effect on a software system s as such (such as the formatting of a hard disk) but *formal damage*, i.e., the effect on the truth of the predicate **correct** (defining [the circumstances of the] harmfulness [of the formatting]) that matters.

[13] no friendly fire.

[14] Damage is insufficient!

[15] non-violent self-defence.

[16] The Red Cross needs body guards.

[17] Medware is, as opposed to the Red Cross, not neutral.

[18] defection to the dark side of the force, or, according to [4], *contamination*.

behave (here *behaving* means *damaging and/or repairing*). However, the two formulations are ultimately equivalent in the sense of Theorem 1.

Definition 8 (Comparing software systems with orderings)

- For all $s_1, s_2 \in \mathscr{S}$,
  - s_2 *refines* s_1 w.r.t. Φ, written $s_1 \sqsubseteq_\Phi s_2$, :iff for all $\phi \in \Phi$, if $s_1 \models \phi$ then $s_2 \models \phi$
  - s_2 *is equivalent to* s_1 w.r.t. Φ, written $s_1 \equiv_\Phi s_2$, :iff $s_1 \sqsubseteq_\Phi s_2$ and $s_2 \sqsubseteq_\Phi s_1$
  - s_2 *can simulate* s_1, written $s_1 \sqsubseteq s_2$, :iff for all $s_1' \in \mathscr{S}$,
    1. if s_1 damages° s_1' then there is $s_2' \in \mathscr{S}$ such that s_2 damages° s_2'
    2. if s_1 repairs° s_1' then there is $s_2' \in \mathscr{S}$ such that s_2 repairs° s_2'.
- For all $S \subseteq \mathscr{S} \times \mathscr{S}$,

$$\mathscr{O}_{\sqsubseteq}(S) := \{(s_1, s_2) \in S \mid s_1 \sqsubseteq s_2 \text{ and } s_2 \sqsubseteq s_1\}.$$

- $\approx :=$ the greatest fixpoint of $\mathscr{O}_{\sqsubseteq}$
  $= \bigcup\{\, S \mid S \subseteq \mathscr{O}_{\sqsubseteq}(S)\,\}$,
  by Knaster-Tarski ($\mathscr{O}_{\sqsubseteq}$ is monotonic).

The informal meaning of $s_1 \approx s_2$ is that s_1 and s_2 are the same as far as the underlying relations are concerned, and so when it comes to model-checking, one can be substituted by the (hopefully smaller) other. $s_1 \equiv_\Phi s_2$, on the other hand, says explicitly that s_1 and s_2 satisfy the same formulae. Hence $\approx$ and $\equiv_\Phi$ are actually the same relation. Note that any standard *bisimilarity* relation, such as $\approx$, when restricted to a finite domain, such as the set of all *actual* (those that exist now)—as opposed to *potential*—software systems, is computable in polynomial time (cf. [25, p. 274] and [26]).

Definition 9 (Classifying software systems with characteristic formulae) Let $S \subseteq \mathscr{S}$, $s \in S$,

$$D(S, s) := \{s' \in S \mid s \text{ damages}^\circ\, s'\},$$
$$R(S, s) := \{s' \in S \mid s \text{ repairs}^\circ\, s'\},$$

and $M_s \in \mathscr{M}$. Then, the *characteristic formula* $\chi(s, S)$ of the software system s w.r.t. S is the solution of the equation system

$$M_s \overset{\boldsymbol{\nu}}{=} \forall\mathbf{D}(\textstyle\bigvee_{s' \in D(S,s)} M_{s'}) \wedge \forall\mathbf{R}(\bigvee_{s' \in R(S,s)} M_{s'}) \wedge [\bigwedge_{s' \in D(S,s)} \exists\mathbf{D}(M_{s'})] \wedge [\bigwedge_{s' \in R(S,s)} \exists\mathbf{R}(M_{s'})],$$

(where $\bigvee \emptyset := \bot$ and $\bigwedge \emptyset := \top$) which can be obtained in a standard way [19], by translating each equation $M^i \overset{\boldsymbol{\nu}}{=} \psi^i(S)$ into a formula $\boldsymbol{\nu} M^i(\psi^i(S))$ and recursively substituting these formulae for the corresponding free variables in the first formula $\boldsymbol{\nu} M_s(\psi_s(S))$.

Fact 4 *1. For all $s \in \mathscr{S}$, $\chi(s, \mathscr{S}) \in \Phi$.*
(By Corollary 1.2, conjunction and disjunction in χ is finite modulo $\Leftrightarrow$.)
2. For all $s \in \mathscr{S}$, $s \models \chi(s, \mathscr{S})$.
(By construction of χ.)

The characteristic formula of a system can be seen as the way the system looks as viewed by the logic: it captures all the logically expressible behaviour of the system, but contains no other information. Characteristic formulae are useful for classifying software systems including malware and its derivatives into equivalence classes, as states the following standard characterisation result leveraged from Fact 1.3.

Theorem 1 *For all $s, s' \in \mathscr{S}$,*

$$s \equiv_\Phi s' \quad \text{iff} \quad s \approx s' \quad \text{iff} \quad s \models \chi(s', \mathscr{S}).$$

That is, (1) equivalence w.r.t. the language (a set) Φ of MalLog, (2) bisimilarity in MalLog, and (3) satisfaction of a (single!) characteristic formula coincide.

Corollary 2 (Normal form) *For all $\phi \in \Phi$,*

$$\phi \Leftrightarrow \bigvee_{\substack{s \in \mathscr{S} \text{ and} \\ s \models \phi}} \chi(s, \mathscr{S}).$$

Again by Corollary 1.2, disjunction is finite modulo $\Leftrightarrow$.

3.3 Examples

We discuss a few examples of recent (in)famous pieces of malware and their harmful effects on software systems and their users, in order to illustrate the spirit of our approach. Each discussion concludes with the derivation of a typical, necessary condition for the correctness of the affected software system. The condition represents a general lesson learnt from the harmful attack. The conjunction of all necessary conditions is an illustrative example of an approximative definition of the predicate **correct**. Definite definitions of **correct** however are not our business. Given that harmful attacks on software systems are falsifications of necessary conditions for the correctness of those systems, correctness really is the responsibility of the system designers. Finally, our discussions do not mention malware code: in our approach, it is possible to discuss malware abstractly, i.e., at the level of its harmful effects and the lessons learnt. Hopefully, such lessons will eventually be expressed as correctness conditions, as a matter of course.

Melissa The Melissa virus was detected in March 1999 [27, Sect. 5.5]. It could cause heavy overload to email servers. Concretely, Melissa infected Windows Word documents and spread via Microsoft Outlook email attachments. The spreading of the virus was triggered by opening an infected attachment. On opening the attachment, the virus would write itself

to the local hard disk, and, when Microsoft Outlook was present in the infected computer, send messages with an infected attachment to addresses in the user's address book. A harmful side-effect of the write operation was to violate the intended typing of a Microsoft Word template file. Hence, a necessary condition for the correctness of a file system is that always all files be well-typed.

CodeRed The CodeRed worm was detected in July 2001 [3, Sect. 10.4.3]. It could cause Microsoft's IIS web server under Windows 2000 to become hijacked. Concretely, CodeRed usurped the exception handling of the web server by injecting its own code into the stack and heap memory of the affected computer. The code injection could take place because the worm would make an invalid request to the web server, which the server would handle nevertheless. The request was invalid because it contained two pieces of invalid data. The first piece was data whose encoding was invalid; the second piece, actually the worm's main code, was data whose mere existence as a request body was invalid. The server would misinterpret the invalid encoding by writing the worm's launch code beyond the limits of an exception frame in the stack. Further, the server would write the worm's main code into the heap. The buffer-overflow in the stack would trigger an exception that would interpret the worm's launch code in the stack by transferring control to the worm's main code in the heap—without return. Hence, a necessary condition for the correctness of a web server is to accept only valid incoming requests.

Slammer The Slammer worm was detected in January 2003 [3, Sect. 10.4.5]. It could cause large-scale denial of service in the Internet. Concretely, Slammer affected Microsoft SQL by creating a buffer overflow in the stack-based parameter passing mechanism of an SQL server function. The mechanism failed to enforce that only strings of the specified maximal length were effectively passed as parameters. As a result, the worm would get passed through as a parameter and eventually get control of the computer. By definition, a necessary and sufficient condition for the correctness of an implementation (the police so to say) is that it enforces its specification (the law so to say)!

Opener The Opener virus was detected in October 2004 [27, Sect. 5.11]. It could cause a computer under Macintosh OS X to become remote-controlled against the user's will. Concretely on an infected computer, Opener would turn the local firewall off and the file sharing on, and install various other pieces of malware that would try to steal confidential information and violate the user's privacy. Hence, a necessary condition for the correctness of the operating system is that the system preferences cannot be modified without the user's permission.

Conficker The Conficker worm was detected in November 2008, and has not been neutralised so far [28]. It can cause a computer under the Windows operating system to become a component of a remote-controlled botnet against the user's will. On an infected computer, Conficker causes a buffer overflow in which harmful excess code is executed by the operating system. The excess code downloads more code that hijacks the server services of the operating system, in order to update and spread the worm via the network. What is more, variant code inhibits also the security services of the operating system and connections to anti-malware websites. Hence, a necessary condition for the correctness of the operating system is to be always free of buffer-overflows.

4 Conclusion

4.1 Assessment

Particular (general) detection of computer viruses being (un)decidable [16], the existence of computer viruses induces an *arms race* between, on the one side, the malware community (the black-hat hackers) and, on the other side, the anti-malware community (the white-hat hackers).

Our approach to the definition of malware confines this arms race to software systems engineering. Therein, formal specification and verification, as well as validation are the weapons of the white-hat hackers, who must say more and more *what* must and *what* must not be. We can assist the white-hat hackers in this task with the following, iterative methodology, which is in the spirit of *counterexample-guided abstraction refinement* [29]:

1. define or refine the predicate **correct** for the chosen target domain of software systems (choose your favourite school of expression[19])
2. apply our framework to the domain (if **correct** is decidable then "apply" means "run")
3. if, as a result, a system does not qualify formally as malware that intuitively should, go to 1.

Metaphorically speaking, the white-hat hackers act as the white blood cells of the Internet's digital immune system [3, Sect. 15.6], and our methodology is intended to be a possible support for deploying immune response.

We reduce malware *detection* to (dynamic) software verification and validation (the sandbox) based on modal logic

[19] Except MalLog itself! Such a choice would confuse language and meta-language, and cause the concept of malware to be non-well-defined. An example is to stipulate that a necessary condition for correctness be to be not malware.

for arbitrary software systems. Our approach evidences our conviction that the technological *security* of software systems can be indirectly reduced to their *correctness*. Given that harmful attacks on operating systems are falsifications of necessary conditions for the correctness of the operating system, anti-malware production is the responsibility of operating system producers. Other approaches in a similar spirit are: [30] based on algebraic specification for assembly software systems, [31] based on static program analysis (control-flow graph extraction) for arbitrary software systems, and [32] based on abstract interpretation.

Three major advantages of our approach are: *generality*, *genericity*, and *safety*—all thanks to abstractness. Our approach is general and safe because it focuses on what malware effects but abstracts from how malware actually and potentially does so. In particular, our approach is *hacker-safe* in the sense that it does not enable hackers to derive recipes for how to actually construct malware. Our approach is generic in the sense that different generations of MalLog (and thus malware classifications) can be obtained by redefinitions of damages° and repairs°. Finally, our approach also explicates the ultimate futility of the arms race between the black-hat and the white-hat hackers. Technology is not an end, worse, technology is a means to the race: it is a means to a potentially infinite ascending chain of mutual damage leading to nothing but software systems of ever increasing (specification and thus implementation) complexity, and hence decreasing usability (cf. Fig. 1). Fortunately in practice, the existence of resource bounds implies the existence of an equilibrium of mutual damage.

4.2 Future work

The items on our agenda for future work are:

1. to quantify with *measure* the futility—and thus predict the practical equilibrium—of the arms race between the black-hat and the white-hat hackers
2. to reconsider the class of software systems $\mathscr{S}$ as the class of software system (initial or present) *states*, and to introduce temporal modality into MalLog in order to study *malware evolution* and the *co-evolution* between malware and its derivatives
3. to meet in the middle [21]'s bottom-up approach with our top-down approach.

A possible simplification of MalLog suggested by G. Bonfante is to define both damage and repair in terms of a common relation (suggested by an anonymous reviewer), say **ensures**. Assume that there is $1 \in \mathscr{S}$ s.t. for all $s \in \mathscr{S}$, $1(s) = s$. Then,

$$\begin{aligned} s \ \textbf{ensures}\ s' &:\text{iff } \textbf{correct}(s(s')) \\ s \text{ damages } s' &:\text{iff } 1 \ \textbf{ensures}\ s' \text{ and not } s \ \textbf{ensures}\ s' \\ s \text{ repairs } s' &:\text{iff not } 1 \ \textbf{ensures}\ s' \text{ and } s \ \textbf{ensures}\ s'. \end{aligned}$$

Further, the following two variations of MalLog suggested by S. Pradalier are conceivable:

1. the generalisation of damages° from chains to *trees* of damage, which induces a notion of *contribution* to damage with as tree root the damaged system and as tree leafs the contributing systems
2. the addition of a ternary accessibility relation for formal *vaccination*

   $$\begin{aligned} s \text{ vaccinates } s' \quad \text{against } s'' \quad &:\text{iff} \\ &s'' \text{ damages } s' \text{and not } s'' \text{ damages } s(s') \end{aligned}$$

   with the corresponding binary modality for the definition of what could be called *vacware* (for *vac*cinating soft*ware*, or "vaccine" for affected software).

Finally, we could conceive a variant of MalLog in which correctness is rendered as an atomic proposition $\mathsf{correct}$ so that

$$[\![\mathsf{correct}]\!] := \{ s \in \mathscr{S} \mid \textbf{correct}(s) \},$$

and composability is rendered as modality:

$$\begin{aligned} \|\forall\mathbf{C}(\phi)\|_{[\![\cdot]\!]} &:= \{ s \in \mathscr{S} \mid \text{ for all } s' \in \mathscr{S}, s'(s) \in \|\phi\|_{[\![\cdot]\!]} \} \\ \exists\mathbf{C}(\phi) &:= \neg\forall\mathbf{C}(\neg\phi) \\ \|\overline{\forall\mathbf{C}}(\phi)\|_{[\![\cdot]\!]} &:= \{ s \in \mathscr{S} \mid \text{ for all } s' \in \mathscr{S}, s(s') \in \|\phi\|_{[\![\cdot]\!]} \} \\ \overline{\exists\mathbf{C}}(\phi) &:= \neg\forall\mathbf{C}(\neg\phi). \end{aligned}$$

One could then investigate the definability of *damage and repair as modality* in terms of *composability as modality* in the above or other (e.g., binary) forms, and fixpoint operators in the form of Definition 3.

Acknowledgments The first author thanks Jean-Luc Beuchat, Guillaume Bonfante, Johannes Borgström, Rajeev Goré, George Davida, Olga Grinchtein, Ciro Larrazabal, Mircea Marin, Lawrence S. Moss, Prakash Panangaden, Sylvain Pradalier, Daniel Reynaud-Plantey, Vijay Varadharajan, and Matt Webster for delightful discussions.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

1. Filiol, E., Helenius, M., Zanero, S.: Open problems in virology. J. Comput. Virol. **1**(3–4) (2006)

2. Kramer, S., Bradfield, J.C.: A general definition of malware. presented at the Workshop on the Theory of Computer Viruses (2008)
3. Szor, P.: The Art and Craft of Computer Virus Research and Defense. Addison-Wesley, Boston (2005)
4. Brunnstein, K.: From antivirus to antimalware software and beyond: another approach to the protection of customers from dysfunctional system behaviour. In: Proceedings of the National Information Systems Security Conference (1999)
5. Virus Encyclopedia. http://www.viruslist.com/
6. European Expert Group for IT-Security. http://www.eicar.org/
7. Information Warfare Monitor. http://www.infowar-monitor.net/
8. The Information Warfare Site. http://www.iwar.org.uk/
9. Clarke, E.M. Jr., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
10. Bergstra, J.A., Ponse, A., Smolka, S.A.: (eds.) Handbook of Process Algebra. Elsevier, New York (2001)
11. Fitting, M.: First-Order Logic and Automated Theorem Proving. Springer, New York (1996)
12. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press, Cambridge (2009)
13. Necula, G.: Proof-carrying code. In: Proceedings of the ACM Symposium on Principles of Programming Languages (1997)
14. Filiol, E.: Les virus informatiques: théorie, pratique et applications, 2nd edn. Springer, France (2009)
15. Adleman, L.: An abstract theory of computer viruses. In: Proceedings of CRYPTO, vol. 403 of LNCS (1988)
16. Cohen, F.: Computer viruses: Theory and experiments. J. Comput. Secur. **6** (1987)
17. Dowling, W.F.: There are no safe virus tests. Am. Math. Mon. **96**(9) (1989)
18. Jacob, G., Debar, H., Filiol, E.: Behavioral detection of malware: from a survey towards an established taxonomy. J. Comput. Virol. **4**(3) (2008)
19. Bradfield, J., Stirling, C.: Handbook of Modal Logic, chapter Modal Mu-Calculi. (2007)
20. Alberucci, L., Salipante, V.: On modal μ-calculus and non-well-founded set theory. J. Philos. Log. **33**(4) (2004)
21. Bonfante, G., Kaczmarek, M., Marion, J.-Y.: On abstract computer virology from a recursion theoretic perspective. J. Comput. Virol. **1**(3–4) (2006)
22. Fisher, J.A., Henzinger, T.A.: Executable cell biology. Nat. Biotechnol. **25** (2007)
23. Webster, M., Malcolm, G.: Formal affordance-based models of computer virus reproduction. J. Comput. Virol. **4**(4) (2008)
24. Weaver, N., Paxson, V., Staniford, S., Cunningham, R.: A taxonomy of computer worms. In Proceedings of the ACM workshop on Rapid malcode (2003)
25. Goranko, V., Otto, M.: Handbook of Modal Logic, chapter Model Theory of Modal Logic. (2007)
26. Dovier, A., Piazza, C., Policriti, A.: An efficient algorithm for computing bisimulation equivalence. Theor. Comput. Sci. **311**(1–3) (2004)
27. Salomon, D.: Foundations of Computer Security. Springer, Berlin (2006)
28. Lawson, G.: On the trail of the Conficker worm. Computer (2009)
29. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5) (2003)
30. Webster, M., Malcolm, G.: Detection of metamorphic and virtualization-based malware using algebraic specification. J. Comput. Virol. **5**(3) (2009)
31. Bonfante, G., Kaczmarek, M., Marion, J.-Y.: Architecture of a morphological malware detector. J. Comput. Virol. **5**(3) (2009)
32. Dalla Preda, M., Christodorescu, M., Jha, S.: A semantics-based approach to malware detection. ACM Transactions on Programming Languages and Systems **30**(5) (2008)
33. Blackburn, P., van Benthem, J., Wolter, F.: (eds.) Handbook of Modal Logic, Volume 3 of Studies in Logic and Practical Reasoning. Elsevier, Amsterdam (2007)