J Comput Virol (2008) 4:251–266
DOI 10.1007/s11416-008-0086-0

ORIGINAL PAPER

Behavioral detection of malware: from a survey towards an established taxonomy

Grégoire Jacob · Hervé Debar · Eric Filiol

Received: 15 June 2007 / Revised: 27 October 2007 / Accepted: 31 January 2008 / Published online: 21 February 2008

Abstract Behavioral detection differs from appearance detection in that it identifies the actions performed by the malware rather than syntactic markers. Identifying these malicious actions and interpreting their final purpose is a complex reasoning process. This paper draws up a survey of the different reasoning techniques deployed among the behavioral detectors. These detectors have been classified according to a new taxonomy introduced inside the paper. Strongly inspired from the domain of program testing, this taxonomy divides the behavioral detectors into two main families: simulation-based and formal detectors. Inside these families, ramifications are then derived according to the data collection mechanisms, the data interpretation, the adopted model and its generation, and the decision support.

1 Introduction

Even though behavioral detection seems a recent trend, in antivirus products as well as in virology research, its principles are not really new. In 1986, Cohen [1,2] already established a basis for behavioral detection within his first formal works. He made his point that viruses, just like any other running program, use the services provided by the system. Predicting the viral nature of a program by its behavior was then equivalent to defining what is, and what is not a legitimate use of the system services. This problem was eventually reduced to an appearance analysis of the inputs sent to the system, which is undecidable. Basically, this definition is strongly linked to the operating system but it can easily be extended to the use of any hardware or software resource (processor, memory, programs). This extended definition is often referred as function-based detection. The difference remains a question of perimeter explaining that function-based and behavioral detections are considered indifferently along the article.

1.1 Two opposite approaches for behavioral detection

As stated by Cohen, two opposite approaches can apprehend the problem of behavioral detection. The first approach is to model the behavior of legitimate programs and measure deviations from this reference. The great advantage of this approach lies in its capacity to detect completely unknown viral strains. Nevertheless, defining a global behavior for programs reveals itself extraordinary complex. An obvious reason is the multitude of applications with different natures existing on a system. A web or mail client exhibits an intensive use of the network facilities whereas a multimedia player decodes large buffers of data and renders them over physical devices such as the graphic or sound cards. No common characteristics can be extracted and a different profile is required for each kind of application. Moreover the available information is too important for each program (several megabytes of code, hundreds of system calls) to be considered wholly. As a consequence, legitimate models are always statistical, thus prone to false positive and non resilient to major environment changes. It explains why, in virology, the second opposite approach of modelling and detecting suspicious behaviors is mainly adopted. When a set proves too

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

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

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

complex to be defined exhaustively, the problem can intuitively be addressed working on its complementary. The main drawback is that unknown malware can no longer be detected as soon as they use innovative viral techniques.

It is interesting to parallel antivirus products with intrusion detection systems, where the perception is diametrically opposed. In the intrusion domain, behavioral detection is based on legitimate models whereas the suspicious models used in virology are considered as simple signatures for knowledge-based detection, also called misuse detection [3,4]. Modelling legitimate behaviors goes back to the early works on intrusion detection published by Anderson [5] and Denning [6]. It still remains an active research field as it is clearly impossible to generate misuse signatures for the thousands of vulnerabilities discovered every year. Such models are out of the scope of this paper but, for further information, the reader is invited to refer to the works of Forrest et al. [7] on host-based intrusion detection and the works of Zanero [8] on the use of Markovian Models to capture legitimate uses of systems. To go back to our main focus, viral techniques are less numerous than vulnerabilities and misuse models seem more adequate to the present problem of malware detection.

1.2 Paper contribution and organization

In virology, the domain of behavioral detection shows an increasing activity both in commercial products and research. Paradoxically, no global survey covering this domain has been published. A striking multitude of behavioral detection systems can be observed without any will of consistency in the used vocabulary and designations. To our knowledge, this taxonomy dedicated to behavioral detection in virology is the first of its kind, contrary to intrusion detection where the literature is abounding. The novelty of our approach lies in the parallel made with the domain of program testing which makes a distinction between simulation-based and formal verifications. The domain of behavioral detection should benefit greatly from a consistent reference taxonomy. In effect, this taxonomy should remove the divisions between the different sub-domains of behavioral detection, helping information sharing and reuse.

The scope of this taxonomy has been defined as wide as possible, according to the definition of behavioral detection given in introduction. The virology point of view has been willingly chosen, meaning that the modelling of suspicious behaviors has been implicitly considered. As the need arises, relevant intrusion detection papers are also given as additional references. Globally, the paper has been organized as follows: Sect. 2 explains the recent interest in behavioral detection by the predicted failure of appearance detection, Sect. 3 describes a generic behavioral detection system, Sect. 4 is the core part of the article introducing the taxonomy, and Sect. 5 gives an illustrative overview of both existing commercial products and research prototypes.

2 Why behavioral detection may succeed where form-based detection will undeniably fail

Historically, appearance detection also called form-based detection has been the first technique used to fight malware and still remains at the heart of nowadays antivirus software. These detection techniques search system objects such as files for suspicious byte patterns referenced in a base of signatures. These betraying patterns must exhibit a discriminating character combined with non-incriminating properties for legitimate programs [9, p. 147]. Even if these purely syntactic signatures can precisely identify the threat and name it, form-based techniques are bound to detect known malware or trivial variants.

On the opposite, behavioral signatures are no longer simple byte patterns but complex meta-structures carrying dynamic aspects and a semantic interpretation. Programs with distinct syntaxes can basically have an identical behavior captured by a single behavioral signature. As a consequence, a behavioral signature no longer identifies a single piece of malware but a whole class of malware. Behavioral detection is thus more generic and more resilient to modifications than form-based detection. On the other hand, a precise identification of a piece of malware inside its class is no longer possible, which can be problematic when choosing the relevant countermeasure. Nevertheless, behavioral detection should bring a solution to two of the major problems encountered by form-based detection.

2.1 The signature extraction problem

Form-based detection provides undeniable advantages for operational use. It uses optimized pattern matching algorithms with controlled complexity and very low false positive rates. Unfortunately, form-based detection proves completely overwhelmed by the quick evolution of the viral attacks. The bottleneck in the detection process lies in the signature generation and the distribution process following the discovery of new malware.

The signature generation is often a manual process requiring a tight code analysis that is extremely time consuming. Once generated, it must be distributed to the potential targets. In the best cases, this distribution is automatic but if this update is manually triggered by the user, it can still take days. In a context where worms such as Sapphire are able to infect more than 90% of the vulnerable machines in less than 10 min, attacks and protection do not act on the same time scale.

Moreover this signature can easily be bypassed by creating a new version of a known viral strain. The required modifications are not considerable; they simply need to be performed at the signature level. The numerous versions of the Bagle e-mail worm referenced by certain observatories illustrate the phenomenon [10]. In a few months, several versions have been released by simply modifying the mail subject or adding a backdoor. With regards to more recent developments, a major concern during the last RSA Security Conference was the server-side polymorphic malware Storm Worm [11]. Its writer produces beforehand vast quantities of variants which are delivered daily in massive bursts. Each burst contains several different short-lived variants leaving no time to develop signatures for all of them. On a long-term scale, experts will not be able to cope with this proliferation. As an obvious explanation, formal works led by Filiol [12] underline the ease of signature extraction by a simple black box analysis. This extraction remains possible because of weak signature schemes.

Because of its generic features, a single behavior signature should detect all malware versions coming from a common strain. Experts would be able then to establish a hierarchy in their work, focusing uppermost on new innovative strains. In addition, another side effect of form-based detection is the alarmingly growing size of the signature bases. As a solution, older signatures are regularly removed leaving the system once again vulnerable. On the opposite, the behavior base size is less consequent and the signature distribution less frequent. Regular base updates remain nevertheless necessary, contrary to what certain marketing speeches claim.

2.2 Resilience to automatic mutations

In the previous part, we have considered the manual evolution of malware. What happens when these mutations become automatic during propagation? The first significant generation of mutation engines is born with polymorphism [13, p. 140], [14, p. 252]. Polymorphic malware encrypt their entire code in order to conceal any potential signature. A simple variation of the ciphering key modifies totally their byte sequences. A decryption routine is then required to recover the original code and execute it. This routine must possess its own mutation facilities to avoid becoming a signature on its own.

It was quickly discovered that simple emulation could thwart these engines, making the original code available. But searching for signatures has become far more complex with metamorphism. The malware is not simply encrypted; its whole body suffers transformations affecting its form while keeping its global functioning [13, p. 148], [14, p. 269]. The mutation process always begins with disassembling the code, which is then obfuscated before being reassembled: code reordering, garbage insertion, register reassignment and equivalent instruction substitution. Syntactic analysis is no longer sufficient to fight these mutations. Eventually, Spinellis [15] has shown that the detection of mutating size-bounded viruses by signature is NP-complete. For metamorphic viruses, whose size in unbounded, the result is even worse. Filiol [16] showed that some well chosen rewriting rules could lead to the undecidability of detection.

If these mutations modify the malware syntax, they are not likely to modify its semantic, at least for the known cases. Typically, the malware will always use the system services and resources in an identical way. Behavioral approaches should consequently offer a better resilience to mutations.

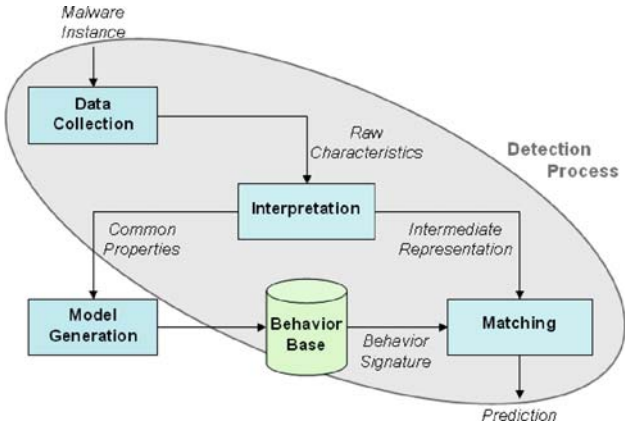


Fig. 1 Functional design of a behavioral detector. This decomposition of the system brings into light the articulation between the generation of the behavior models and the detection process. Each one of the three sequential tasks making up detection, processes the data to a higher level of interpretation until the final assessment

3 Generic description of a behavioral detector

3.1 System architecture and functioning

A behavioral detector identifies the different actions of a program through its use of the system resources. Based on its knowledge of malware, the detector must be able to decide whether these actions betray a malicious activity or not. Information about system use is mainly available in the host environment thus explaining that behavioral detectors work at this level. How malware are introduced in the host is not the main focus of antivirus products. They can either be automatically introduced through a vulnerability, which is the concern of intrusion detection, or manually introduced by negligence of the user. Antivirus products often act as a last local barrier of protection when previous barriers (firewalls, intrusion detection systems. . .) have been successfully bypassed.

Behavioral detectors are basically split into four main components responsible for distinct tasks. This decomposition is schematically represented in Fig. 1.

At first, we will purely focus on detection. The detection process consists in three sequential tasks addressed by individual components. A first component performs the data collection, where dynamic capture and static extraction have been considered indifferently. In both modes, the intended actions of a program can be observed: in the first case only the effectively performed actions are collected whereas in the second all potential actions are. In practise, data can be collected from different sources: the local host for personal computers or host honeypots deployed in strategic points for networks. Because behavioral detectors work at a higher interpretation level than simple form-based detection, a second component is required to analyze and interpret the collected data. This second tasks brings into light the important characteristics of the collected data. These characteristics are then formatted into an intermediate representation to feed the last part of the process. The last component embeds the matching algorithm used to compare the representation to the behavior signatures. According to the result, the program will be labelled as malicious or benign. This division into three pieces is particularly important since it structures the coming taxonomy.

Extending our perspective, a preparatory step is obviously required prior to detection. An initial task is required to generate the behavior signatures stored in the dedicated database. Just as for detection, the signature generation relies on common properties that are extracted by interpretation on a pool of known malware. Even if this generation may not be an actual software component, it requires a dedicated process since it is a key element in the detection efficiency. As a consequence, the signature generation shall also be an element of our taxonomy.

3.2 Basic properties for assessment

It is fundamental to define the important properties of a behavioral detection system since they provide the main basis for assessment. Actual certifications simply confront malware detectors to known strains thereby solely assessing completeness of form-based detection. Assessing antivirus product is still an open problem and several more complete test procedures have been put forward [17,18]. One of them focuses more particularly on behavioral detection using functional mutations [19]. This new kind of mutation generates new viral strains using different known techniques to achieve a same final behavior that should be detected. This test procedure has among others shown that, in order to make up for the false positive rates, the behavioral detection is often confronted to an additional syntactic signature. If a neat decrease is observed in these rates, the behavioral detection remains severely hindered by this measure which prevents the detection of unknown strains using known viral techniques. More generally, any complete test procedure with regards to behavioral detection should at least consider the following properties:

- **Completeness and accuracy.** A system which fails to detect too many malware is said incomplete because its false negative rate is too high. These failures may be explained either by incomplete behavior signatures or missing data that remain uncollected. On the other hand, accuracy determines the system tendency to false positives. Two factors mainly impact this property: the soundness of the chosen signatures and the relevance of the collected data. Regarding coverage, two others important properties are directly related:
  - **Adaptability.** When a system is deemed inaccurate or incomplete, modifications must often be performed on the behavior signatures. Adaptability traduces the ease of modification of the chosen behavior model.
  - **Resilience.** Malware often deploy anti-analysis mechanisms. These techniques introduce bias during the data collection in order to blur any similitude with the behavior models. Obfuscation and, respectively, stealth are effective means used to thwart, respectively, static and dynamic detections. The coverage of the behavior model should resist to such attempts.
- **Efficiency.** Efficiency must not be only restricted to performance which measures the resource consumption introduced by the detector in its environment. This consumption is undoubtedly an important property since the overload it introduces, explains the belated interest in behavioral detection. For several years, the computing power of processors, the available memory space and bandwidth have been insufficient in order to deploy such complex techniques. Performance may vary according to various factors such as the data collection mechanism, the deployment of the detector on personal computers or dedicated honeypots. But another aspect is even more critical to efficiency. The computational complexity of the detection algorithm constitutes the ultimate boundary to the detection efficiency. In case of approximate methods, the precision bounds this complexity. In addition, dynamic properties are introduced by the fact that the malware may be active during the detection process.
  - **Timeliness.** Timeliness checks whereas the detection is reached before the damages, done to the environment by the observed malware, are irreversible.
  - **Fault-tolerance and unobtrusiveness.** Fault-tolerance assesses the capability of the behavioral detector to stand up to any external perturbation and in particular intended attacks launched by malware. On the opposite, according to the principle of the physicist Schrödinger, the behavioral detection must not introduce perturbations in the malware execution.

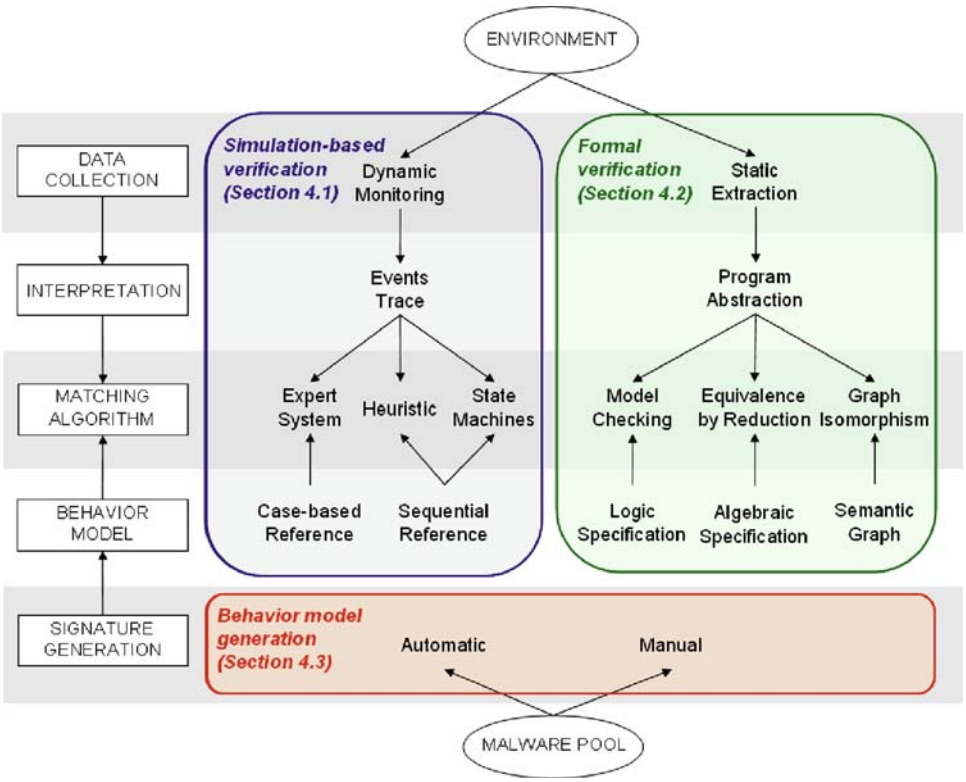


Fig. 2 Characteristics of behavioral detectors. The classification is globally divided into two axes corresponding to the simulation-based verification and formal verification. The type of verification is directly impacted by the method used for data collection: static or dynamic. The behavioral model generation is introduced as an additional transversal axis

Unobtrusiveness guarantees that the observed behavior will not be altered by the detection.

4 Taxonomy of behavioral detector

As said in Sect. 1, the leading thread of our taxonomy is the parallel made between behavioral detection and program testing. The taxonomy is thus built on two main axes dividing the detection process into simulation-based verification and formal verification. A third transversal axis is added for the behavioral model generation. The global structure of the taxonomy and the classification of the detectors is pictured in Fig. 2.

Inside simulation-based verification and formal verification, the different classes of detectors have been divided according to the different tasks of the detection process (see Sect. 3.1). In particular, these tasks define the progression used in the next sections that detail, respectively, the two verification approaches. To simplify the reading, data collection and interpretation are presented in a same part because the nature and the quantity of the collected data strongly impact the possible means of interpretation. Similarly, the matching algorithms and the behavior models are also gathered in a same part since the algorithm directly determine the model format. Now let us describe the three axes of the taxonomy according to this division.

4.1 Simulation-based verification

Simulation-based verification is similar to a black box test procedure and is thus strongly linked to dynamic analysis. Only the current execution path is analyzed making the behavioral detector work on a sequence of discrete events that will be compared to the reference model: the behavioral signature. In particular, this kind of verification requires a dedicated simulation environment for data collection.

4.1.1 Data collection and interpretation: dynamic monitoring

Detection of malware during their execution must rely on elements observable from an external agent. On older operating systems, the interception of interruptions was the first source of information about the resources used by a program. They have been progressively replaced by the interception of system calls with the apparition of 32-bit systems. System calls are particularly interesting since, in order to comply with the C2 criteria from the Orange Book, they remain a mandatory passing point to access kernel services and objects from the user space. In their work on intrusion detection based on system calls, Forrest et al. [7] underline the importance of the collected data and their representation: they strongly influence the analysis and the detection. In the present case, sequential representations are mainly used

```
Process Id:  2884 "Word.exe"
    Privilege Lv:  user
Time:  16/01/2007 1:53:34:536
        #ZwReadFile#
hFile = C:\document.doc:0x24E6B0
    lpBuffer = 0x13E67C
  nNumberOfBytesToRead = 10
    nByteOffset = 0
```

Fig. 3 Extract from a trace of system calls. The whole trace is made up of a list of system calls with various attached information. The process identifier is important to correlate the different system calls from a target process

but other representations like frequency spectres could be considered. In addition, the context of the system calls must also be attached. The passed parameters, the identifier of the calling program as well as its privilege level are useful information to refine the interpretation. By nature any system call is legitimate, only the arguments betray a malicious purpose as stated by Kruegel et al. [20]. As an illustration, a simple extract from a system call trace is given in Fig. 3.

The nature of the collected data is not the only factor to consider for classification. The monitoring conditions are equally important. According to these conditions, several properties of the detector may be impacted: performance, unobtrusiveness, timeliness or the completeness of the available data.

Real-time conditions: The progression of the malware is observed directly in its environment without restrictions. Real-time conditions are often criticized because malevolent actions are effectively executed. Timeliness is thus of utmost importance before the point of no return of the infection is reached. To intercept system calls in real-time, the main technique used by detectors is API hooking which is often used by rootkit writers as well [21]. The overload generated by the interception and the interpretation of the call may be perceptible by the user. Yet, it remains less significant than for the other capture conditions.

Real-time with action recording: Action recording is a particular case of real-time capture where the actions taken by the observed program are recorded as well as the intermediate states of the environment [22,23]. This trade-off benefits from the advantages of real-time monitoring while keeping a possibility to restore the environment in a healthy state as soon as a threat is detected. This countermeasure remains possible as long as the restoration mechanism and the records stay uncompromised.

Sandboxes: The observed target is first run in a sandbox where its execution is isolated in a confined space [24,25]. This technique, popularized by JAVA, constrains the execution in an escape-proof memory space with low privileges and limited accesses to services. The main advantage is that the external observer has a total access over the memory space and can control the execution step by step. Sandboxes offer better observation facilities than real-time conditions. On the other hand, they use more significant resources since they introduce an intermediate layer between the program and its environment. To reduce the overload, only suspicious code portions of the program are analysed. Once this preanalysis is performed, the normal execution of legitimate programs is resumed without hindrance. Unfortunately, sandboxes provide a set of services more restricted than real systems and can easily be detected. Debugging detection techniques checking the execution time or using error handling structures can succeed easily. Once the sandbox is detected, the malware can adapt its execution to look benign. If the privileges and service accesses are not properly restrained, malware can even escape through open interfaces of the sandbox.

Virtual machines: Virtual machines can emulate a whole environment with minimal risks to be detected. In effect, the host environment controls every access point to the hardware from the unaware guest system. In the case of purely software virtual machines, system calls can be intercepted at the level of the emulated processor by recognizing the INT 2E and SYSENTER instructions. The analysis can then be performed before entering or after returning from the system call without leaving any trace for the virtual environment that can carry on its execution [26]. In comparison to sandboxes, virtual machines emulate any fictive resource, either hardware (network connections) or software (mail or P2P clients). These resources are often used malevolently by the malware to its own profit either to propagate or gather information. Total virtualization enables the observation of these interactions without risks for the host. On the other hand, virtual machines require large amount of resources making them impossible to use in an operational context except with restricted virtualization support (only the file system for example). They remain mainly used by experts and researchers on the purpose of analysis and classification. Just as sandboxes, they can be detected by the observed program but no escaping technique has been reported yet [27,28].

Whatever dynamic condition is considered, they all globally exhibit the same properties. As a comparison basis, we have identified the following ones:

Assets: Dynamic monitoring proves resilient to most mutations techniques like polymorphism and metamorphism. These mutations are fundamentally based on syntax and

```
Deny   Write       Run Registry Key
Deny   Write       Win.ini File
Deny   Terminate   Antivirus Process
```

Fig. 4 Rules for expert systems. A rule always specifies the nature of the action (reading, writing, opening, terminating...), the target along with the associated decision (permission, refusal). If no rule is defined, the action is allowed by default

thus do not modify the final execution. The different versions issued of a same mutating strain eventually provide the same event trace.

Limitations: The interception of system calls is not the ultimate solution. Certain behaviors such as encryption do not use the system services for obvious stealth reasons. Some malware even redefine whole system primitives for these exact same reasons. Another phenomenon to take into account is the migration of malware towards the system kernel, in order to acquire privileges equal to antivirus products. Using these privileges, complex stealth techniques become possible since malware can interact directly with the hardware and the system objects without necessarily using any of the monitored system calls [13, p. 188]. This limitation could be solved by capturing additional data from more privileged sources. On the other hand, the second limitation can not easily be solved. By nature, dynamic monitoring only captures the current execution path. This execution path could be biased since non deterministic behaviors may be randomly executed or conditioned by external stimuli and observations (sandbox and virtual machine detection for example).

4.1.2 Matching algorithms and models: expert systems

Expert systems rely on a set of case-based rules modelling the experience and expertise of an analyst confronted to a particular situation [29]. Like the ones pictured in Fig. 4, rules are defined for each known suspicious attempt to use system facilities. Every separated action taken by the observed program is dynamically confronted to the related rules. The target and the privilege level of the caller are important factors because they often draw the distinction between a legitimate action and a malicious one. The class of complexity for the rule-matching algorithms remains acceptable since it is equivalent to pattern matching algorithms that are in the class P.

The decision whether a behavior is malicious or not must then be preemptively taken. Attempts to use a service can be intercepted by systems such as the one in Fig. 5. They can react consequently before these attempts are resolved, explaining why these proactive systems are often called "behavioral blockers" [30]. Generally speaking, expert systems are prone to false positives because it proves really

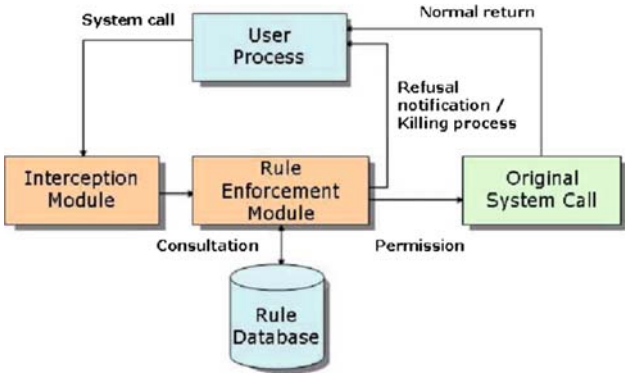


Fig. 5 Rules enforcement. For each captured system call, the related rules will be scanned. According to the relevant rule, the system yields the control to the originally called function or sends a refusal/killing notification to the calling process

intricate to judge the legitimacy of separated actions without correlation.

4.1.3 Matching algorithms and models: heuristic engines

Historically, heuristic engines were the first detectors deployed to detect malicious functionalities. Contrary to the previous expert systems, the captured actions are no longer considered separately but sequentially. They function on the basis of interruptions and system calls, usually collected thanks to a sandbox, along with their preceding instructions defining their parameters. Basically, heuristic engines are made up of three parts [31,32]:

Association mechanism: Association mechanisms label the different atomic behaviors of malware. An atomic behavior corresponds to a functional interpretation of one or several instructions as pictured in Fig. 6. Fundamentally, two labelling techniques exist. Weight-based association uses quantitative values, obtained by experimentation, in order to express the action severity. Flag-based association uses semantic symbols to express a corresponding functionality [33,34]. Figure 7 presents a typical example of flag-based association where atomic actions eventually corresponds to real instructions sequences.

Rule database: This database defines the detection criterion. In the case of weight-based systems, there is a unique detection rule consisting in a threshold above which the accumulation of malicious behaviors betrays a malicious activity. Otherwise, the detection rules consist in flag sequences. These sequences are combined together into a detection tree like pictured in Fig. 8.

Detection strategy: The detection strategy impacts the progression within the detection rules. In the case of a

Fig. 6 Atomic behaviors. This example is quoted from the documentation of the Bloodhound engine [35]. It illustrates the association between several instruction sequences and a final atomic action

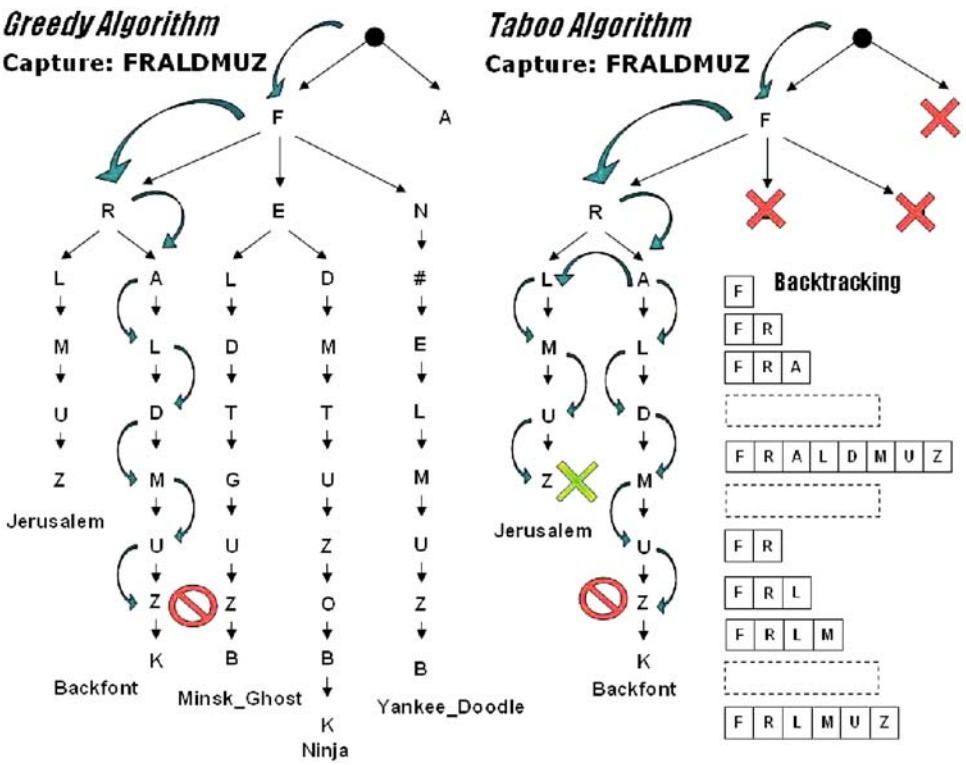
```
Terminate program
1.   MOV AX, ??4Ch
     INT 21       ;B8??4CCD21
2.   MOV AH, 4Ch
     INT 21       ;B44CCD21
3.   MOV AH, 4Ch
     MOV AL, ??h
     INT 21       ;B44CB0??CD21
4.   MOV AL, ??h
     MOV AH, 4Ch
     INT 21       ;B0??B44CCD21
```

```
Open File
100.  MOV AX, 023Dh
      MOV DX, ????h
      INT 21       ;B8023DBA????CD21
101.  MOV DX, ????h
      MOV AX, 023Dh
      INT 21       ;BA????B8023DCD21
```

Fig. 7 Behavior base. This example has been extracted from the base of the TBScan engine [33]. Each behavior is associated to a flag carrying a semantic value

```
F = Suspicious file access     R = Suspicious code relocation
N = Wrong name extension       A = Suspicious memory allocation
# = Deciphering routine        L = Trapping the loading of software
E = Flexible entry-point       D = Direct write access to the hard drive
M = Memory resident code       T = Invalid timestamp
G = Garbage instructions       Z = Search routine for EXE/COM files
B = Back to entry-point        K = Unusual stack structure
O = Overwriting or moving programs in memory
```

Fig. 8 Detection rules and strategies. The trees are built according to rules from the TBScan engine, corresponding to five viral strains [33]. The first chosen strategy is a simple greedy algorithm where the first valid path is always taken without possibilities to go back. This combination fails to detect Backfont but another strategy where back-steps are possible can detect the virus. A backtracking mechanism is integrated to taboo algorithms in order to store the explored nodes, thus allowing back-steps for authorized branches (other branches are called taboo). Irrelevant behaviors can then be ignored to detect Jerusalem. This observation underlines the importance of the strategy



weight-based association, the strategy is the accumulation function, chosen to correlate the captured values. Otherwise, the strategy determines the tree search algorithm. Several kinds of algorithm exist: greedy without possible back-steps during exploration as in Fig. 8, genetic, taboo or simulated annealing with

conditioned back-steps [36]. The choice of the strategy is primordial since it allows to find approaching but still satisfactory values in a reasonable delay for NP-complete problems [13, p. 67].

4.1.4 Matching algorithms and models: state machines

Just like heuristic engines, state machines are based on sequential models of system calls. The malicious behaviors are described as Deterministic Finite Automata (DFA) according to the following principle [37,38]:

- The states S of an automaton corresponds to the internal states of the malware along their lifecycle.
- The set of input symbols Σ defined upon the collected data which are mainly system calls.
- The transition function T describes the symbol sequences known as suspicious.
- The initial state s_0 corresponds to the beginning of the analysis.
- The set of accepting states A conveying the detection of a suspicious behavior.

From an initial state, the automaton will progress step-by-step by evaluating the elements from the sequence of collected data. If during its progression, the automaton reaches an accepting state, a malicious behavior has been discovered. Otherwise, if the automaton does not reach an accepting step before the end of the data sequence or reaches an error state, only behaviors supposed legitimate have been captured. Figure 9 gives an example of automaton detecting a file infection mechanism. In state machines, the matching algorithm is defined by the word acceptance problem by an automaton. Using deterministic finite automata, the complexity of this problem remains in P [39].

Notice that state machines can also be used for the opposite approach of behavioral detection that is to say modelling legitimate behaviors. The considered automaton is then no longer deterministic but probabilistic. The probabilities of the different transitions may be based on the frequency of certain system call sequences during a healthy execution [40]. Unfortunately, the model put forward is used to detect macroviruses and consequently targets a specific type of application: office software. It remains almost impossible to extend generically legitimate models to every application.

4.2 Formal verification

Behavioral detection is traditionally associated to dynamic execution and thus to simulation-based verification. This seems a short-sighted view since behaviors are originally written down in the code of malware. Thereby, the malware actions can also be discovered formally through static

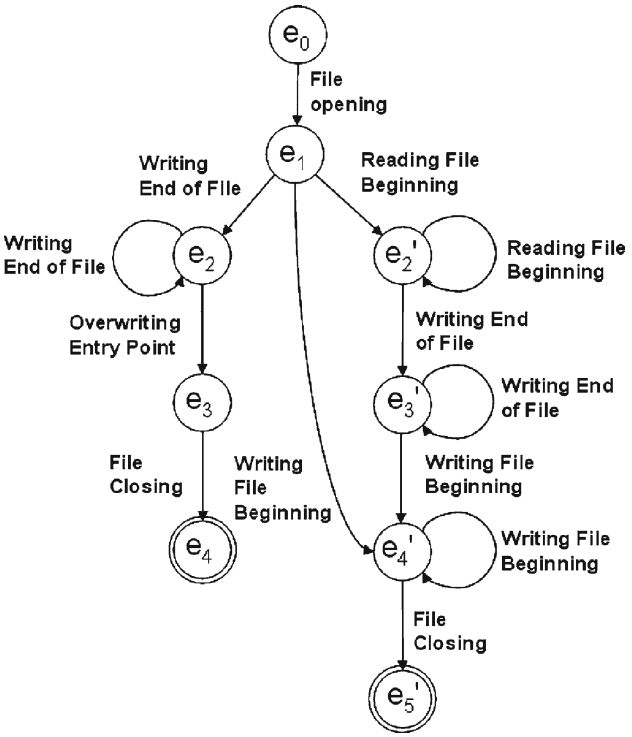


Fig. 9 Automaton of the infection mechanism. This automaton describes two types of file infection. The *left branch* depicts the "append" infections where the viral code is copied at the end of the file and the entry-point is redirected. The *right one* depicts the "prepend" infections, destructive or not. Either the original code is saved at the intermediate states e'_2 and e'_2 or the automaton jumps directly to the infection point at state e'_4

analysis. Formal verification, in the context of behavioral detection, consists in verifying that a program abstraction satisfies or not a behavior formal specification, which is basically a bisimulation problem. Thanks to this white box approach, these detectors can combinatorially explore the different execution paths. Only few systems have been referenced for formal verification since it remains a recent approach.

4.2.1 Data collection and interpretation: static extraction

Static extraction provides richer and more complete information about potential actions than dynamic monitoring which is bound to collect observable elements only. The original code sample may simply be a local file from the system but also a file rebuilt from different payloads collected by a honeypot. The main challenge is to reach, from the binary code, a semantic level of interpretation traducing the intended actions. Consequently, the data extraction is quite complex and requires several processing steps to get an intermediate representation of the program.

Static extraction, described with more details in Fig. 10, uses the traditional techniques of reverse engineering, that is

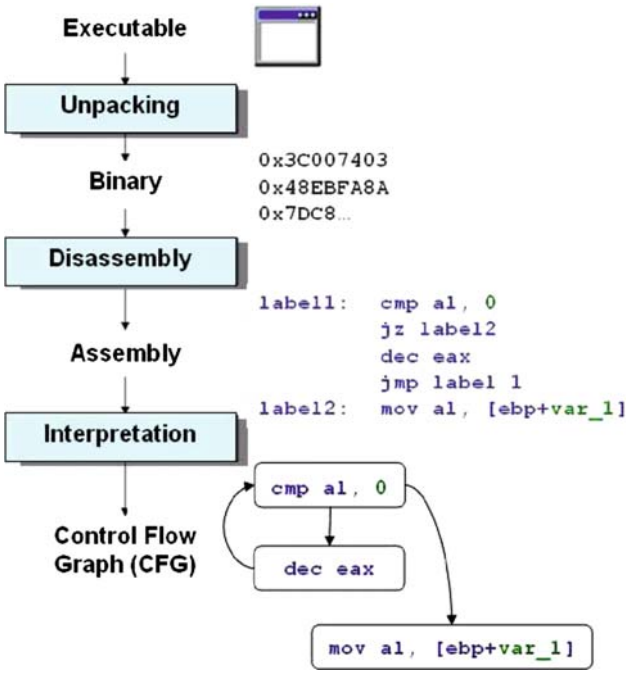


Fig. 10 Incremental steps of the static extraction. This scheme describes the different processing stages applied to the program in order to extract the intermediate representation: unpacking when required, disassembly and interpretation

to say, disassembly and building of Control Flow Graphs and Data Flow Graphs (CFG and DFG). Graph-based representations are used as a majority since they bring into light the different execution paths of the program. In certain cases, the instructions and values stored in the nodes of the graphs can even be interpreted according to a more generic semantic.

In the simplest cases, existing tools can automatically achieve the process, but additional human interventions are often required [41]. This is explained by the existence of software protection techniques which can skew the result of the extraction. For example, automatic disassembly can be thwarted by the simple introduction of fake instructions that hinder the code alignment. Generally speaking, static extraction is very sensitive to the obfuscation techniques used by metamorphic engines [42]. Complex dedicated techniques are required to bypass these software protections and, unfortunately, they can hardly be automated [43]. In fact malware are often protected using automatic packers like UPX which deploy these kinds of protections. Unpacking has become a challenging problem in static analysis, requiring more and more advanced techniques [44].

Just like dynamic monitoring, the intrinsic properties of static extraction provides advantages but also drawbacks. By comparing its properties with those of dynamic capture, it becomes obvious that these two capture methods are complementary:

Assets: The main advantage of static extraction lies in the fact that all execution paths are enumeratively available. Since malware are not running during the capture, they are not able to adapt their execution or deploy proactive defence during the analysis.

Limitations: Predicting the behavior of a program from its simple description is equivalent to the "halting problem". Unfortunately this problem has been proven undecidable by A. Turing in 1936. Still, under certain conditions, the necessary information can be gathered. Anyhow, static extraction remains possible as long as disassembly can be performed, which is a quite strong hypothesis because of the protection techniques mentioned previously. Theoretical works to assess the resistance of static semantic analyzers to obfuscation transformations have already been addressed by Preda et al. [45].

4.2.2 Matching algorithms and models: annoted graph isomorphism

Isomorphism of annoted graphs works exclusively with static extraction since it uses Control Flow Graphs (CFG). The instructions stored in the nodes of the extracted graphs are often replaced by an associated label to reach a higher level of abstraction than simple assembly code. The labelling procedure may follow two approaches: either the instructions are translated into an intermediate representation carrying a semantic value [45,46] or instructions are reduced to their basic class of operation (arithmetic, logic, function call...) [47,48]. A behavior template, or behavioral signature, is thus specified by a graph structure using an annotation mechanism. Figure 11a provides an outlook of a behavior template with its graph and its associated semantic labels made up of symbolic instructions, variables and constants.

Detection is achieved by checking that a program satisfies a given template, which is equivalent to finding a subgraph of its extracted CFG, isomorphic with the behavior graph. The localisation of this subgraph in stand-alone malware may be easy to determine but it proves much more complex for program infectors since it requires finding out the insertion point first. The isomorphism algorithm then begins with associating the nodes from the extracted CFG with those of the template as pictured in Fig. 11. An additional constraint steps in since a sensible correspondence must be possible between the labels from the graph nodes. Using a representation with semantic labels, this association eventually determines the equivalences between the symbolic elements (variables, constants) and the real values (registers, memory locations). An additional step may be deployed to check the preservation of these values from their affectation until their use.

Theoretically, the subgraph isomorphism on its own is NP-complete but its complexity can often be reduced in the detection context. In effect, CFG nodes, except in the case of indirect jumps and function returns, have a bounded number of successors, typically one or two. Isomorphism remains

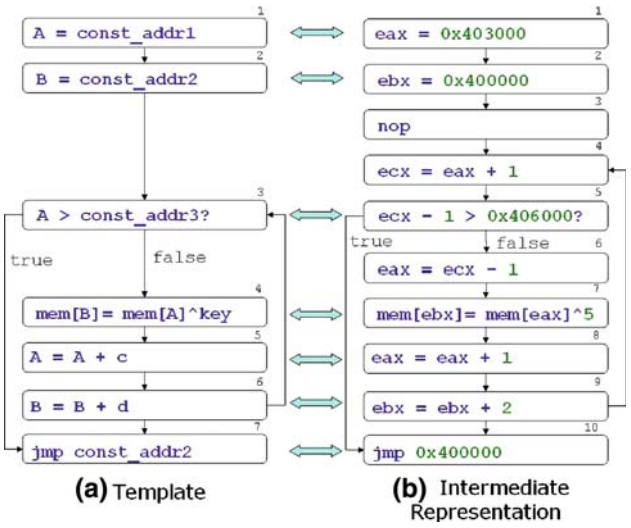


Fig. 11 Graph isomorphism with semantic equivalence. The template (**a**), quoted from the paper of Christodorescu et al. [46], generically represents a simple ciphering by XOR between two addresses. During the verification, each node from the instance (**b**) is associated to its potentially equivalent node in the template. The instance (**b**) satisfies the template (**a**). Once the correspondence is established, the variable preservation can be checked. In this concrete case, the value affected to eax at node 1 must be equal to the value of ecx used at node 5

very sensitive to mutation techniques and in particular to any modification impacting the graph resulting from the extraction: code permutation or injection (dead code hidden behind opaque predicate, additional intermediate variables). These transformations can partially be addressed by optimization techniques developed for compilers [47,49,50]. The ultimate goal would be to reach a canonical and minimal form for malware, to revert most mutation effects.

4.2.3 Matching algorithms and models: equivalence by reduction

In equivalence by reduction, the detection relies on an algebraic approach. The algorithm progresses by deduction using logical equivalence at each reasoning step [51,52].

The original program is first translated into an algebra: commonly a formal specification of the processor instruction set which attempts to erase differences between equivalent functionalities. A single algebraic expression will stand for several equivalent instructions such as 'mov' operations using different registers for example.

Once translated, the program abstraction is then simplified by reduction using rewriting rules preserving some equivalence and semi-equivalence properties. Basically, equivalent expressions have an identical effect on the whole memory whereas semi-equivalent ones only preserve specific variables and locations. The final purpose is to reduce the number of syntactic variants like pictured by the rewriting rules of Fig. 12, reversing some metamorphic transformations.

The reduced form is finally checked using an interpreter to evaluate the results of its execution on different variables or memory locations such as the stack. These results are compared to the results for known malware specifications, given in the same precise algebra. Because of the problem complexity, which is equivalent to the halting problem and thus undecidable, this technique can only be deployed on limited code samples from malware.

4.2.4 Matching algorithms and models: model checkers

In model checking, the model used to describe the behaviors is more peculiar. A behavioral signature is defined by a temporal logic formula [53,54] which introduces dynamic aspects in the first-order logic. A detailed example is given in Fig. 13. For more information, it is recommended to refer to the corresponding literature [55]. The verification algorithm takes as input a control flow graph as well as one or several logic formulae. In return, it sends back all the intermediate states in the different execution paths satisfying these formulae. This kind of algorithm is strongly recursive since it tries to explore enumeratively all the possible execution paths which can unfortunately be infinite. As a matter of fact, symbolic temporal model checkers exist which prove to be PSpace-complete [56].

In the most recent logics, registers, free variables and constants are referenced as generic values for a better abstraction [57]. This improvement particularly addresses the mutations by reassignment as shown by Fig. 13. During the verification process, the algorithm links the generic values with real registers and variables and stores this information all along the explored execution path. Notice that a higher level of semantic abstraction could also be possible just like for the two previous kinds of algorithm. In addition, the temporal predicates used to explore the different paths prove to be really useful thwarting garbage code insertion and code reordering.

```
eq execS NOP in EVL /\/\ FL = EVL /\/\ FL.
eq execS do SL1 while (T) in EVL /\/\ FL = execSL SL1 ;; while(T)
do SL1 ; eof in EVL /\/\ FL.
```

Fig. 12 Reduction rules reversing metamorphic transformations. These two rewriting rules quoted from Webster paper [51] are written using the OBJ formalism. Given a virus in SPL, the first rule is used to remove the NOP that may have been inserted during possible mutations. The second one may seem more complex but simply says that a $do\{\_\}\ while(\_)$ is equivalent to a $while(\_)\ do\{\_\}$

$$C_1 : \exists r EF(mov(r,0) \wedge EF(push(r) \wedge EF(call(GetModuleHandle))))$$

$$C_2 : \exists r EF(mov(r,0) \wedge AX(push(r) \wedge AX(call(GetModuleHandle))))$$

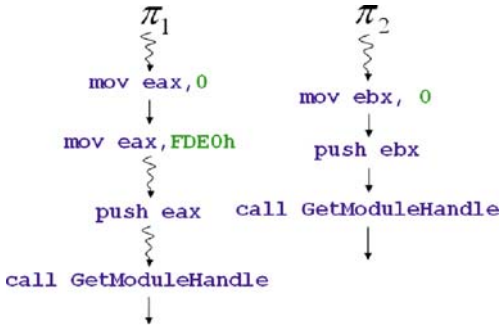


Fig. 13 Temporal logic formulae to detect auto-reference accesses. A and E are path quantifiers whereas X and F are temporal operators. The combination $EF(p)$ means that an execution path exists where an undetermined future state satisfies the predicate p. In the present case, C_1 means that there is a possible path where 0 is affected to a register r that will be pushed on the stack before a call to the function $GetModuleHandle$. These operations may not be consecutive. Replacing the operators EF by AX in C_2 compels the register affectation and the following call to be immediate in every possible path (and no longer in at least one). π_1 and π_2 are two illustrative execution paths satisfying, respectively, C_1 and C_2 but only C_2 captures auto-reference accesses which are basically calls to $GetModuleHandle$ with a null value

4.3 Behavior model generation

Along the two previous sections, we have described several behavior models without mentioning the creation process of the behavioral signatures. This third part is dedicated to the third transversal axis of signature generation.

4.3.1 Manual definition

Manual definition, though time consuming, remains the principal generation method, because of its reliability. Two main sources of knowledge are used to feed the process of model creation. In most cases, an expert with significant experience defines generic and opaque behavior models that are interoperable between the different customer machines. But in certain systems, this responsibility is passed on to the users. They are then free to define their own policy, which will be more adapted to their own system since they can take into account the different installed software. On the other hand, they must be well taught and be aware of the possible repercussions of their choices.

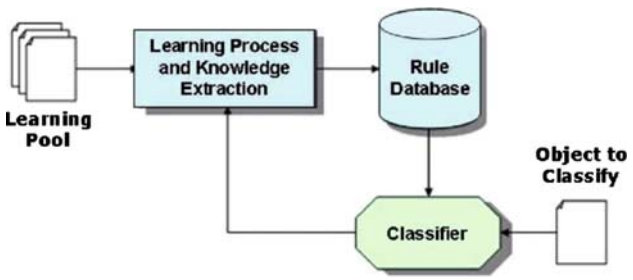


Fig. 14 Learning process. The original knowledge is extracted from a learning pool and integrated to the rule database. The rules are then evaluated by the classifier. According to their relevance, the process is iterated until stabilization of the rule set

4.3.2 Automatic learning: data mining and classifiers

The automatic generation of behavioral signatures is a critical improvement, necessary to avoid the shortcomings of the simple byte signatures. Up until now, the learning process has only been applied to certain models since the manipulated structures in a behavioral context are more complex and thus harder to learn. The learning mechanism relies on classification rules built by classifiers combined with data mining techniques. Whatever the used classifier, the general procedure remains the same. The system is first confronted to a learning pool made up of large sets of malware and legitimate samples already labelled as malicious or benign. The size of the pool must be well chosen and sufficiently important to exhibit no bias. Like any learning process, the generation of behavioral signatures remains very sensitive to noise injection in the training pool. Some effective attacks have already been published against similar worm signature generators [58]. During the training period, the classifier crawls into this data repository to extract common properties between the different considered classes. In a behavioral context, the extraction of these common properties relies on three major paradigms which are briefly described (for further information, relevant references are given) (Fig. 14):

```
Mail worm::=Call.Connect() ∧ Call.Send() ∧ (¬Call.Receive()) ∧
            String.HELLO ∧ String.MAILString.RCPT
```

Fig. 15 Boolean expression of the e-mail worm class. The following rules determine the characteristics (system calls and specific strings) common to the different mail worms. The main difference with a legitimate mail client lies in the fact that the worm does not try to receive data since it does not wait for any acknowledgement message or response

```
P(OpenFile|Benign) = 95%          | P(OpenFile|File infector) = 100%
P(GetModuleHandle|Benign) = 20%   | P(GetModuleHandle|File infector) = 70%
P("*.exe"|Benign) = 10%           | P("*.exe"|File infector)=90%
```

Fig. 16 Statistics for file infectors. These results are only given as examples. However, they bring into light the prevalence of certain characteristics. Opening a file on its own is insufficient to decide of the action nature as it is widely used by both benign and infector programs. On the contrary, accessing the handle of the current module to copy this image in a target is more significant of an infection

Rules induction: This first paradigm specifies the belonging conditions for the different classes of behavior. For each sample received by the classifier, it integrates or removes certain characteristic data in the condition in order to preserve the class consistency [59–61]. Such rules are often formulated as Boolean expression as pictured in Fig. 15 or as decision trees [62].

Bayesian statistics: The second paradigm based on statistics is used in classifiers like Bayesian networks. For each collected characteristic, the probability of finding it in a given class of malware is measured [60–62]. Figure 16 describes examples using system calls and strings as collected data. Ultimately, only the results exhibiting the most important powers of discrimination are kept. An important criterion in this choice is the minimal overlapping of the characteristics in the different classes. The ideal case would obviously be when a characteristic exists with a probability of 100% in a unique class whereas it is absent of any other.

Clustering: The third paradigm relies on predefined cases. During the learning procedure, average profiles are built for each class of malware. When deployed, the classifiers measure a distance between the profiles and the tested programs [63]. The program is classified according to the profile with which it exhibits a minimal distance. The method used to measure this distance may vary from a system to another but it remains a factor impacting heavily on the classification accuracy. Figure 17 gives an example where the distance is calculated on the basis of the number of modifications necessary to pass from a call sequence to another.

5 Panorama of existing behavioral detectors

As an illustration, we have classified several existing behavioral detectors according to the elements of our taxonomy. The result is given in Table 1 completed with additional practical information about their usage, their environment as well as their target. The detectors have been separated into two parts, the first one for the research prototypes and the second for known commercial products. This table has been built according to the information made available by the different editors, which are sometimes very limited.

The main trend brought into light is that most commercial systems are based either on heuristic algorithms with sandboxing or real-time expert systems. It can be explained by the fact that the diverging research prototypes often require too much resource or do not exhibit sufficiently low error rates. These prototypes remain mainly used by researchers and analysts until their optimization. This is particularly true for static analysis which is currently used only for analysis and signature extraction but not for detection. A second observation, that was also visible through the referenced papers, is the convergence of the antivirus products using behavioral detection with host-based intrusion prevention systems (HIPS). It becomes less and less obvious to draw a clear demarcation line between the two. This is not really surprising since virology and intrusion detection are connected security domains.

6 Conclusions

The main idea to retain of this paper is that under the terms of behavioral detection lies a whole set of heterogeneous techniques relying on a common principle: the identification of the functionalities. In particular, we observe in the taxonomy a clear distinction between simulation-based verification and formal verification which are directly linked to the dynamic and static modes. Yet, these modes are complementary as they exhibit opposite strengths and weaknesses.

Several researchers have already thought of means to combine the static and dynamic modes in order to take advantage of their respective assets. Dynamic analysis makes it

Operation	Profile	Capture	Cost
Insert		RegWriteKey	1,5
*	WriteFile	WriteFile	0
Delete	CreateProcess		0,7
*	RegWriteKey	RegWriteKey	0
Replace	RegWriteKey	RegReadKey	0,1
Replace	Send	Receive	2,0
Distance			4,3

Fig. 17 Distance between traces. Two call sequences from a profile and a capture are compared in this table. Each operation required to pass from one to another is associated to a different cost. The final distance is equal to the addition of these costs

Table 1 Classification of existing behavioral detectors

Name (origin)	Date	Reference	Capture	Input	Target	Engine type	Usage	Environment
TBScan (N/C)	1994	[33]	Dyn.(SB)	Interruptions	File infectors	Heuristic algorithm (flags)	Det.	Ms DOS
VIDES (Unv. Namur & Hamburg)	1995	[37]	Dyn.(RT)	Interruptions	COM and EXE infectors	Deterministic finite automata	D./C.	Ms DOS
N/C (Unv. Columbia & N.Y.)	2003	[60]	Static	Imported functions, strings	All kinds of malware	Data mining and classifier	Det.	Win
GateKeeper (Florida Inst. of Tech.)	2004	[23]	Dyn.(RT[a])	System calls	All kinds of malware	Heuristic algorithm (weight)	Det.	Win
N/C (Unv. Camegie et al.)	2005	[46]	Static	Control flow graphs	Polymorphic mail worms	Semantically annoted graph isomorphism	Det.	Win
N/C (Unv. Munich)	2005	[57]	Static	Control flow graphs	Worms	Model checking	Det.	Win
N/C (Unv. Liverpool)	2006	[52]	Static	Algebraic program abstraction	Metamorphic viruses	Equivalence by reduction	Det.	IA32
TTAnalyze (Technical Unv. Vienna)	2006	[26]	Dyn.(VM)	System calls	All kinds of malware	Simple activity log	Class.	Win
N/C (Microsoft Corp.)	2006	[63]	Dyn.(VM)	System calls	All kinds of malware	Data mining and classifier	Class.	Win
N/C (Unv. California & Vienna)	2006	[64]	Dyn./Stat.	COM and system calls	Web client spywares	Expert system	Det.	Internet Explorer
ThreatSense - NOD32 (Eset)	N/C	[65]	Dyn.(SB)	Instructions associated to actions	All kinds of malware	Heuristic algorithm	Det.	Win/Linux/ FreeBSD
AVG Anti-Virus (Grisoft)	N/C	[66]	Dyn.(SB)	Instructions associated to actions	All kinds of malware	Heuristic algorithm	Det.	Win/Linux/ FreeBSD
ViGUARD (Softed)	N/C	[67]	Dyn.(RT)	System calls	All kinds of malware	Expert system (user's decision)	Det.	Win
B-HAVE - Bit Defender (Softwin)	N/C	[68]	Dyn.(SB)	Instructions associated to actions	All kinds of malware	Heuristic algorithm	Det.	Win/Linux/ FreeBSD
Bloodhound - Norton (Symantec)	1997	[35]	Dyn.(SB)	Instructions associated to actions	File infectors	Heuristic algorithm	Det.	Win
Entercept (Mc Affee)	2004	[69]	Dyn.(RT)	System calls	All kinds of malware	Expert system (predefined policy)	Det.	Win/Linux
Safe'n'Sec Antivirus (Safen Soft)	2004	[70]	Dyn.(RT)	System calls	All kinds of malware	Expert system (predefined policy)	Det.	Win/Linux/ FreeBSD
TruPrevent (Panda Software)	2006	[71]	Dyn.(RT)	System calls	All kinds of malware	Heuristic algorithm	Det.	Win/Linux
Virus Keeper (AxBa)	2007	[72]	Dyn.(RT)	System calls	All kinds of malware	Expert system (user's decision)	Det.	Win

RT real-time, *SB* SandBox, *VM* virtual machine

[a] Actions recording, for the system usage: *Det.* detection, *Class.* classification

possible to determine a reduced perimeter where a static analysis would be worth deploying. Based on this principle, a system has already been put forward in order to detect spyware parasiting web browsers [64]. The dynamic phase is used to find the processing routines associated to the different web events. Once localized a static analysis is deployed to detect any malicious activity. Generally speaking, a static analysis could be deployed at each reached branching to explore the alternative execution paths that will not be executed.

To combine efficiently both modes, it remains necessary to evolve towards a common model of reference. This model could then be refined according to the class of system considered, while remaining compatible with others. Unfortunately, such a model is still missing.

Acknowledgments We want to thanks specially Pierre Crégut whose valuable remarks helped greatly to synthesize this taxonomy. We would also like to thank the anonymous reviewers for their interesting comments which helped to improve this paper.

References

1. Cohen, F.: Computer viruses. Ph.D. thesis, University of South California (1986)
2. Cohen, F.B.: Computer viruses: Theory and experiments. Comput. Secur. **6**(1), 22–35 (1987)
3. Debar, H., Dacier, M., Wespi, A.: Towards a taxonomy of intrusion-detection systems. Comput. Netw. Spl Issue Comput. Netw. Secur. **31**(9), 805–822 (1999)
4. Mé, L., Morin, B.: Intrusion detection and virology: an analysis of differences, similarities and complementariness. In: Bonfante, G., Marion, J.-Y. (eds.) J. Comput. Virol., vol. 3, no. 1, WTCV'06 Special Issue, pp. 39–49 (2007)
5. Anderson, J.: Computer security threat monitoring and surveillance. Tech. rep., James P. Anderson Company (1980)
6. Denning, D.: An intrusion–detection model. IEEE Trans. Softw. Eng., vol. SE-13 (1987)
7. Warrender, C., Forrest, S., Pearlmutter, B.: Detecting intrusion using system calls: Alternative data models, In: Proceedings of IEEE Symposium on Security and Privacy, pp. 133–145 (1999)
8. Zanero, S.: Behavioral intrusion detection. In: Proceedings of the 19th International Symposium on Computer and Information Sciences (ISCIS), pp. 657–666 (2004)
9. Filiol, E.: Computer viruses: from theory to applications. Springer, Heidelberg, IRIS Collection (2005). ISBN:2-287-23939-1
10. Fortinet observatory. http://www.fortinet.com/FortiGuardCenter/
11. Malware outbreak trend report: Storm-worm, Commtouch Software Ltd (2007). http://www.commtouch.com/downloads/Storm-Worm_MOTR.pdf
12. Filiol, E.: Malware pattern scanning schemes secure against black-box analysis. In: Broucek, V., Turner, P. (eds.) J. Comput. Virol., vol. 2, no. 1, EICAR 2006 Special Issue, pp. 35–50 (2006)
13. Filiol, E.: Techniques Virales Avancées. Springer, Heidelberg, IRIS Collection (2007). ISBN:2-287-33887-8
14. Sz'or, P.: The Art of Computer Virus Research and Defense. Addison-Wesley, Reading (2005). ISBN:0-321-30454-3
15. Spinellis, D.: Reliable identification of boundedlength viruses is np-complete. IEEE Trans. Inf. Theory **49**, 280–284 (2003)
16. Filiol, E.: Metamorphism, formal grammars and undecidable code mutation. In: Proceedings of the International Conference on Computational Intelligence (ICCI), Published in the Int. J. Comput. Sci., vol. 2, issue 1, pp. 70–75 (2007)
17. Christodorescu, M., Jha, S.: Testing malware detectors, In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 34–44, ACM Press, New York (2004)
18. Josse, S.: How to assess the effectiveness of your anti-virus? In: Broucek, V. (ed.) J. Comput. Virol., vol. 2, no. 1, EICAR 2006 Special Issue, pp. 51–65 (2006)
19. Filiol, E., Jacob, G., Liard, M.L.: Evaluation methodology and theoretical model for antiviral behavioural detection strategies. In: Bonfante, G., Marion, J.-Y. (eds.) J. Comput. Virol., vol. 3, no. 1, WTCV'06 Special Issue, pp. 23–37 (2007)
20. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. In: Proceedings of the European Symposium on Research in Computer Security, pp. 326–343 (2003)
21. Hoglund, G., Butler, J.: Rootkits, Subverting the Windows Kernel. Addison-Wesley Professional, Reading (2006). ISBN: 0-321-29431-9
22. Vivanco, A.D.: Comprehensive non-intrusive protection with data-restoration: A proactive approach against malicious mobile code. Master's thesis, Florida Institute of Technology (2002)
23. Wagner, M.E.: Behavior oriented detection of malicious code at run-time. Master's thesis, Florida Institute of Technology (2004)
24. Norman's sandbox malware analyzer. Norman ASA. http://www.norman.com/microsites/malwareanalyzer/fr/
25. Cwsandbox. Sunbelt Software. http://www.cwsandbox.org
26. Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic analysis of malicious code. In: Broucek, V., Turner, P., (eds.) J. Comput. Virol., vol. 2, no. 1, EICAR 2006 Special Issue, pp. 67–77 (2006)
27. Rutkowska, J.: Red pill... or how to detect vmm using (almost) one cpu instruction (2005). http://invisiblethings.org/papers/redpill.html
28. Ferrie, P.: Attacks on virtual machine emulators. In: Proceedings of the AVAR Conference (2006)
29. Debbabi, M.: Dynamic monitoring of malicious activity in software systems. In: Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS) (2001)
30. Nachenberg, C.: Behavior blocking: The next step in anti-virus protection, SecurityFocus, 2002. http://www.securityfocus.com/infocus/1557
31. Schmall, M.: Classification and identification of malicious code based on heuristic techniques utilizing meta-languages. Ph.D. thesis, University of Hamburg (2002)
32. Schmall, M.: Heuristic techniques in av solutions: An overview, SecurityFocus (2002). http://www.securityfocus.com/infocus/1542
33. Veldman, F.: Heuristic anti-virus technology. In: Proceedings of the International Virus Protection and Information Security Council (1994)
34. Zwienenberg, R.: Heuristics scanners: Artificial intelligence? In: Proceedings of the Virus Bulletin Conference, pp. 203–210 (1994)
35. Understanding heuristics: Symantec bloodhound technology. Tech. rep., Symantec White Paper Series, vol. XXXIV (1997)
36. Glover, F.W., Kochenberger, G.A.: Handbook of Metaheuristics. Springer, Heidelberg (2003). ISBN:1-402-07263-5
37. Charlier, B.L., Mounji, A., Swimmer, M.: Dynamic detection and classification of computer viruses using general behaviour patterns. In: Proceedings of the Virus Bulletin Conference (1995)
38. Sekar, R., Bendre, M., Bollineni, P., Dhurjati, D.: A fast automaton-based approach for detecting anomalous program behaviors.

In: Proceedings of IEEE Symposium on Security and Privacy, pp. 144–155 (2001)
39. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages and Computation, 2nd edn. Addison Wesley, Reading (1995). ISBN:0-201-44124-1
40. Mazeroff, G., Cerqueira, V.D., Gregor, J., Thomason, M.G.: Probabilistic trees and automata for application behavior modeling. In: Proceedings of the 43rd ACM Southeast Conference (2003)
41. Kaspersky, K.: Hacker Disassembling Uncovered, 2nd edn. A-LIST, LLC (2007). ISBN:1-931-76964-8
42. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Tech. rep., Technical Report 148, Department of Computer Science, University of Auckland (1997)
43. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium, pp. 18–18 (2004)
44. Josse, S.: Secure and advanced unpacking using computer emulation, extended version from the avar conference. J. Comput. Virol. **3**(3), 221–236 (2007)
45. Preda, M.D., Christodorescu, M., Jha, S., Debray, S.: A semantic-based approach to malware detection. In: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2007)
46. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantic-aware malware detection. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 32–46 (2005)
47. Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In: Proceedings of the Conference on the Detection of Intrusions and Malwares and Vulnerability Assessment (DIMVA), pp. 129–143 (2006)
48. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: International Symposium on Recent Advances in Intrusion Detection (RAID) (2005)
49. Periot, F.: Defeating polymorphism through code optimization. In: Proceedings of the Virus Bulletin Conference, pp. 142–159 (2003)
50. Bruschi, D., Martignoni, L., Monga, M.: Using code normalization for fighting self-mutating malware. In: Proceedings of the International Symposium on Secure Software Engineering, pp. 37–44, IEEE CS Press (2006)
51. Webster, M.: Algebraic specification of computer viruses and their environments. In: Selected Papers from the First Conference on Algebra and Coalgebra in Computer Science Young Researchers Workshop (CALCO-jnr 2005), University of Wales Swansea Computer Science Report Series (CSR 18-2005), pp. 99–113 (2005)
52. Webster, M., Malcolm, G.: Detection of metamorphic computer viruses using algebraic specification. J. Comput. Virol. **2**(3), 149–161 (2006)
53. Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M.M., Lavoie, Y., Tawbi, N.: Static detection of malicious code in executable programs. In: Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS) (2001)
54. Singh, P., Lakhotia, A.: Static verification of worm and virus behavior in binary executables using model checking. In: Proceedings of the IEEE Information Assurance Workshop, pp. 298–300 (2003)
55. Clark, E., Grumberg, O., Long, D.: Model Checking. MIT Press, Cambridge (1999). ISBN:0-262-03270-8
56. Schnoebelen, P.: The complexity of temporal logic model checking. Adv. Modal Logic **4**, 393–436 (2003)
57. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. Lect. Notes Computer Sci. **3548**, 174–187 (2005)
58. Perdisci, R., Dagon, D., Fogla, P.W.L., Sharif, M.: Misleading worm signature generators using deliberate noise injection. In: Proceedings of IEEE Symposium on Security and Privacy (2006)
59. Lee, W., Stolfo, S., Chan, P.: Learning patterns from unix process execution traces for intrusion detection. In: Proceedings of the AAAI97 Workshop on AI Approaches to Fraud Detection and Risk Management, pp. 50–56. Addison Wesley, Reading (1997)
60. Schultz, M.G., Eskin, E., Zadok, E.: Data mining methods for detection of new malicious executables. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 38–49 (2001)
61. Wang, J.-H., Deng, P.S., Fan, Y.-S., Jaw, L.-J., Liu, Y.-C.: Virus detection using data mining techniques. In: Proceedings of IEEE on Security Technology, pp. 71–76 (2003)
62. Kolter, J., Maloof, M.: Learning to detect malicious executables in the wild. In: Proceedings of the 2004 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 470–478. ACM Press, New York (2004)
63. Lee, T., Mody, J.: Behavioral classification. In: Proceedings of EICAR (2006)
64. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.: Behavior-based spyware detection. In: Proceedings of the 15th USENIX Security Symposium (2006)
65. Frost&Sullivan, Protection en temps réel contre toutes les menaces, Tech. Rep., White Paper Eset
66. Avg anti-virus. Grisoft. http://www.grisoft.com/doc/39/lng/fr/tpl/tpl01
67. Viguard. Softed. http://www.viguard.com/detail_163_logiciel_antivirus_viguard-platinium#
68. Bitdefender antivirus technology, Tech. Rep., BitDefender White Paper
69. Host and network intrusion prevention, competitors or partners? Tech. rep., Mc Affee White Paper (2004)
70. Safe′n′sec antivirus. Safen Soft. http://www.safensoft.com/technology/
71. Truprevent. Panda Software. http://www.pandasoftware.com/products/truprevent_tec.htm?sitepanda=particulares
72. Virus keeper. AxBa. http://www.viruskeeper.com/fr/faq.htm