On Malfunction, Mechanisms, and Malware Classification

Giuseppe Primiero
Department of Philosophy, University of Milan

Frida J. Solheim
Department of Informatics, King's College London

Jonathan M. Spring
Department of Computer Science, University College London

November 14, 2018

Abstract

Malware has been around since the 1980s and is a large and expensive security concern today, constantly growing over the past years. As our social, professional and financial lives become more digitalized, they present larger and more profitable targets for malware. The problem of classifying and preventing malware is therefore urgent and it is complicated by the existence of several specific approaches. In this paper, we use an existing malware taxonomy to formulate a general, language independent functional description of malware as transformers between states of the host system and described by a trust relation with its components. This description is then further generalised in terms of mechanisms, thereby contributing to a general understanding of malware. The aim is to use the latter in order to present an improved classification method for malware.

1 Introduction

In the era of cyber-physical systems and the Internet of Things, miscomputation is an urgent issue for computer scientists, technologists and laymen alike. The situation is worse because miscomputations can be produced by errors, but also induced by targeted attacks. A full description of miscomputations requires considering their different layers: from design, through specification and implementation, up to use. [16] offer such an analysis, considering the different Levels of Abstraction and the agents involved at each such level.

The more focused task of identifying and preventing malfunctions in the software of a computing system, is one objective within the computer correctness literature. This large research area includes formal verification by model

checking and theorem proving; formal methods with dedicated systems like separation logic [17, 35]; and empirical analyses like testing and fuzzing [8]. These contribute to the common aim of obtaining more reliable and secure systems.

In [15], the authors perform a conceptual analysis of the problem of miscomputation, extending the existing literature in the philosophy of technology on the problem of misfunctioning for non-computational artefacts. This tradition has already played an important role in providing taxonomical formats and policy advice on errors of functioning in the technological context. For computational artefacts, and for software in particular, a similar analysis is possible but requires specific qualifications: a software token dysfunctions when either does not (sometimes) or cannot (ever) do what it is supposed to; a software token misfunctions when it may do what it is supposed to but, at least occasionally, it also yields some unintended and undesirable effects. When software is understood at the type level (i.e. not as individual instances of running programs, but as the equivalence class of such programs), it may misfunction in some limited sense, but cannot dysfunction. This analysis both clarifies the level of abstraction at which correctness problems need to be tackled, as well as illustrates the extent of different qualifications of software errors.

These analyses rely on software with specifications defined by a set of well-defined functions, and the assumption that errors are *unintended* interruptions of such functionalities, due to programming or designing mistakes. Alternatively, this paper focuses on software whose *intended* function and use is precisely the temporary or indefinite suspension of other systems' functionalities: hence not *bugs*, but rather *malware*. Stealth malware refers to a large class of *malicious software* [29, p.105]:

Definition 1 (Malicious Software) *Software that harmfully attacks other software, where to harmfully attack can be observed to mean to cause the actual behaviour to differ from the intended behaviour.*

More precisely, at code level one speaks of malicious logic [44]:

Definition 2 (Malicious Logic) *Hardware, firmware, or software that is intentionally included or inserted in a system for a harmful purpose.*

Malware growth over the past years has been constant, reaching over 600,000,000 units in 2017 [3], including the following types:

- virus, a malicious software characterised by a replicating structure affecting non-mobile files, requiring user action to propagate, see [48];
- worm, identified as self-propagating across networks, exploiting security or policy flaw, see [50];
- trojan, a type of malware that is often disguised as legitimate software, but has a backdoor, used to gather information or to damage software, see [42, p. 9];

- spyware, software that secretly monitors and collects information, such as keystrokes and screen dumps, and sends it to a third party without the user's knowledge or consent, see [42, p. 10];

- rootikit, a set of binaries, scripts and configuration files that allows someone covertly to maintain access to a computer so that he can issue commands and scavenge data without alerting the system's owner, see [39].

It is essential to characterise misfunctioning and dysfunctioning behaviours in computational systems that are induced by a program that 'deliberately tries to conceal its presence in the system', [14, p.3]. The problem of classifying malware has been ongoing since the end of the 1980s. Modern efforts seek classifications that are complete and exhaustive. However, how malfunctions induced by hidden programs are defined, structured, analysed and understood requires more conceptual work, providing more solid strategies for malware analysis and incident response. In particular, existing classifications and languages do not consider systems in their generality, but rather aim at detailed description of objects. This strategy leads to difficulty with generalising properties of objects in relation to the type of malfunctioning they induce and common traits of errors. This aim is not purely conceptual: abstract, automatable support for deciding on the nature of potential malware is a goal of the security community. In this context, one is missing a general decision procedure that can be implemented on top of existing tools. Such a procedure would require well-defined, concise and conceptually clear tools that are feasible for formal translation. In this light, the present work has several aims:

1. we explicate an existing taxonomy of malware – presented in [41] – in terms of a general property of trust for the relation between a system and its components;

2. we use this basic property to clarify how malware act as transformers from functional to (several types of) non-functional systems;

3. we revisit the mentioned theory of software malfunctioning from [15] in the light of the above results;

4. we contribute, through the above, to a general, abstract mechanistic explanation of malware operations, complementing the approach to malware attacks presented in [46], and illustrate how this can provide useful indications for an improved and simplified classification of malware.

We therefore will offer a functional reading of malware types in terms of what type of damage they induce and use mechanisms to support it conceptually and (eventually) formally.

The present work intersects theoretical research in security and philosophy of computing. For the latter, our work is motivated by the need of establishing criteria of functionality for computational systems and their limits, in line with the analysis offered in [16, 15]. This task originates in the philosophy of technology

and earlier still in the philosophy of science. In the security domain, our goal is to provide a level of abstraction that is amenable to both logical decision-making as well as heuristics for mechanism discovery involving malware. As argued by [38], reasoning in computer science benefits tremendously when the logical model and the scientific model (e.g., the mechanistic model) can be genuinely aligned. In particular, malware categorization through specific languages is extensive and it offers a detailed identification of all properties of malware artifacts. However, because of this level of detail, it cannot explain what makes certain software to be malware: existing languages do not provide any general qualification of the dynamic and static properties of the objects under investigation that can robustly be used to contribute to our general knowledge about malware. Any such generalizations are left as purely manual, human tasks. It is desirable, given the scale of malware samples, to provide decision support to malware analysts in the form of automatable suggestions about categorizing malware. This is a longstanding goal of the security community, and deserves a fundamental new approach, as the detail-oriented attempts by practitioners have not yet yielded such a decision support framework. Our contribution will lay the foundation for a language that can appropriately abstract decision-making. This aim requires a more concise definition of categories that can be translated into operational tools, for example by rendering the concepts essential for a logical translation. A logical translation should also, ultimately, align with the practitioner's mechanistic explanation heuristics for malware. Hence, a more technical aim of this paper is to prepare for a formal logic of malware as entities and functions. A first aproach of this type is offered in [29]. Our treatment of malware classification should be able to provide a more solid basis to extend their formal understanding. We leave this task to future research.

The structure of the paper is as follows. We first provide an overview of related work in Section 2. In Section 2.5 we overview the existing languages for malware classification. We then introduce in Section 3 an existing taxonomy of malware from [41] and provide clarifications on how it applies to several examples. In this Section, we further qualify a malware ontology in terms of trust and offer a functional analysis of their behaviour as transformer from functional to (several types of) non-functional systems. In Section 4 we use this description to improve the analysis of malware as a mechanism, extending and improving previous work in [46]. We conclude with indications on how this conceptual description can facilitate the task of malware classification.

2 Related Work

This section brings together related work on philosophy of technology, information security, malware analysis, and malware classification all relevant to the present analysis. This presentation provides the conceptual frame to understand the present contribution.

2.1 Philosophy of Technology

The Philosophy of Technology has investigated at large the concept of function and its implications.[1] In this respect, the notion of malfunction is crucial. In [30], malfunction statements are intended as normative statements made about technical artefacts and related theories have to account for those. Possible approaches include a privative stand (a malfunctioning F is not an F) or a subsective one (a malfunctioning F is a subset element of F), see [26, 27]. In [49, ch. 2], a mechanistic approach is applied to the ascription of functions in engineering. Mechanistic explanation is based on the identification of the phenomenon to be explained, its decomposition into entities and activities so as to identify the organization of them which produces the phenomenon, see [24]. This task is based on role function ascription, i.e. the description and representation of roles played by each entity and activity in the mechanism. Hence, for technical artefacts the mechanistic approach aims at explaining their functioning by identifying basic and complex functions, the entities performing them and how these are organised, see also [10]. [49] argues that role function alone is insufficient for the individuation of mechanisms in engineering, and it should be supported by behaviour function, where functions are specified in terms of I/O flows referring to specific physical behaviours and effect function, without reference to their requirements.

The notion of mechanism, even in its most simple understanding, is well-suited to the explanation of computing systems, namely as mechanisms whose function is to generate output strings from input strings and internal states, according to rules (expressed as computable functions), see [36]. The assumption at the basis of the present work is that exploiting the extensive and solid analysis in the definition of mechanisms originating in the philosophy of science and applying it in the philosophy of computing, in particular focusing on malware as mechanisms, can be helpful in their classification. This task extends an already started trend in applying this type of methodology in the field of computer security, by referring in particular to both complete behaviour and their effects. Our focus is, obviously, on the use of mechanism for the explanation of induced malfunctions.

2.2 InfoSec and Malware Analysis

The task of interpreting malware as mechanisms in order to facilitate and support their classification, should be understood in the context of the established area of information security. The intuitive meaning of security in this field is captured by the following definition [44, p. 265]:

Definition 3 (Security Architecture) *A plan and set of principles that describe*

(a) the security services that a system is required to provide to meet the needs of its users,

[1] For a general introduction see [21].

(b) the system components required to implement the services, and

(c) the performance levels required in the components to deal with the threat environment.

Our focus is on computer security,[2] often articulated in two main phases, monitoring and response:

Definition 4 (Computer Network Defence) *Actions taken to defend against unauthorized activity within computer networks. CND includes monitoring, detection, analysis (such as trend and pattern analysis), and response and restoration activities.*[3]

Definition 5 (Incident Response) *Actions taken to resolve or mitigate an incident [i.e., cyberattack], coordinate and disseminate information, and implement follow-up strategies to prevent the incident from happening again.*[4]

Malware analysis is one of many sub-fields within computer security and often, but not always, occurs as part of responding to a computer security incident. That is, a defender wants to figure out what a malware sample can do or has done to their system. Two fundamental approaches to malware analysis are available: static and dynamic:[5]

Definition 6 (Static Analysis) *Static Analysis consists of examining the malware without running it. This includes simple brief identification work, such as file sizes and fingerprinting (for example, to compare to lists of known-bad files), or it may include involved analysis of the structure of the file, disassembling its instructions to guess at their purpose.*

Definition 7 (Dynamic Analysis) *Dynamic Analysis involves running the malware and observing its behaviour. Such analysis is usually performed on a specially protected and instrumented system to study the malware more safely. The objective might be to simply observe the inputs and outputs during the malware execution, or it might take the more involved step of using special software (debuggers) or specially-designed reactive environments to extract additional information.*

Further analysis of malfunctioning software includes determining its propagation techniques.

[2] An organization might also have security violations in administrative, communications, personnel, or physical security, for example. Security is from the perspective of the system to be secured, i.e. there is not one absolute concept.

[3] NIST glossary entry: `https://csrc.nist.gov/Glossary/?term=5475`.

[4] See [2, p.3]. The goal is to handle the situation in a way that limits damage and reduces recovery time and costs.

[5] See e.g. [45, Ch.0].

2.3 Malware Propagation

Malware authors organize their efforts in to 'campaigns' [23], i.e. systematic sets of attacks to breach a system's security. A campaign is a collection of attacks that share important properties, such as their techniques, targets, or infrastructure [7]. There has been a tremendous amount of work within information security to pin down the various types of attacks and how they work, see, e.g. [22, 33, 44]. We focus on malware, which is a technical attack subverting the computer and is opposed to 'social engineering', in which the adversary achieves their goals via "e.g., blackmail, bribery, coercion, impersonation, intimidation, lying, or theft" [44]. Phishing, for example, is a type of impersonation and/or lying, albeit via a technical medium, that (by a strict definition) does not involve malware.

Despite a number of taxonomies proposed over the years (see next subsection), there is no consensus on what features of malware should be considered. For example, a malware attack can be usefully characterised by what vector is used to initiate the attack (e.g., the Internet, websites, local networks, USB drives, or email [1]). Alternatively, one can subdivide malware propagation techniques by whether human interaction is necessary.

2.4 Classifications

The problem of classifying malware is crucial and ongoing since the end of the 1980s.[6] Previous formulations of malware taxonomies (or of a significant subset of this family) have been built around malware functionality. For example, [50] defines a taxonomy of worms around the following elements:

- Target discovery: The mechanism by which a worm discovers new targets to infect. It includes scans, fixed lists, externally and internally generated target lists.
- Carrier: The mechanism the worm uses to propagate to the target. Propagation modes include self-carry, secondary channels, and embed in normal communication channels.
- Activation: The mechanism by which the worm's code begins operating on the target. Options include user action (either directly or indirectly), scheduled process activation, or self-activation.
- Payloads: Other actions that are not propagation that accomplish the author's goal. Payloads include non-functional actions (in turn complying with our definition of dysfunctioning), spam relays, HTML proxies, perform Denial of Service attacks, collect or damage data, manipulate cyber-physical systems, conduct further reconnaissance, and maintenance.

[6] For early classifications see [9] and [13].

Another taxonomical approach is to use behaviour; that is, to identify the action performed by the malware rather than its syntactic markers. Cohen [9] introduced two such approaches:

1. model the behaviour of legitimate programs and measure deviations from this reference. The complication in this approach is to reach a common general description of well-behaved programs;

2. model and detect suspicious behaviours. The problem here is that unknown malware can remain undetected as long as they use innovative methods.

[25] formulates a taxonomy dedicated to behavioural detection, based on program testing and divided into simulation-based and formal verification. For each, [25] illustrates data collection and monitoring conditions, interpretation, algorithm, definition of the behavioural model and signature generation. [25] then offers a list of behavioural detectors along these criteria. The main aspect that we preserve from behavioural detection is the classification of systems transformation induced by malware attacks in view of trust relations.

More ambitious malware classification endeavors appeared as the problem grew in both importance and complexity. A 2012 review identified multiple failings in experiment design surrounding malware, and suggested methodological improvements [40]. We endorse this work, but our focus is on methodology. In order to lay the ground for a general improvement on current methodology, we overview the most important among the existing specific languages for malware classification.

2.5 Languages

Malware detection and characterization requires determining behaviours and attributes, typically through static and dynamic analysis, as Section 2.2 discussed. During the 2010s, these processes became increasingly automated. As humans involvement declined, machine-readable languages have become practical necessities for malware classification. Current machine-readable languages generally do not support provable or verifiable reasoning about malware, which is what we are building towards in Sections 3 and 4. This section continues with a survey of existing languages. The challenge that will become clear is how to develop unambiguous definitions and characterizations of malware.

2.5.1 MMDEF

The goal of MMDEF[7] is to provide a format to share information relevant to anti-virus software, including metadata of benign files. Conceptually, knowing all benign files would be a tremendous advantage, as anything else would be malware. In practice, we cannot list all benign programs. However, making

[7] See `https://standards.ieee.org/develop/indconn/icsg/mmdef.html`.

```
<mmdefb
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="mmdef-v2"
xsi:schemaLocation="mmdefb-v1 mmdefb-1-0-schema.xsd">
 <subject md5="35ed51749a8987b8dcda050647f6c8d7" size_in_bytes="18087"/>
 <action_findings>
 <file pid="352" action="create" name="i1ru74n4.exe" normalized_path="csidl_system"/>
 <registry_key pid="352" action="write" hive="HKEY_LOCAL_MACHINE"
 key="Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders"
 value_name="Common Desktop" value_data="C:\Documents and Settings\All Users\Desktop"/>
 <process pid="440" parent_pid="352" action="create" filename="i1ru74n4.exe"/>
 <mutex pid="440" action="create" name="CTF.Asm.MutexDefaultS-1-5-21-1229272821-
1004336348-527237240-1003"/>
 </action_findings>
</mmdefb>
```

Figure 1: An example of MMDEF bundle

sure an anti-virus program does not block a benign file that is critical to the function of the system is still valuable. MMDEF provides structured reporting of items such as hashes, filenames, installation paths, signature information, and file versions. The aim is to provide a system for content creators and third-party providers to check the nature of files and ascertain whether they are potential malware [19]. In this regard, one might see MMDEF as a language for sharing information about software generally, not just malware.

Figure 1 presents an example of a MMDEF bundle.[8] The first lines declare the MMDEF file format. What follows is the subject file's MD5 hash, or fingerprint, and the file size to help with quick identification; details of a new file the subject creates, with the pid, name and normalized path; and the data the subject writes to an existing system file, suspiciously a registry key targeted by the action.

2.5.2 MAEC

MAEC [5], like MMDEF, distinguishes between static and dynamic elements in malware detection and characterization. MAEC distinguishes three separate dynamic elements, as Figure 2 indicates:

- *capabilities*: high-level, what the malware is capable of producing, addressing the abilities of groups of behaviours;[9]
- *behaviours*: mid-level, how the malware operates, addressing the purpose of groups of actions;

[8] See `http://grouper.ieee.org/groups/malware/malwg/Schema1.2/full_clean_file_example.xml`.

[9] It is interesting to note that MAEC capabilities where first called *mechanisms*.

- *actions*: low-level, malware system actions. Described without addressing intention. Actions can be viewed as syntactic analysis of the software's linguistic constructs, or as semantic analysis of what those constructs do when abstracted from language specifics.

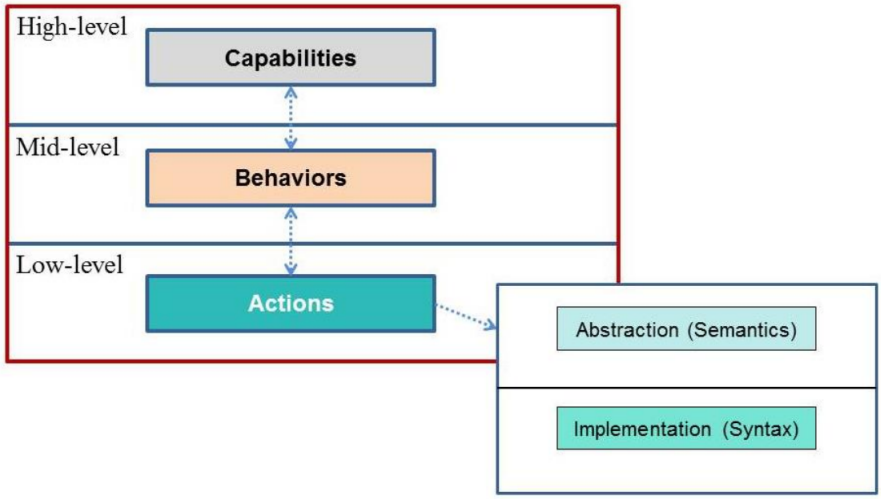

Figure 2: MAEC Bundle Overview [5]

MAEC also includes three types of static elements [5, pp.10-11]:

- *attributes*: a descriptive characteristic of malware. Attributes can be low-level Actions, mid-level Behaviours, the categories of the high-level Capabilities, or metadata.
- *objects*: a CybOX (Cyber Observable Expression, another Mitre language) entity with object details; for example, of a file, registry key, or process.
- *indicators*: information such as importance, author, and target information.

Figure 3 demonstrates a sample with actions and objects. The example is a Windows executable file. The description of the object states the name, size, and hash. The actions it has been observed to take are to create a file and write to process memory. The language provides fields for the names, types, associated objects, sub-actions, and the object created.

MAEC can capture various details during static malware analysis. Details range from static attributes of a binary file, such as on the packaging style and obfuscation techniques defeated, to interesting code snippets from manual reverse engineering [5, pp.16-17]. MAEC can also capture various details during dynamic analysis, at flexible levels of abstraction. Lower-level information

MAEC Bundle (Excerpt)

```
Bundle id="maec-example-bnd-1" schema_version="4.0.1" defined_subject="true"
content_type="dynamic analysis tool output"
  Malware_Instance_Object_Attributes
    Properties type="WindowsExecutableFileObjectType"
      File_Name=dg003_improve_8080_V132.exe
      Size_In_Bytes=196608
      Hashes
        Hash
          Type=MD5 type="HashNameVocab"
          Simple_Hash_Value=4EC0027BEF4D7E1786A04D021FA8A67F
  Actions
    Action id="maec-example-act-1"
      Name=create file type="FileActionNameVocab"
      Associated_Objects
        Associated_Object idref="maec-example-obj-1"
          Association_Type=output type="ActionObjAssocVocab"
    Action id="maec-example-act-2"
      Name=write to process memory type="ProcessMemoryActionNameVocab"
      Associated_Objects
        Associated_Object idref="maec-example-obj-1"
          Association_Type=input type="ActionObjAssocVocab"
  Objects
    Object id="maec-example-obj-1"
      Properties type="WindowsExecutableFileObjectType"
        File_Name=msvcr.dll
```

Figure 3: Stand-alone Bundle example (excerpt) with separate object

includes specific machine-like operations and code instructions. Higher-level information might be a malicious functionality, such as 'keylogging', that summarizes the purpose of myriad possible implementations [5, pp.16-17]. MAEC can capture information on the analysis process as well. For example, the language includes fields for findings, tools used, and the analysis environment. As such, MAEC permits the analysis of a malware instance to be described in a standard fashion and captured in a single document, the MAEC Package. In practice, the content of some MAEC fields is unstructured text; as such, MAEC is a combination of suggestions to humans and actual machine-readable language. Sections 3 and 4 jointly provide an improvement on the formulation of this MAEC bundle, demonstrating the practical import of an abstract, well-defined malware categorization.

2.5.3 STIX

STIX provides a mathematical graph to represent an attack. The graph is comprised of domain objects as nodes and relationships as edges. Examples (static) domain objects include attack pattern, campaign, course of action, identity, intrusion set, and report. Importantly, one of the static objects can be a MAEC object, as described in Section 2.5.2. Relationships are generic and associated to objects, e.g. the 'indicator' object has an associated 'indicates' relation. STIX objects subsume malware description objects, among others, and so is a

way of describing a higher-level type of knowledge than malware; the idea is to structure comprehensive information about a threat.

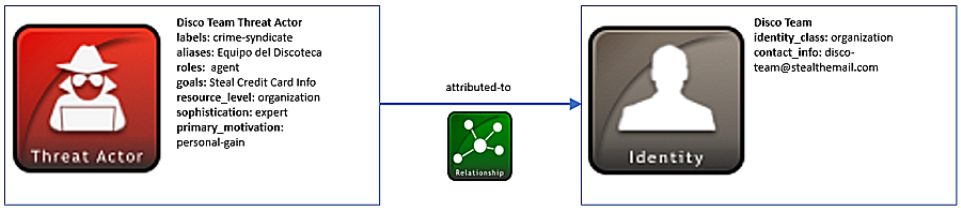

Figure 4: STIX Threat Actor profile

Figure 4,[10] shows a scenario representing a threat actor group named "Disco Team" using the Threat Actor SDO (STIX Domain Object). Information relevant to threat actors can be captured within this object; there are various other SDO for other information. This example uses the Identity SDO.

The difference between MMDEF, MAEC, and STIX is clear. MMDEF and MAEC provide classification which directly targets malware, although MMDEF focuses its approach on whitelists. STIX has a more general focus, on attacks, including e.g. the kill chain [23]. However, all of the available languages are both practical, specific, and operationally contingent. Our work is intended to offer a more general contribution, independent of the mentioned approaches and therefore useful to all.

3 Malware as Malfunction-inducing Artefacts

In order to clarify what types of malfunction malware may induce, one should first understand by what relevant kinds of attacks malware can be categorized. The distinction between worms, viruses, etc., is traditional in information security, but the distinction is not informative about exploitation methods. MAEC and MMDEF are not helpful in taxonomizing exploitation strategies, as they are documentation languages. And efforts such as the common weakness enumeration (CWE) are likewise too fine-grained and not quite on target. Therefore, we seek an orthogonal taxonomy of malware based on exploit strategy.

The taxonomy offered in [41] classifies malware into four ground types, based on the way the software enacts the exploitation and installation phases to attack the system. The purpose of the taxonomy is pragmatic; the four types would be detected and defended against very differently. We summarise the classification as follows:

- **Type 0**: an attack of the system limited to monitoring activities and possibly data leaks, without inducing any interruption of the system's

[10] See `https://oasis-open.github.io/cti-documentation/examples/identifying-a-threat-actor-profile`.

functionalities. While not actually "malware from the system compromise detection," Type 0 malware can still use or abuse features of the system to perform malicious actions [41, p. 2].[11]

- **Type I**: the attack occurs by breaking the integrity of constant resources, like in-memory code sections of the running kernel and/or processes. Type I malware is common and destructive, but is a tractable problem because detecting changes in things that are not supposed to change is feasible, though practically hard.

- **Type II**: breaks the integrity of dynamic resources, such as pointers in memory. Type II is a "much more dangerous and challenging" problem than Type I, in a practical sense, because it is nearly impossible to detect malicious changes in amongst all the regular changes that happen to dynamic resources [41, p.6]. However, like Type I, Type II malware attack the system itself.

- **Type III**: represent a conceptually different attack. The system software and data are themselves untouched; however, the malware inserts itself as a layer mediating access between the system and the hardware. It becomes a virtualization layer, which allows the malware to observe or change system behaviour essentially and arbitrarily. From this position, the malware is in principle invisible to the system, and must be detected and remediated from a different perspective. Such malware is more difficult to make and install than the other three types.

Note that, in [41], whether a resource is static or dynamic is based on its changeability from the perspective of the system under attack. So we do not need a complete classification of static and dynamic resources, we just need to be able to "divide [resources] to those which are (or at least should be) relatively constant ('read-only') and to those which are changing all the time" [41, p. 3].

For our purposes, this classification also groundly divides three sorts of malware, fitting well with the definition of malfunctioning software offered in [15], when this is applied to the target system. Recall from Section 1 that malfunctioning software (tokens) can be divided into those whose functionalities are (temporarily or indefinitely) interrupted (dysfunctioning software), and those presenting (additionally or in alternative) unintended functionalities (misfunctioning software). When we consider this behaviour as the result of malicious activity, we can map to the following sorting:

- **Type 0**: no compromise of target system functionalities (no side-effects, no dysfunctioning) but production of additional, unintended functionalities (misfunctioning)

[11] Writing in 2006, [41] marks this type as uninteresting. The more recent prevalence of ransomware, which uses normal system features to disrupt the user's tasks to extort money, indicates that Type 0 malware can nonetheless significantly harm an organization's security architecture.

- **Type I-II**: dysfunctioning creating software in different parts of the target system, possibly accompanied by misfunctioning.
- **Type III**: either dysfunctioning or misfunctioning of the system may occur because the malware has virtualized the whole system.

In order to explain the four types of malware, and then eventually characterize them individually, we need to associate more precisely malfunctioning (in the two forms here considered) to system properties that are affected by the malicious software. When considering malware, the most relevant system property is *computer security*. Computer security refers to "measures to implement and assure security services in a computer system, particularly those that assure access control service" [44, p. 74].[12] Computer security contributes to a security architecture that is essentially concerned with the protection of data from unauthorized disclosure (confidentiality), unauthorised modification (integrity) and unjustified limitation of functionalities (availability).

In line with standard terminology, we can adapt correctness (i.e. well-functioning) criteria to security criteria under control and establish these in the form of authorizations. Accordingly, system correctness is bound to security and therefore we assume that unless a system can be totally proven correct (an impossible task), it cannot be totally secure.

Definition 8 (Secure system) *A system S is secure if authorizations are defined for every component $c_i, \ldots, c_j$.*

Definition 9 (Partially secure system) *A system S is partially secure if authorizations are defined only on some components $c_i, \ldots, c_j$.*

Notice that this understanding of computer security as authorization completeness is necessary, but not sufficient to guarantee that no alterations of functionalities occur. First, security is not a fixed property: with every new component, a new authorization is required. But even if the system administrator can guarantee completeness is always obtained, by ensuring that an appropriate authorization is defined for each newly added component, another necessary property is *precision* with respect to valid authorizations for all components, see [43]. An imprecise system refers to one that does not identify a component for what it is, but rather for something else. Malicious software aims at establishing a trustful relation with its target system, to acquire certification: malware aims at finding hooks to avoid being blocked. Hence authorization imprecision can be better translated into a relation of trust. The notion of trust in a computational setting has received much attention in several fields and, accordingly, various definitions of trust are available. For the present purposes, we need a generic and abstract definition of trust as a property of the relation of access control between a software and its host system:[13]

[12] For a canonical perspective on defining access control, see [6].

[13] The following definition is formulated as a special case of the more general one provided in [37].

Definition 10 (Trust) *Given a first-order relation of access control between a software component c and its host system S functional to a I/O process according to the specification of S, if S can achieve the intended output through the task performed by c, and if S deems secure to delegate to c performing such task, then the relation has a second-order property of trust.*

While conceptually it is appropriate to say that the host system instantiates a trustworthy functional relationship with each of its components, for brevity we will often say that a software component c is trustworthy or trusted.

In this sense, systems can be imprecise according to two trust-related aspects. We believe such trust aspects could be formally connected to well-known notions from the information security literature used to define program correctness [31, p.125]: safety and liveness properties. We give an informal introduction to safety and liveness, but leave the formal translation for future work on a logic for malware:

- A safety property is one which states that something (usually something bad) will not happen. For example, the partial correctness of a single process program states that if the program is started with the correct input, then it cannot stop if it does not produce the correct output.

- A liveness property is one which states that something must happen. For example, the statement that a program will terminate if its input is correct.

The relationship between safety, liveness, and trust can be formulated roughly as follows. In a safety breach, an untrusted component is taken as a trusted one, thereby allowing that something which should not happen does in fact happen. In a liveness breach, a trusted component is taken as an untrusted one, thereby allowing that something which should happen does in fact not happen. In this sense, a liveness breach refers to cases of unjustified denial of service or access. This is the category that has been mainly analysed in [15] in terms of *malfunctioning software tokens*.

On this basis, the standard general definition of correct system can be formulated in view of safety and liveness:

Definition 11 (Correct system) *A system S is correct if it presents no safety or liveness breaches and authorizations are fully defined (per Definition 8) in accordance with the relevant security architecture.*

Definition 12 (Incorrect system) *A system S is incorrect if it presents a safety or liveness breach or authorizations are only partially defined (per Definition 9) in accordance with the relevant security architecture.*

We have now characterised the ontology of software systems by properties of security (by authorization), safety and liveness, with the latter ones identified by trust qualification of components. On their basis we can offer a tentative ontological definition of malware:

Definition 13 (Malware) *Malware refers to a software component inducing unjustified trust authorization for itself or denial of justified trust for another component.*

To link this abstract definition of malware, characterised in terms of authorizations and trust, with the taxonomy offered by [41], we note that malware of **Type 0** are misfunction inducing, while **Type I-II** are dysfunction-inducing and malware of **Type III** may induce either. Then a further qualification is possible in view of Definition 13:

Definition 14 (Malware of Type 0) *Malware of* Type 0 *refers to a software component that abuses the trust it has been granted by a system to induce misfunctioning in the system.*

Definition 15 (Malware of Type I-II) *Malware of* Type I-II *refers to software components acquiring unjustified trust from an incorrect system to induce dysfunctioning in the system.*

Definition 16 (Malware of Type III) *Malware of* Type III *refers to a software component that takes control of a system by mediating its trust relationships at a level of abstraction outside the system, usually 'closer' to the hardware. Such malware may induce dysfunctioning or misfunctioning.*

The reason to offer a definitional description of malware in terms of trust is to prepare for a better, more general and comprehensive approach to detection. Therefore, this approach can be qualified as a contribution to the tools of Computer Networks Defense (CND). In view of Incident Response (IR) practices, our aim is to bind the previous definitional description of malware to a corresponding functional qualification. Such an analysis can be given if we qualify the notion of safety breach by appropriate transformations of relevant system states through trust operations.

Let us consider a malware m as a software component inducing a transition $S \xrightarrow{\mathtt{m}} S'$ from a system S to a variant system S', where the former is considered before the successful deployment of the attack, and the latter after it. Qualifying m means now to associate the security statuses of S and S' (in terms of control) with their safety and liveness (i.e. in terms of correctness). We provide in Figure 5 the set of formal transitions obtained by these combinations.

`Transformation 1` considers a partially secure system S as by Definition 9 with authorization over some components; if a safety breach of S occurs induced by a malware m of **Type 0**, then the resulting transition according to m induces a misfunctional system S'. `Transformation 2` considers a partially secure system and a breach of S by a malware m of **Type I** or **Type II**; it results in a dysfunctional system S'. `Transformation 3` considers a system S which is controlled by a malware m of **Type III** at a lower level, e.g. a rootkit, and which therefore can be considered not secure; it results in a misfunctional or dysfunctional system S'.

Transformation 1:

$$\texttt{partiallysecure(S)} \xrightarrow{\texttt{TO}} \texttt{misfunctional(S}')$$

Transformation 2:

$$\texttt{partiallysecure(S)} \xrightarrow{\texttt{TI/II}} \texttt{dysfunctional(S}')$$

Transformation 3:

$$\neg\texttt{secure(S)} \xrightarrow{\texttt{TIII}} \texttt{mis/dysfunctional(S}')$$

Figure 5: Transformations

This approach allows us to explain malware attacks by understanding displayed behaviour, identifying the transformation at stake and thus informing an incident response strategy (see, e.g., [2]). This ontological and functional analysis of malfunction inducing software offers the chance to reconsider the definitions of dysfunctioning and misfunctioning software (token and type) from [15] and to reformulate them in view of malfunctioning *inducing* software, i.e. malware. In this attempt, we can make use of all the tools developed in this section, concerning safety, liveness, and computer security.

Recall from [15] that software dysfunctioning and misfunctioning at the token level t (i.e. an instance of a program) is considered impossible when analysed in isolation: in the first case, because a program instance cannot be less reliable or effective in performing its function F compared with other tokens of the same type T *independently* of the supporting hardware used to run it; in the second case, this is due to the fact that all instances will inherit the same software design D of the corresponding type T. This obviously does not hold for induced malfunctioning, which most of the time targets individual instances of software. Instead, it is perfectly possible to associate individual software instances with misfunctioning and dysfunctioning behaviours, due to an appropriate breach:

Thesis 1 (Induced misfunctioning of a Software Token) *A software token t can suffer induced misfunction if it is hosted on a system S following a malware transformation m and it produces undesired side-effects on such a system compared to the design D of t in common with other tokens of the same type T hosted on secure systems.*

Thesis 2 (Induced dysfunctioning of a Software Token) *A software token t can suffer induced dysfunction if it is hosted on a system S following a malware transformation m and t has becomes less reliable or efficient in performing its function F compared with other tokens of the same type T.*

Moreover, in [15] it was argued that misfunctioning can be assessed comparatively at the type level, where two software with similar functionalities can

be compared as tokens of a more general category of software (e.g. LibreOffice Writer and MS Word in the category of word processors). A similar comparison can be made for induced misfunctioning by malware, when system protection is taken into account and which can be viable for a type of software but not for another comparable one:

Thesis 3 (Induced Misfunctioning of a Software Type) *A software type T_x can suffer induced misfunction comparatively, when any of its tokens t_x hosted on a system S following a malware transformation m is subject to side-effects which are not produced on the same system S by tokens t_y of a (possible) type T_y, where T_x and T_y are tokens of a higher order type T_0, if T_x is exposed to a safety breach from which T_y is protected (e.g. because a patch exists).*

The above analysis shows that it is correct to talk about induced malfunctioning for software, both in terms of dysfunctioning and of misfunctioning. Induced malfunctions count as genuine cases of misfunction and dysfunction, in which malicious software manipulates trust relationships, or violates safety or liveness properties.

4 Malicious Software as Mechanism

In the Philosophy of Science, the mechanistic approach is one useful tool for explaining and understanding phenomena, see, e.g., [11]. The mechanism discovery literature is also a rich source of heuristics for generating hypotheses and how to test them, see, e.g., [4, 12]. These benefits begin from the notion of what a mechanism is, so that we know what an explanation should contain or what we are missing that needs to be discovered. A recent synthesis dubbed 'minimal' proposes [18]:

> "[a] mechanism for a phenomenon consists of entities (or parts) whose activities and interactions are organised so as to be responsible for the phenomenon".

In this larger context, the attempt at using mechanisms to explore and understand information security is very recent. In [46], the authors take a practitioner view and use mechanistic modeling of computer security incidents to clarify and improve the existing model. In [47], the authors take a philosophical view, synthetise the existing mechanisms literature into a view of how a mechanistic approach builds general knowledge, and argue that good practices in information security already exemplify building general mechanistic knowledge, though the process is painstaking. The final step in the present contribution is to connect our definition of malware as malfunction-inducing artefacts to a mechanistic explanation of malware attacks. The goal is to implement mechanistic explanation on malware attacks, so as to facilitate malicious software classifications.

Malicious software classifications have several synergies with mechanism discovery. The two broad categories of methods, static and dynamic analysis introduced above in Section 2.2, approach the problem of classification rather differently. In this context, we leverage the distinction between code as data and code as program. [47] notes that malware analysis makes use of the etiology of the malware's mechanism. Here, we will focus on the instructive properties of the entities and activities of the mechanism instead. [20] proposes a connection between the practical security task of differentiating whether information is a data element or executable code with the mechanism discovery task of determining whether to model an aspect of a phenomenon as an entity or an activity. Our descriptions shall illustrate the difference between static and dynamic analysis in regards to mechanism discovery heuristics related to entities and activities, respectively.

1. *static analysis*: In this style of analysis, code is treated and understood as an *entity*. Typically the analyst searches for clues that can help discovering the origin, purpose and operational working of the malware. For example, static analysis of the Stuxnet malware identified strings of bits that match the product codes of specific Siemens industrial control products [34]. This serves a clue to its purpose, because we learn that the malware wants to know something about specific Siemens products. There is a whole discipline of reverse engineering malware, for example using the ROSE decompiler [32] to do tasks such as recovering software-development objects to make the static analysis more intelligible [28].

2. *dynamic analysis*: In this style of analysis, a controlled environment (sandbox) is deployed that is both instrumented document of what the malware does and architected to draw out as much of the malware's behaviour as plausible. Code is understood as an *activity*, and the controlled environment is used to measure or test hypotheses made on its functioning. This process is part of an arduous creation of general knowledge, situating the activity in relation to the phenomenon and mechanism of the attack [47]. Again as an example, consider the attack scenario elicited by dynamic analysis of Stuxnet by the Symantec Security Response Team, see [34].

A useful example model of a malware attack on a computer is the kill chain, first proposed by the US defense contractor Lockheed Martin as a summary of the steps taken to attack its systems [23]. The kill chain is useful because it informs the incident respondent what kinds of artefacts or activities to look for if certain other things have already been observed, both in the past to explain how the adversary got in, and into the future to try to prevent further damage.[14] For our purposes, it is enough to understand that the central activity in the model is 'exploitation,' or in our terms, to induce a malfunction in the system, see Figure 6. The rest of the kill chain either describes how the adversary gets to this

[14] See [46] and [47] for detailed discussion of the kill chain and its role in building knowledge in InfoSec.

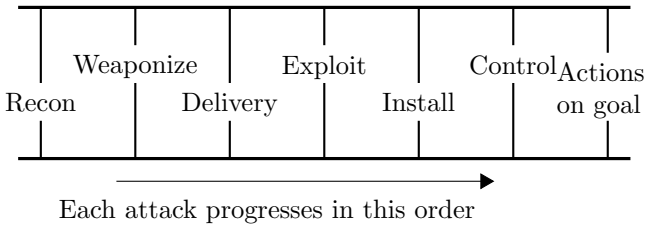

Figure 6: The seven steps of the intrusion kill chain [23].

point, or what the adversary does once successful. Static and dynamic analysis are useful in unique ways, and provide interlocking support for discovering the mechanism by which exploitation happens, and what happens afterwards.

An advantage of a mechanistic model is that it can be applied at several levels, and its granularity depends on how coarse or fine the system to be modelled is. In that respect, the aim of using mechanisms to identify malware seems to be strictly related to the type of taxonomy one is using. Functional-based taxonomies like the one in [50] require the whole attack to be considered as a mechanism; behaviour-based taxonomies like the one in [9] and [25] require not just the modelling of the full attack, but also the inclusion of a comparison activity with legitimate programs. A structural taxonomy like in [41], has allowed us to consider the trust relationships between malicious software and the host system to investigate the type of transformations induced. Such a situation is of interest for attacks whose origin is harder to qualify, or which at the beginning do not offer elements in that respect. Therefore, one might be forced to focus on a fragment of the attack model, and to determine the elements of that fragment as a mechanism in itself. We shall be doing so with reference to entities and activities extracted directly from the taxonomy of Section 3. With the available notion of induced malfunction, one way to apply it is in terms of constraints on a finer-grained part of the kill chain, namely exploitation. The different types of malware will offer different constraints on the rest of the kill chain. For example, Type 0 malware can only induce misfunction, and so the actions on objectives available to the adversary are limited to misfunction of the system.

Let's consider a small example of how mechanistic thinking might be used to understand the transformations malware can induce.[15] At a high level of generality, let's say all the analyst knows of the system entities is:

1. the system in the initial state S;

2. the assumed trustworthy component c_i;

[15] We do not claim this is the only way to analyze, or describe the analysis of, the situation. Some of these steps will be intuitive to professional malware analysts or program verification logicians. We view this similarity as a main contribution. By casting malware analysis in this mechanistic lens, we can see similarities between fields in biology and computer science that otherwise appear starkly dissimilar.

3. the system in the final state S'.

Two critical activities that protect a system are (definitions based on [44]):

1. Authentication: the activity by which the component c_i has its identity verified by the system S;

2. Authorization: the activity by which the system S grants access (to a system resource, potentially a component c_j) to some component c_i.

We could instantiate each of these activities, as [12] calls the heuristic, and describe in detail the mechanism by which the activity of authentication is achieved in, for example, Kerberos version 5 on Ubuntu Linux version 18.04. But for the purposes of this example we view them as activities playing a role in describing malfunction. Malware does not exploit (that is, induce malfunction) in just any system activity, but preferentially targets either the authentication or authorization functionalities of a system. Especially Type I and II malware are in practice the types most interesting from a system compromise prevention (e.g., including program verification) perspective. In Type I and II malware, system components (either static or dynamic, respectively) that should not have been modified are modified. That is, either the authentication or authorization process protecting that component dysfunctioned. At this level of abstraction, exploitation is the phenomenon to be explained. The three types of transformations illustrated in Figure 5 provide distinct but interrelated possible manifestations of that phenomenon – that is, a cluster of mechanisms [12]. This helps guiding the explanation effort by giving options for the analyst to test or attempt to gather evidence for or against.

With these new definitions of malware transformations and our situated mechanistic understanding of them available, it becomes possible to better qualify the type of description presented above in the kill chain, Figure 6. Exploitation can be explained in terms of types of transformations and unwarranted or misused trust, probably by causing dysfunction in certain important system functionalities (in the case of the common Type I and II malware). This level of description is amenable to logical analysis. Logical analysis would be tremendously helpful if it could be applied to scale up malware analysis, either static or dynamic. We hypothesize that, like other general knowledge in InfoSec, such a logic would need to be painstakingly built up out of myriad cases carefully unified across clusters of mechanisms [47]. Thus, it is no small task to build up a precise, automatable account of malware's transformations. Like in other areas, the logic would almost certainly need to be carefully tooled to match the scientific or engineer's model of malware [38].

However, the work presented so far can be immediately applied to improve existing malware documentation, such as MAEC and MMDEF. The purpose of this is to enable the careful knitting together of specific MAEC objects into mechanistic clusters, so similarities can be drawn out and applied to explanation of such malware in future work. In Section 2.5.2, we looked at an example of the MAEC language, with a clear distinction between actions and objects, see Figure

```
<Bundle id="maec-example-bnd-1" schema_version="4.0.1" defined_subject="true">
content_type="dynamic analysis tool output"
  Malware_Instance_Object_Attributes
    Properties type="WindowsExecutableFileObjectType"
      File_Name=dg003_improve_8080_V132.exe
      Size_In_Bytes=196608
      Hashes
        Hash
          Type=MD5 type="HashNameVocab"
          Simple_Hash_Value=4EC0027BEF4D7E1786A04D021FA8A67F
Behaviours
Behaviour id="Transformation_2"
    Associated_Breach="safety"
      Associated_Malfunction="dysfunction"
Actions
  Action id="maec-example-act-1"
    Name=create file type="FileActionNameVocab"
      Associated_Object idref="maec-example-obj-1"
        Association_Type=output type"ActionObjAssocVocab"
Action id="maec-example-act-2"
  Name=write to process memory type="ProcessMemoryActionNameVocab"
  Associated_Objects
    Associated_Object idref="maec-example-obj-1"
      Association_Type=input type="ActionObjAssocVocab"
Objects
  Object id="maec-example-obj-1"
    Properties type="WindowsExecutableFileObjectType"
      File_Name=msvcr.dll
```

Figure 7: An example of MAEC bundle improved with a `Behaviours` category

3: while this description stops at a lower level of abstraction, our analysis offers the possibility to add a further layer expressing the kind of transformation taking place, the associated breach and malfunction (misfunction and/or dysfunction). The malware under consideration is called `dg003_improve_8080_V132`, is a Windows executable object, which creates a file and writes to program memory. This is obviously a breach as an untrusted component is taken as a trusted one, and this matches `Transformation 2`, where we start from a partially secure system and end up in a mis/dysfunctioning system. In Figure 7, we have added a new category to the existing bundle from Figure 3: the category `behaviour` extends the bundle with the information explained previously, leading to a more complete and comprehensive understanding of the malware. We can now clearly distinguish between objects, actions and behaviours, the associated breach is safety, the associated malfunction is of the dysfunction type

according to `Transformation 2`.

5 Conclusions

In this paper we analysed several malware taxonomies and discussed the current languages for malware classification. A taxonomy of malware is generalised based on a trust relation between system and components, and an appropriate functional description of malware as transformers between states of the host system has been introduced. This was done to present an improved classification method for malware, based on a mechanistic explanation. We showed how this can be used to facilitate the existing malware classifications, by extending an example of the MAEC langage. We are convinced that the present treatment of malware classification can provide a more solid basis to extend their formal understanding. Next, we aim at formulating such a formal treatement in terms of a logic of malware.

References

[1] Benedict Addis and Stewart Garrick. Botnet takedowns – our GameOver Zeus experience. In *Botconf*, Nancy, France, Dec 3, 2014. AILB-IBFA.

[2] Chris Alberts, Audrey Dorofee, Georgia Killcrece, Robin Ruefle, and Mark Zajicek. Defining incident management processes for CSIRTS: A work in progress. Technical Report CMU/SEI-2004-TR-015, Software Engineering Institute, Carnegie Mellon University, 2004.

[3] AV-Test. Malware Statistics. Technical report, The Indendent IT-Security Institute, March 2017.

[4] William Bechtel and Robert C. Richardson. *Discovering complexity: Decomposition and localization as strategies in scientific research.* Princeton University Press, Princeton, NJ, 1st edition, 1993.

[5] D. Beck, I. Kirillov, and P. Chase. The MAEC Language – Overview. Technical report, The Mitre Corporation, June 2012.

[6] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[7] Sergio Caltagirone, Andrew Pendergast, and Christopher Betz. The diamond model of intrusion analysis. Technical report, Center for Cyber Intelligence Analysis and Threat Research, 2013. `http://www.threatconnect.com/methodology/diamond_model_of_intrusion_analysis`.

[8] CERT/CC. "basic fuzzing framework (bff)". `https://www.cert.org/vulnerability-analysis/tools/bff.cfm`, 2017. accessed Feb 6, 2017.

[9] F. Cohen. Computer Viruses: Theory and Experiments. *Computers and Security*, 6(1):22–35, February 1987.

[10] Carl F. Craver. Role functions, mechanisms, and hierarchy. *Philosophy of Science*, 68:53–74, 2001.

[11] Carl F. Craver. *Explaining the brain: mechanisms and the mosaic of unity of neuroscience*. Oxford University Press, 2007.

[12] Lindley Darden. *Reasoning in Biological Discoveries: Essays on Mechanisms, Interfield Relations, and Anomaly Resolution*. Cambridge University Press, 2006.

[13] P. Denning. Computer Viruses. Technical report, Research Inst. for Advanced Computer Science, March 1988.

[14] G. Erdélyi. Hide 'n' Seek? Anatomy of Stealth Malware. Technical report, F-Secure Corporation, March 2004.

[15] Luciano Floridi, Nir Fresco, and Giuseppe Primiero. On malfunctioning software. *Synthese*, 192(4):1199–1220, 2015.

[16] N. Fresco and G. Primiero. Miscomputation. *Philosophy & Technology*, 26(3):253–272, 2013.

[17] Didier Galmiche, Daniel Méry, and David Pym. The semantics of BI and resource tableaux. *Mathematical Structures in Computer Science*, 15(06):1033–1088, 2005.

[18] S. Glennan and P. Illari. *Mechanisms and the New Mechanical Philosophy*. Routledge, 2017.

[19] ICSG Malware Metadata Exchange Format Working Group. Malware metadata exchange format behavioral, 2011.

[20] Eric Hatleback and Jonathan M. Spring. A refinement to the general mechanistic account. *Under review*, 2018.

[21] Wybo Houkes and Pieter E. Vermaas. *Technical Functions – On the Use and Design of Artefacts*, volume 1 of *Philosophy of Engineering and Technology*. Springer, 2010.

[22] John D Howard and Thomas A Longstaff. A common language for computer security incidents. Technical Report SAND98-8667, Sandia National Laboratories, Oct 1998.

[23] Eric M Hutchins, Michael J Cloppert, and Rohan M Amin. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, 1:80, 2011.

[24] P. Illari and J. Williamson. What is a mechanims? thinking about mechanisms across the sciences. *European Journal for Philosophy of Science*, 2:119–135, 2012.

[25] Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4(3):251–266, 2008.

[26] B. Jespersen and M. Carrara. Two conceptions of technical malfunction. *Theoria*, 77(2):117–138, 2011.

[27] Bjørn Jespersen and Massimiliano Carrara. A new logic of technical malfunction. *Studia Logica*, 101(3):547–581, 2013.

[28] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. Recovering C++ objects from binaries using inter-procedural data-flow analysis. In *Program Protection and Reverse Engineering Workshop*, San Diego, Jan 25, 2014. ACM.

[29] Simon Kramer and Julian C. Bradfield. A general definition of malware. *Journal in Computer Virology*, 6(2):105–114, 2010.

[30] Peter Kroes. *Proper functions and technical artefact kinds*, pages 89–125. Springer Netherlands, Dordrecht, 2012.

[31] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[32] Lawrence Livermore National Laboratory. Rose compiler infrastructure. `http://rosecompiler.org/`, 2016.

[33] MITRE. Common Weakness Enumeration: A Community-Developed Dictionary of Software Weakness Types v2.9. `http://cwe.mitre.org`, Dec 2015.

[34] E. Chien. N. Falliere, L.O. Murchu. Symantec security response, v.1.4. w32.stuxnet dossier, 2011.

[35] Peter W. O'Hearn. From Categorical Logic to Facebook Engineering. In *Logic in Computer Science (LICS)*, pages 17–20. IEEE, 2015.

[36] G. Piccinini. Computing mechanisms. *Philosophy of Science*, 74(4):501–526, 2007.

[37] Giuseppe Primiero and Mariarosaria Taddeo. A modal type theory for formalizing trusted communications. *J. Applied Logic*, 10(1):92–114, 2012.

[38] David Pym, Jonathan M. Spring, and Peter O'Hearn. Why separation logic works. *Philosophy & Technology*, 2018.

[39] J. Rhee, R. Riley, D. Xu, and X. Jiang. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In *2009 International Conference on Availability, Reliability and Security*, pages 74–81, March 2009.

[40] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (S&P), IEEE Symposium on*, pages 65–79, 2012.

[41] J. Rutkowska. Introducing Stealth Malware Taxonomy. Technical report, COSEINC Advanced Malware Labs, November 2006.

[42] D. Salomon. *Foundations of Computer Security*. Springer, 2006.

[43] Robert Schaefer. The Epistemology of Computer Security. *SIGSOFT Softw. Eng. Notes*, 34(6):8–10, December 2009.

[44] R. Shirey. Internet Security Glossary, Version 2. RFC 4949, August 2007.

[45] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, 1st edition, 2012.

[46] Jonathan M. Spring and Eric Hatleback. Thinking about intrusion kill chains as mechanisms. *Journal of Cybersecurity*, 3(3):185–197, Jan 2017.

[47] Jonathan M Spring and Phyllis Illari. Building general knowledge of mechanisms in information security. *Philosophy & Technology*, 2018.

[48] P. Szor. *The Art and Craft of Computer Virus Research and Defense*. Addison-Wesley, 2005.

[49] D. van Eck. *The Philosophy of Science and Engineering Design*. Springer International Publishing, 2016.

[50] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. In Stuart Staniford and Stefan Savage, editors, *Proceedings of the 2003 ACM Workshop on Rapid Malcode, WORM 2003, Washington, DC, USA, October 27, 2003*, pages 11–18. ACM Press, 2003.