Behavioral Detection of Malwares: From a Survey Towards an Established Taxonomy

Grégoire Jacob[1/2], Hervé Debar[1], Eric Filiol[2]

[1] France Télécom R&D, Caen, France
{gregoire.jacob|herve.debar}@orange-ftgroup.com

[2] French Army Signal Academy,
Virology and Cryptology Lab., Rennes, France
eric.filiol@esat.terre.defense.gouv.fr

Abstract

The behavioral detection differs from the appearance detection in that it relies on an identification of the actions performed by the malware rather than syntactic markers. Identifying these malicious actions and interpreting their final purpose is a complex reasoning process. In this paper, we draw up a survey of the different reasoning techniques we have come across in several behavior-based detection systems. These systems have been classified according to a new taxonomy introduced inside the paper. This taxonomy divides the behavior-based detectors into distinct families according to the data collection conditions, the capture interpretation, the adopted model and generation process of the behavior signatures as well as the decision support.

Keywords: Malware – Behavioral detection – Behavior models – Dynamic monitoring – Static analysis.

1 Introduction to the behavioral principles

Even if the behavioral detection seems a recent trend in antivirus products as well as in the virology research area, its principles are not really new. In 1986, F. Cohen already established the basis of behavior-based detection within his first formal works [1, 2]. He put forward the fact that viruses, just like any other running program, use the services provided by the system. The prediction of the viral nature of a program according to its behavior was then equivalent to defining what is and what is not a legitimate use of the system services. He finally found that this problem could be reduced to an appearance analysis of the request inputs sent to the system and was thus undecidable. If this first definition seems strongly linked to the operating system, it can easily be extended to the use of any hardware or software resource such as the processor, the memory or programs. This larger definition is often referred as function-based detection. Along this article we will use them indifferently as it is just a question of system perimeter.

1.1 Two opposite approaches of the problem

As stated by Cohen, there are two opposite approaches to apprehend the problem. The first one is to model the behavior of legitimate programs and measure deviations from this reference. This approach provides the great advantage of being able to detect completely unknown viral strains. Nevertheless, the task of modelling a global behavior for programs reveals itself extraordinary complex. An obvious reason is the multitude of applications of different nature existing on a system. A web or mail client will exhibit an intensive use of the network facilities whereas a multimedia player will decode large buffers of data and render them over physical devices such as the graphic or sound cards. No common characteristics can be extracted and a different profile would be required for each kind of application. Moreover the information available is too important for each program (several mega bytes 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 opposite approach of modelling and detecting suspicious behaviors is mainly adopted. When a set proves too complex to be defined exhaustively, the intuitive and easier approach to address the problem is to work on its complementary. The main drawback is that we can no longer detect unknown malwares 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, behavior-based detection is based on legitimate models whereas suspicious models as in virology are considered as simple signatures for knowledge-based or 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 the reader is invited for further information to refer to the works of Forrest et al. on host-based intrusion detection [7] and the use of Markovian Models for capturing legitimate uses of systems described by S. Zanero [8]. To go back to our main focus, viral techniques are less numerous than vulnerabilities and misuse models seem more adequate to our problem of malware detection.

1.2 Paper organization

We have willingly adopted the virology point of view and will thus implicitly consider the modelling of suspicious behaviors all along our speech. In certain particular cases, relevant intrusion detection papers will also be given as additional references. Originally, this paper was motivated by a simple observation. There is no global survey covering this domain whereas we observe an increasing activity both in commercial products and research. The multitude of behavioral detection systems is striking, and so is the inconsistency in the vocabulary and the designations used. We have thus decided to make the scope of our survey as wide as possible, in a way relevant to the stated definition, in order to establish a common base for our taxonomy. To achieve this, we have organised this paper as follows: Section 2 explains the recent interest in behavioral detection by the predicted failure of appearance detection, Section 3 describes a generic behavior based detection system, Section 4 introduces the taxonomy, and Section 5 illustrates our speech with an overview of both existing commercial products and research prototypes.

2 Why behavioral detection may succeed where appearance detection will undeniably fail

Historically, appearance detection also called form-based detection have been the first used to fight against malwares and still remain at the heart of nowadays antiviruses. Their functioning principle is the search in files for suspicious byte patterns stored in a signature base. These betraying patterns must exhibit a discriminating character combined with non-incriminating properties for legitimate programs [9, pp.147]. As a consequence, these form-based techniques are bound to detect known malwares contrary to the behavioral detection. On the other hand, it can identify the threat more precisely and name it whereas the behavioral approach can not.

2.1 The signature extraction problem

Form-based detection provides undeniable advantages for operational use. It uses optimized pattern matching algorithms whose complexity is controlled and results exhibit very low false positive rates. Unfortunately, it proves completely overwhelmed by the quick evolution of the viral attacks. The bottleneck in the detection process lies in the signature generation and distribution after the discovery of a 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 minutes, attacks and protection does 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 important, 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. An even more relevant example, which was a major concern during the last RSA Security Conference in San Fransisco, is 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 cause, formal works led by E. Filiol underline the ease of signature extraction by a simple black box analysis because of weak signature schemes [12]. A second side effect is the alarmingly growing size of the signature bases. As a solution, older signatures are regularly removed leaving the system once again vulnerable.

Behavior signatures are no longer simple byte patterns but carry a semantic interpretation. As a consequence, they prove to be more generic and thus resilient to simple modifications. A single behavior signature should then detect all malware versions coming from a common strain. Experts could then establish more easily a hierarchy in their work, focusing uppermost on new innovative strains. The behavior base size should be less consequent as well and the signature distribution less frequent. Regular base updates remain nevertheless necessary contrary to what is claimed in certain marketing speech.

2.2 Resilience to automatic mutations

In the last part, we have consider the manual evolution of malware but what happens when these mutations become automatic during propagation. The first significant generation of mutation engines is born with polymorphism [13, pp.140][14, pp.252]. Polymorphic malwares have their entire code ciphered in order to conceal any potential signature. A simple variation of the ciphering key modifies totally the byte sequences of the virus. A deciphering routine is required to recover the original code and execute it. This routine must possess its own mutation facilities if it wants to avoid becoming a signature on its own.

It was quickly discovered that simple emulation could make the original code available, thus thwarting these engines. But searching for signature has become far more complex with metamorphism. The malware is not simply ciphered but its whole body suffers a certain number of transformations affecting its form while keeping its global functioning [13, pp.148][14, pp.269]. The mutation phenomenon always begins with the code disassembly, which is then obfusctated before to be reassembled: code reordering, garbage insertion, register reassignment and equivalent instruction substitution. Syntactic analysis is no longer sufficient to fight against these mutations. Eventually, D. Spinellis has shown that the detection of mutating bounded virus by signature is NP-complete [15].

If these mutations modify the malware syntax, they do not modify its semantic. Typically, the malware will always use the system services and resources in an identical way. So behavioral approaches should not be affected by these modifications. As a conclusion, all these reasons we have set forth lead to consider behavioral detection as a promising alternative solution to malware detection.

3 Generic description of a behavioral detector

3.1 System architecture and functioning

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

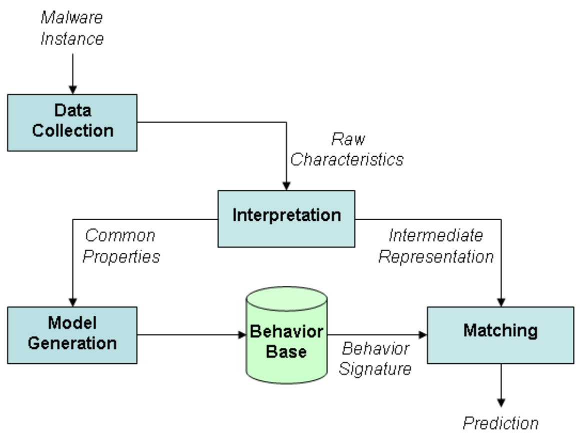

Figure 1: Generic description of a behavior-based detector. This decomposition of the system brings into light the articulation between the first step of behavior model generation and the three sequential steps of detection, each one processing the data to a higher level of interpretation until the final assessment.

Behavioral detectors can basically be split into four main components. The detection process consists in three sequential tasks addressed by individual components as shown in Figure 1. Prior to detection, an initial step is required to generate the behavior signatures stored in a dedicated database. The signature generation according to the adopted model may not be an actual software component but requires a dedicated process since it is a key element in the detection efficiency. Coming back to detection, a first step is performed with the data collection where we have considered indifferently dynamic capture and static extraction as 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. The collected data can be gathered from different sources: the local host for personal computers or from host honeypots deployed in strategic points over networks. As behavioral detectors work at a higher interpretation level than simple appearance detection, collected data need to be analyzed and interpreted in a second step. This step brings into light the important characteristics of the sample and format them into an intermediate representation to feed the last part of the process. The last step consist in a matching algorithm comparing the representation to the behavior signatures. According to the result, the program will be labelled as malicious or benign.

3.2 Basic properties for assessment

It is fundamental to define the important properties of a behavioral detection system since they will provide the basis for efficiency assessment. Actual certifications simply confront malware detectors to known viral strains thereby assessing solely appearance detection. Assessing antivirus product is still an open problem and several more complete test procedures have been put forward [16, 17]. One of them focus more particularly on behavioral detection using functional metamorphism [18]. This new kind of mutation generates new viral strains using different known techniques to achieve a same final behavior that should be detected. The results have notably shown that, in order to make up for the false positive rates, the behavioral detection is often confronted to an additional detection by signature. If a neat decrease is observed, the behavioral detection remain severely hindered by this measure which prevents the detection of unknown strains using known viral techniques. More generally, Any test procedure to be complete with regards to behavior detection should at least consider the following properties:

- **Performance.** Performance is determined by the overload introduced by the detection system in its environment. This overload depends on various factors such as the capture conditions or the complexity of the matching algorithm. It can be measured by comparison of the resources used between a normal execution and a detection process. This is an important property since it is a cause for the belated interest in behavioral detection. For several years, the calculation power of processors, the available memory space and bandwidth prove to be insufficient in order to deploy such complex techniques. In the case of dedicated honeypots, the performance constraints are even heavier than for personal computers since the system must cope with the incoming traffic.

- **Completeness and Accuracy.** A system who fails to detect too many malwares is said incomplete because its false negative rate is too high. These failure 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 on this properties: the soundness of the chosen signatures and the relevance of the collected data.

- **Adaptability.** When a system is deemed inaccurate or incomplete, modifications must often be performed on the behavior signatures. Adaptability traduces the ease of these updates allowed by 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 static and dynamic detection.

- **Fault-tolerance, Unobtrusiveness and Timeliness.** These additional properties are more specific to dynamic detection because they presuppose that the malware is active during the detection process. Fault-tolerance assess the capability of the behavioral detector to stand up to any external perturbation and in particular intended attacks launched by the malware. On the opposite, according to the principle of the physicist Schrödinger, the observation of the malware behavior must not introduce perturbations in its execution. Unobtrusiveness guarantees that the observed behavior will not be altered by the analysis. At last, timeliness checks whereas the detection is reached before the damages done to the environment by the malware are irreversible.

4 Taxonomy of behavioral detector

The concepts we use to classify behavior-based systems derive directly from their generic description. The main axes described in Figure 2 correspond to the components forming the detector. As a matter of fact, every combination of components is not possible. Different models and algorithms are used whether the input data is collected dynamically or extracted statically. We will now describe individually the different elements of the taxonomy.

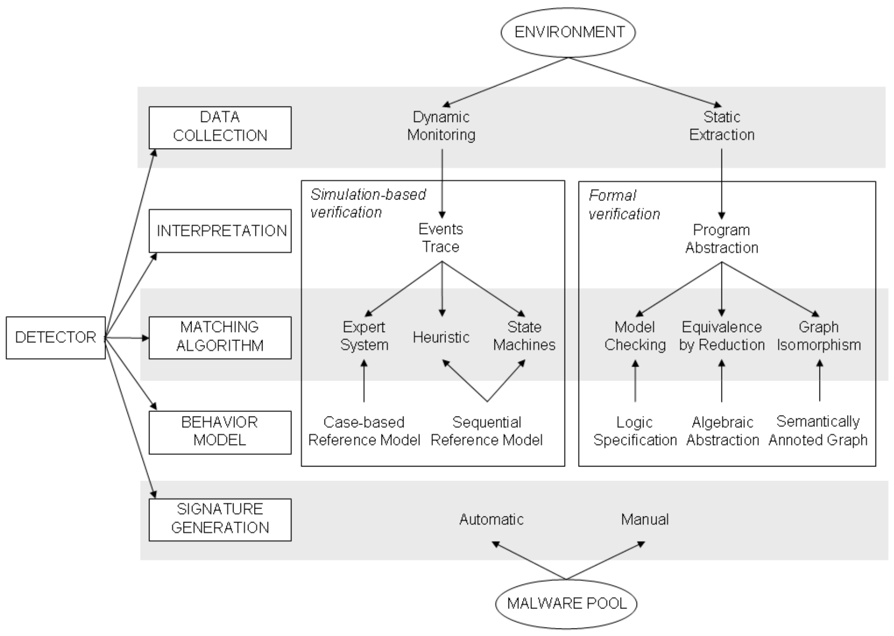

Figure 2: Characteristics of behavioral detection systems. We can observe that the classification is globally divided into two branches corresponding to the dynamic and static modes. This picture put forward the importance of the collection method as it will impact strongly the models and algorithm used thereafter.

4.1 Capture conditions and nature of the data

Behavioral detection is traditionally associated to a dynamic execution. This seems a short-sighted view to us since these behaviors are originally written down in the malware code. Thereby, the malware actions can also be discovered through a static analysis. We have willingly considered both operative modes and compared them.

4.1.1 Dynamic monitoring

Detection of a malware during its execution must rely on elements observable from an external agent. On former operating systems, the interception of interruptions was the first source of information about the resource use made by a program. It has been progressively replaced by the interception of system calls with the apparition of 32 bits systems. The main interest lies in the fact that system calls remain a mandatory passing point from the user space to access kernel services and objects. In their work on intrusion detection based on system calls, S. Forrest et al. underline the importance of the collected data and their representation as they will have an important influence on the analysis and the matching algorithm [7]. In our particular case, sequential representations are mainly consider but other representations like frequency spectres could be consider. 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 will betray a malicious purpose as stated by Kruegel et al. [19]. As an illustration, a simple extract from a system call trace is given in Figure 3.

```
       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
```

Figure 3: Extract from a trace of system calls. The whole trace is a made up of a list of system calls like this one with various attached information. The process identifier is important because it makes it possible to correlate the system calls from a target process.

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 available data.

Real-time Conditions: The progression of the malware is observed directly in its environment without restrictions. This type of capture is always criticized because malevolent actions are effectively executed. Timeliness is thus primary before the point of no return of the infection is reached. To intercept system calls in real-time, the main technique used by the detector is API hooking which is often used by rootkit writer as well [20]. The overload generated by the interception and the call processing may be perceptible by the user. Yet, it remains less significant than for the other capture conditions.

Real-time with action recording: This is a particular case of real-time capture where the actions taken by the observed program are recorded as well as the intermediate sates of the environment [21, 22]. This trade-off makes it possible to benefit 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 remain 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 [23, 24]. This technique popularized by JAVA, makes it possible to constrain the execution in an escape-proof memory space with low privileges and limited service accesses. The main advantage is that the external observer has a total access over the memory space and can control the execution step by step, offering better observation facilities than real-time conditions. On the other hand, the sandbox uses more significant resources since it introduces an intermediate layer between the program and its environment. To reduce the overload, only suspicious code portions of the program are analysed. Once this pre-analysis performed, the normal execution of legitimate programs is resumed without hindrance. Unfortunately, sandboxes do not provide the same facilities than real systems and can easily be detected. Debugging detection techniques checking the execution time or using error handling structures succeed easily. Once the sandbox detected, the malware can adapt its execution to seem benign whereas it is not. If the privileges and service accesses are not properly restrained, a 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 control every access point to the hardware from the unaware guest system. In case of a 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 processing can then be performed entering or returning from the system call without trace for the virtual environment that can carry on its execution [25]. In comparison to sandboxes, virtual machines make it possible to emulate any fictive resources either hardware (network connections) or software (mail or P2P clients). These resources are often used malevolently by the malware to its own profit for propagation or gathering information. Total virtualization enables the observation of these interaction without risks for the host. On the other hand, virtual machines require large amount of resources making them impossible to use in operational contexts or with restricted virtualization support to the file system. 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 [26, 27].

Whatever dynamic condition is considered, all of them 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 thus do not modify the final execution. The different versions issued of a same mutating strain will eventually provide the same event trace.

Limitations: The interception of system calls is not the ultimate solution. Certain behaviors such as ciphering do not use the system services for stealth reasons. Some malwares even redefine whole system primitives for the exact same reasons. An other phenomenon to take into account is the migration of malwares towards the system kernel in order to acquire privileges equal to antiviruses. Using these privileges, complex stealth techniques become possible since the malware can interact directly with the hardware and system objects without necessarily using any of the monitored system calls [13, pp.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 capture 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 like in the case of sandbox and virtual machine detection.

4.1.2 Static extraction

Static extraction provides richer information than dynamic monitoring which is bound to collect observable elements only. In effect, every potential action of a malware is presumably written down in its code. The 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 more complex and requires several processing steps to get an intermediate representation of the program.

Static extraction uses the traditional techniques of reverse engineering, that is to say, disassembly and building of control and data flow graphs (CFG and DFG). Certain tools can automatically achieve this process described with more details in Figure 4. This type of representation is used by a majority since it brings 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. Notice that most of malwares are often protected using packers like UPX in order to increase the extraction difficulty. Unpacking has become a challenging problem in static analysis, requiring more and more advanced techniques [28].

Just like dynamic monitoring, the intrinsic properties of static extraction provides advantages but also drawbacks. By comparing these 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. As the malware is not running during the capture, it is not able to adapt its 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 proved undecidable by A. Turing in 1936. Still, under certain conditions, the necessary information can be gathered. Anyhow, static extraction remain possible as long as disassembly can be performed. Techniques of software protection can skew the result by introducing fake instructions hindering the code alignment. Moreover static extraction remain very sensitive to the obfuscation techniques used by metamorphic engines.

Theoretical works to assess the resistance of static semantic analyzers to common transformations have already been addressed by Preda et al. [29].

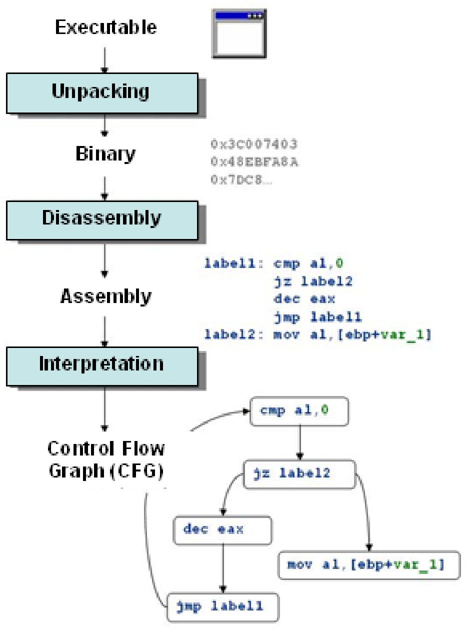

Figure 4: Incremental steps during the static extraction. This scheme describes the different processing stages applied to the program in order to extract the intermediate representation: unpacking when required, disassembly and interpretation.

4.2 Matching algorithms and models

Generally speaking, a detection engine embed a certain amount of knowledge about malwares to support its decision. In the context of behavioral detection, this knowledge is made up of a base of behavior signatures modelled according to the engine nature. During the analysis the collected data must be interpreted and formatted to make its confrontation to the behavior model easier.

To put things in perspective, the engine nature will determine the behavior modelling, the intermediate representation as well as the confrontation method. That is why we have chosen to present the interpretation and confrontation steps of the detection process at the same time. In this part we will detail the different classes of systems previously mentioned in Figure 2 according to two main axes: simulation and formal verification. The generation of the behavior signature will finally be treated as a third transversal axis.

4.2.1 Simulation-based verification

Simulation-based verification can be seen as a black box test procedure and is thus strongly linked to a dynamic mode . This kind of verification requires a simulation environment, typically one of the capture conditions introduced in 4.1.1. Only the current path is analyzed making the system work on a sequence of discrete events that will be compared to the reference model: the behavior signature. The following systems have been listed.

4.2.1.1 Expert systems

This kind of detection engine relies on a set of case-based rules modelling the experience and expertise of an analyst confronted to a particular situation. A rule will be defined like the ones pictured in Figure 5 for each known suspicious attempt to use system facilities. Every separated action taken by the observed program will be confronted to the related rules [30]. 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 of the rule-matching algorithm remain acceptable since it is equivalent to pattern matching algorithms which are in the class P.

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

Figure 5: Rules examples. 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.

The decision of whether a behavior is malicious or not must then be taken preemptively. Systems such as the one in Figure 6 enable the interception of any attempt to use a system service and make it possible to react consequently before its resolution. These proactive systems are often called "behavioral blockers" because of this property [31]. Generally speaking, this kind of engine is prone to false positives because it proves really fussy to judge the legitimacy of a separated action without correlation.

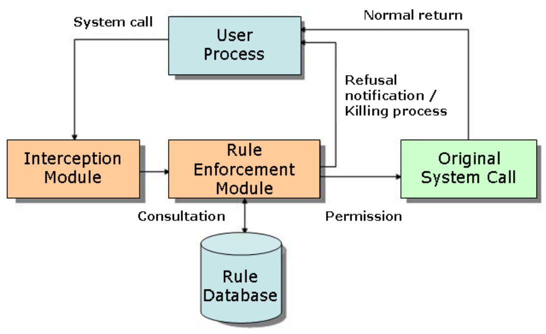

Figure 6: Rules enforcement. For each captured system call, the related rules will be scanned. According to the dedicated rule, the engine will yield the control to the originally called function or send a refusal/killing notification to the calling process.

4.2.1.2 Heuristic engines

Historically, heuristic engines were the first to be deployed to detect malicious functionalities. Contrary to the previous expert systems, the captured actions are no longer considered separately but sequentially. They functions on the basis of interruptions and system calls along with their preceding instructions defining the parameters, usually collected thanks to a sandbox. Basically, heuristic engines are made up of three parts [32, 33]:

Association mechanism: Its purpose is to label the different atomic behaviors of a malware. An atomic behavior corresponds to a functional interpretation of one or several instructions as pictured in Figure 7. Fundamentally, there exist two labelling techniques. 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 [34, 35]. The Figure 8 presents a typical example of flag-based association where atomic actions eventually corresponds to real instructions sequences.

```
Terminate program                Open File
1.  MOV AX, ??4Ch                100.  MOV AX, 023Dh
    INT 21       ;B8??4CCD21           MOV DX, ????h
2.  MOV AH, 4Ch                        INT 21       ;B8023DBA????CD21
    INT 21       ;B44CCD21       101.  MOV DX, ????h
3.  MOV AH, 4Ch                        MOV AX, 023Dh
    MOV AL, ??h                        INT 21       ;BA????B8023DCD21
    INT 21       ;B44CB0??CD21
4.  MOV AL, ??h
    MOV AH, 4Ch
    INT 21       ;B0??B44CCD21
```

Figure 7: Atomic behaviors. This example is quoted from the documentation of the Bloodhound engine [36]. It illustrates the association between several instructions sequences and a final atomic action.

Database of rules: This database defines the detection criterion. In the case of weight-based systems, there is a unique rule consisting in a threshold above which the accumulation of malicious behaviors betrays a malware. Otherwise, rules consist in flag sequences. These sequences are brought together as a detection tree like in Figure 9.

Detection strategy: The strategy impacts the detection process with regards to the progression among the rules. In the case of a weight-based association, the strategy is the accumulation function chosen to correlate the captured values. Otherwise, the strategy determines the tree search algorithm. Several kind of algorithm exist: greedy as in Figure 9, genetic, taboo or simulated annealing [37]. The choice of the strategy is primordial since it will allow to find approaching but still satisfactory values in reasonable times for problems of NP-complete complexity [13, pp.67].

```
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
```

Figure 8: Behavior base. This example has been extracted from the base of the TBScan engine [34]. Each behavior is associated to a flag carrying a semantic value.

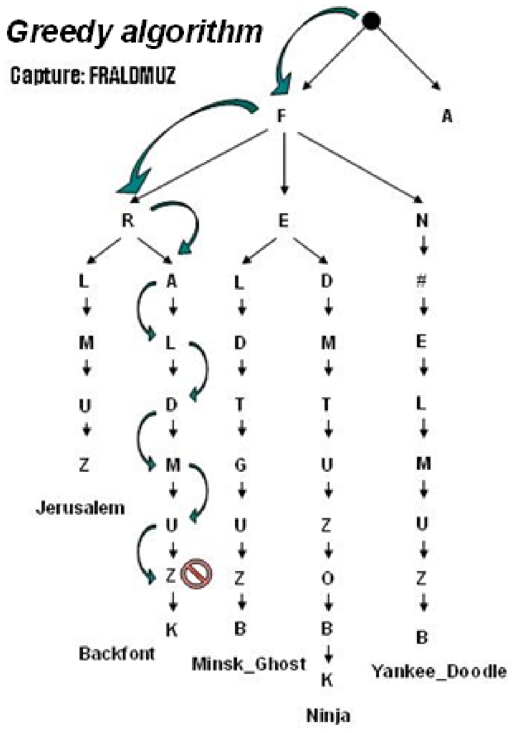

Figure 9: Detection rules and strategy. The tree has been built according to five rules from the TBScan engine, corresponding to five viral strains [34]. The chosen strategy is a simple greedy algorithm where the first valid path is always taken whitout possibilities to go back. This combination fails to detect Backfont but an other strategy where backward movement are possible would have detected the virus Jerusalem. This observation once again underlines the importance of the strategy.

4.2.1.3 State machines

Just like heuristic algorithms, 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 [38, 39]:

- The states S of an automaton corresponds to the internal states of the malware along its lifecycle,

- The set of input symbols Σ made up of 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 signal when a suspicious behavior has been detected.

From an initial state, the collected data are evaluated step-by-step making the automaton progress. If during its progression, the automaton reaches an accepting state, a malicious behavior has been discovered. Otherwise, if the automaton reaches an error state or does not progress until a final step, only behaviors supposed legitimate have been captured. The Figure 10 gives an example of automaton detecting a file infection mechanism. In state machines, the matching algorithm is defined by the problem of word acceptance problem by an automaton. Using deterministic finite automata, this problem remains NP-complete [40].

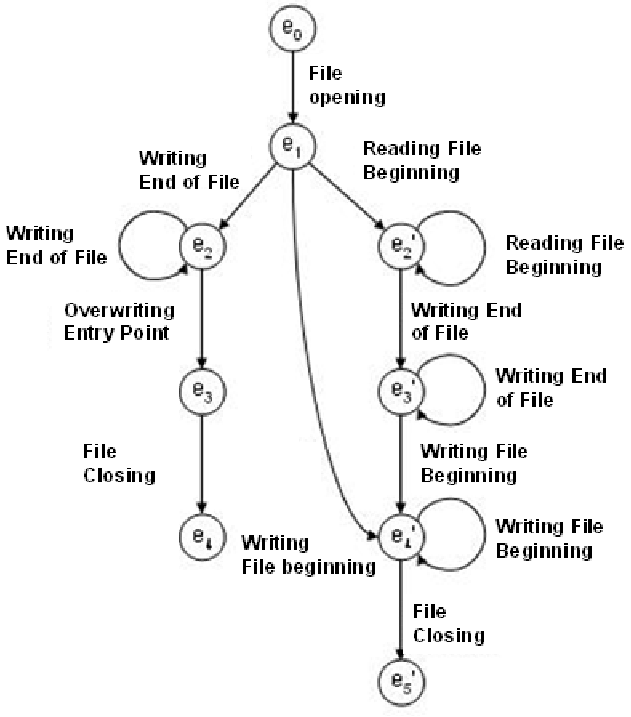

Figure 10: Automaton of the infection mechanism. This automaton describes two types of file infection. The left branch depicts the "append" infections where the viral code is copied at the end of the file and the entry-point is redirected. The right one depicts the "prepend" infection, destructive or 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.

Notice that state machines can also be used for the opposite approach modelling legitimate behaviors. But the considered automaton is no longer deterministic but probabilistic. The probabilities of the different transitions are based on the frequency of certain system call sequences during an healthy execution [41]. Unfortunately, the model put forward is used to detect macroviruses and consequently targets a specific type of application: office softwares. It remains almost impossible to extend generically legitimate models to every application.

4.2.2 Formal verification of properties

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. Enabled by the white box approach, these systems work on combinatorial explorations of the different execution paths. Only few systems have been referenced here since it remains a recent approach.

4.2.2.1 Annoted graph isomorphim

This kind of detection works exclusively with static extraction since it uses control flow graphs. The analysis relying on a higher level of abstraction than simple assembly code, the instructions stored in the nodes of the extracted graphs will be replaced by an associated label. The label attribution may follow two approaches: either the instructions are translated into an intermediate representation carrying a semantic value [42, 29] or only the classes of the instructions are stored (arithmetic, logic, function call, ...) [43, 44]. A behavior will thus be specified by a template with a similar graph structure using the same annotation mechanism. The Figure 11 (a) provides an outlook of a behavior template graph with its semantic labels made up of symbolic instructions, variables and constants.

Detection by checking that a program satisfies a template is equivalent to find a subgraph of its CFG which is isomorphic with the behavior graph. The localisation of this subgraph in a stand-alone malware may be easy to determine but it proves much more complex for a program infector since it requires to find out the insertion point first. The algorithm will then begin with associating the nodes from the CFG with those of the template as pictured in Figure 11. An additional contraint steps in since a sensible correspondence must be possible between the labels from the graph nodes. When adopting a semantic representation for labels, this association will eventually determine the equivalences between the symbolic elements (variables, constants) and the real values (registers, memory locations). An additional step is then required 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, most of CFG nodes exept in the case of indirect jumps and function returns have a bounded number of successors, typically one or two. This kind of algorithm remains very sensitive to mutation techniques and in particular to any modification impacting the resulting graph: code permuation or injection of useless instructions (dead code hidden behind opaque predicate, additional intermediate variables). These transformations can partially be addressed by optimization techniques developped for compilers [45, 46, 43]. The ultimate goal would be to reach a canonical and minimal form to revert most of mutation.

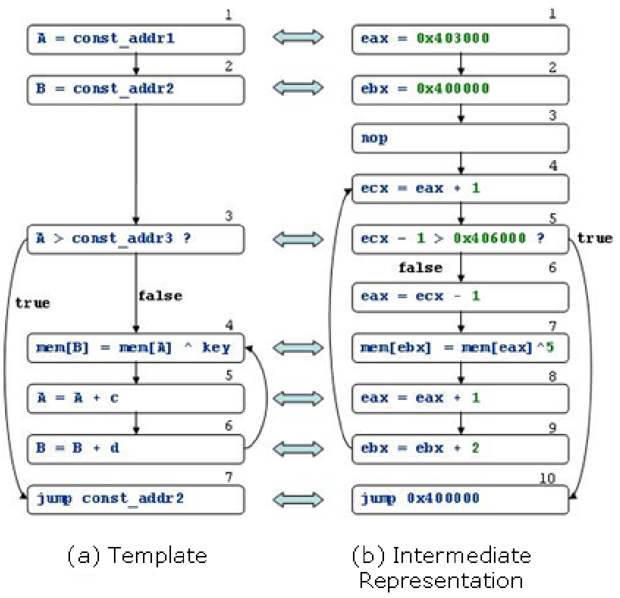

Figure 11: Semantic equivalence. The template (a), quoted from the paper of Christodorescu et al. [42], represents in a generic way a simple ciphering by XOR between two addresses. During the verification, each node from the instance (b) is associated to its potentially equivalent node in the template. Once the correspondence established, the preservation of the variables is checked. In this concrete case, the matter is to ascertain that that the value affected to eax at node 1 is equal to the value of ecx used at node 5. As a result, the instance (b) satisfies the template (a).

4.2.2.2 Algebraic equivalence by reduction

Working from an algebraic approach, the detection is made by deduction using logical equivalence at each reasoning step [47, 48].

In first place the program is translated into a given algebra. This algebra is commonly a formal specification of the processor instruction set which attempts to erase differences between equivalent functionalities. For example a single semantic expression will stand for several equivalent instructions using different registers such as 'mov'.

Once translated, the program abstraction is then simplified by reduction using rewriting rules preserving the 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 in the Figure X showing rewriting rules reversing metamorphic transformations.

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

```
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.
```

Figure 12: Reduction rules reversing metamorphic transformations. These two rewriting rules quoted from M. Webster paper [47] 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*{_}.

4.2.2.3 Model checkers

In model checking, the model used to describe the behaviors is more peculiar. A behavior will be defined by a temporal logic formula [49, 50] which introduces dynamic aspects in the first-order logic. An example completely detailed is given in Figure 12. For more information, it is a recommended to refer to the corresponding literature [51]. 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 must explore enumeratively all the possible execution paths. As a matter of fact, symbolic temporal model checker prove to be PSpace-complete [52].

In the most recent logic, registers, free variables and constants are referenced as generic values for genericity sake [53]. This improvement particularly address mutations by reassignment as shown by the figure. During the verification process, the algorithm will link the generic values with real registers and variables and store this information all along the explored execution path. It is also important to underline the fact that the temporal predicates used to explore the different paths prove to be useful thwarting garbage code insertion and code reordering.

4.2.3 Behavior model generation

Along the two previous sections, parallelly to the different matching algorithms, we have described several behavior models without mentioning the creation process of the behavior signatures. This third part is dedicated to the generation methods for these signatures.

4.2.3.1 Manual definition

Though time consuming, manual definition remain the principal generation method because of its reliability. Two main sources of knowledge are use to feed the process of model creation. In most cases, an expert with significant experience will define generic and opaque behavior models for interoperability sake between the different customer machines. But in certain systems, the responsibility will be passed on to the user. He is then free to define its own policy which will be more adapted to his own system since he can take into account the different installed softwares. On the other hand he must be well taught and be aware of the possible repercussions of his choices.

$$\begin{aligned} C_1 : \exists r EF(&mov(r,0) \\ &\wedge EF(push(r) \\ &\wedge EF(call(func)))) \end{aligned}$$

$$\begin{aligned} C_2 : \exists r EF(&mov(r,0) \\ &\wedge AX(push(r) \\ &\wedge EF(call(func)))) \end{aligned}$$

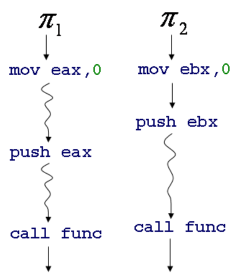

Figure 13: Examples of temporal logic formulae. The operators A and E are path quantifiers whereas X and F are temporal operators. For example, the combination $EF(p)$ means that an execution path exists where an undetermined future state satisfies the predicate p. In the present case, the first formula C_1 means that there is possible path where the value 0 is affected to a generic register r which is then pushed on the stack before a call to the function $func$. Notice that these operations may not be consecutive. Replacing the operators EF by AX in the second condition compels the register affectation to be in every possible path (and no longer in at least one). Moreover, pushing the register value on the stack must be the immediate following action. As an illustration, π_1 and π_2 are two execution paths satisfying respectively C_1 and C_2.

4.2.3.2 Automatic learning: Data mining and classifiers

The automatic generation of behavior 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 classifier is used, the general procedure remain the same. In a first time, the system is 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 sufficiently important and well chosen to exhibit no bias. Like any learning process, the generation of behavior signatures remain very sensitive to noise injection in the training pool. Some effective attacks have already been published against similar worm signature generators [54]. During the training period, the classifier will crawl into this data repository to extract common properties between the different considered classes. The functioning principle is reminded schematically in the Figure 13. In a behavioral context, the extraction of these common properties relies on three major paradigms:

Rules induction: This first paradigm specifies belonging conditions for the different classes. For each sample received by the classifier, it integrates or removes certain characteristic data in the condition in order to preserve the class consistency [55, 56, 57]. This kind of rules is often formulated as Boolean expression as it is pictured in Figure 14 or as decision trees [58].

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 [56, 57, 58]. The Figure 15 describes examples using system calls and strings as collected

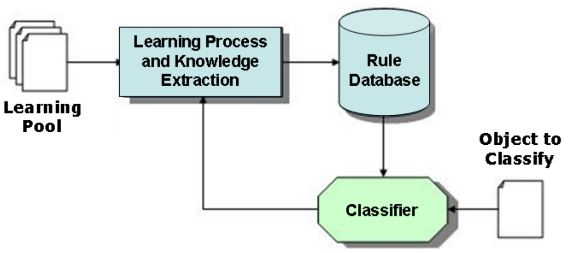

Figure 14: Learning process. The original knowledge is extracted from a learning pool and integrated to the rule database. The rules are then evaluated by the classifier. According to their relevance, the process is iterated until stabilization of the rules set.

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

Figure 15: Mail Worm class example. The following rules determines the chracteristics (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.

data. Ultimately, only the results exhibiting the most important powers of discrimination will be kept. An important criterion in this choice will be the minimal overlapping of the characteristics in the different classes. The ideal case would obviously be when a characteristic is existing with a probability of 100% in a unique class whereas it is absent of any other.

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%

Figure 16: Probabilities of several characteristics. These results are only given as examples. Nevertheless 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 use 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.

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 will measure distances between the profiles and the tested programs [59]. 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 an other but it remains a factor impacting heavily on the classification accuracy. The Figure 16 gives an example where the distance is calculated on the basis of the number of modifications necessary to pass from a call sequence to an other.

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

Figure 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.

5 Panorama of existing behavioral detectors

As an illustration, we have classified several existing behavior-based systems of detection according to the elements of our taxonomy. The result is given in Table 1 completed with additional practical information. They 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 resources or do not exhibit error rates enough low. 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 antiviruses using behavioral detection with host-based intrusion prevention systems (HIPS). It becomes less and less obvious to draw a demarcation line between the two. This is not really surprising since virology and intrusion detection are connected security domains.

6 Conclusion

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 of functionality identification. In particular, we observe in the taxonomy a clear distinction between the static and dynamic 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 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 spywares parasiting web browsers. 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

Name (Origin)	Date	Ref.	Capture	Input	Target	Engine type	Usage	Environment
TBScan (N/C)	1994	[34]	Dyn.(SB)	Interruptions	File infectors	Heuristic algorithm (flags)	Det.	Ms DOS
VIDES (Unv. Namur & Hamburg)	1995	[38]	Dyn.(RT)	Interruptions	COM and EXE Infectors	Deterministic finite automata	D./C.	Ms DOS
N/C (Unv. Columbia & N.Y.)	2003	[56]	Static	Imported functions, strings	All kinds of malware	Data mining and classifier	Det.	Win
GateKeeper (Florida Inst. of Tech.)	2004	[22]	Dyn.(RT*)	System calls	All kinds of malware	Heuristic algorithm (weight)	Det.	Win
N/C (Unv. Camegie et al.)	2005	[42]	Static	Control flow graphs	Polymorphic mail worms	Semantically annotec graph isomorphism	Det.	Win
N/C (Unv. Munich)	2005	[53]	Static	Control flow graphs	Worms	Model checking	Det.	Win
N/C (Unv. Liverpool)	2006	[48]	Static	Algebraic program abstraction	Metamorphic viruses	Equivalence by reduction	Det.	IA32
TTAnalyze (Technical Unv. Vienna)	2006	[25]	Dyn.(VM)	System calls	All kinds of malware	Simple activity log	Class.	Win
N/C (Microsoft Corp.)	2006	[59]	Dyn.(VM)	System calls	All kinds of malware	Data mining and classifier	Class	Win
N/C (Unv. California & Vienna)	2006	[60]	Dyn./Stat.	COM and system calls	Web client spywares	Expert system	Det.	Internet Explorer
ThreatSense - NOD32 (Eset)	N/C	[61]	Dyn.(SB)	Instructions associated to actions	All kinds of malware	Heuristic algorithm	Det.	Win/Linux/ FreeBSD
AVG Anti-Virus (Grisoft)	N/C	[62]	Dyn.(SB)	Instructions associated to actions	All kinds of malware	Heuristic algorithm	Det.	Win/Linux/ FreeBSD
ViGUARD (Softed)	N/C	[63]	Dyn.(RT)	System calls	All kinds of malware	Expert system (user's decision)	Det.	Win
B-HAVE - Bit Defender (Softwin)	N/C	[64]	Dyn.(SB)	Instructions associated to actions	All kinds of malware	Heuristic algorithm	Det.	Win/Linux/ FreeBSD
Bloodhound - Norton (Symantec)	1997	[36]	Dyn.(SB)	Instructions associated to actions	File infectors	Heuristic algorithm	Det.	Win
Entercept (Mc Affee)	2004	[65]	Dyn.(RT)	System calls	All kinds of malware	Expert system (predefined policy)	Det.	Win/Linux
Safe'n'Sec Antivirus (Safen Soft)	2004	[66]	Dyn.(RT)	System calls	All kinds of malware	Expert system (predefined policy)	Det.	Win/Linux/ FreeBSD
TruPrevent (Panda Software)	2006	[67]	Dyn.(RT)	System calls	All kinds of malware	Heuristic algorithm	Det.	Win/Linux
Virus Keeper (AxBa)	2007	[68]	Dyn.(RT)	System calls	All kinds of malware	Expert system (user's decision)	Det.	Win

Table 1: Classification of existing behavioral detectors. Used abbreviations for the capture conditions: RT = Real-Time / SB = SandBox / VM = Virtual Machine / * = Actions recording, for the system usage: Det. = Detection / Class. = Classification.

activity [60]. Generally speaking, a static analysis could be deployed at each reached branching to explore the alternative execution paths that will not be executed.

If we want to combine efficiently both modes it remain necessary to evolve towards a common model of reference. This model could then be slightly adapted according to the class of system considered while remaining compatible with others. Unfortunately, such a model is still missing.

References

[1] F. Cohen, *Computer Viruses.* PhD thesis, University of South California, 1986.

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

[3] H. Debar, M. Dacier, and A. Wespi, "Towards a taxonomy of intrusion-detection systems," *Computers Networks, Special Issue on Computer Network Security*, vol. 31, no. 9, pp. 805–822, 1999.

[4] L. Mé and B. Morin, "Intrusion detection and virology: an analysis of differences, similarities and complementariness," *Journal in Computer Virology*, vol. 3, no. 1, WTCV'06 Special Issue, G. Bonfante and J-Y. Marion Eds., pp. 39–49, 2007.

[5] J. Anderson, "Computer security threat monitoring and surveillance," tech. rep., James P. Anderson Company, 1980.

[6] D. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, vol. SE-13, 1987.

[7] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusion using system calls: Alternative data models," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 133–145, 1999.

[8] S. Zanero, "Behavioral intrusion detection," in *Proceedings of the 19th International Symposium on Computer and Information Sciences (ISCIS)*, pp. 657–666, 2004.

[9] E. Filiol, *Computer viruses: from theory to applications.* Springer, IRIS Collection, 2005, ISBN:2-287-23939-1.

[10] "Fortinet observatory." url=`www.fortinet.com/FortiGuardCenter/`.

[11] "Malware outbreak trend report: Storm-worm," Commtouch Software Ltd, 2007, url=`www.commtouch.com/downloads/Storm-Worm_MOTR.pdf`.

[12] E. Filiol, "Malware pattern scanning schemes secure against black-box analysis," *Journal in Computer Virology*, vol. 2, no. 1, EICAR 2006 Special Issue, V. Broucek Ed., pp. 35–50, 2006.

[13] E. Filiol, *Techniques Virales Avancées.* Springer, IRIS Collection, 2007, ISBN:2-287-33887-8.

[14] P. Ször, *The Art of Computer Virus Research and Defense.* Addison-Wesley, 2005, ISBN:0-321-30454-3.

[15] D. Spinellis, "Reliable identification of boundedlength viruses is np-complete," *IEEE Transactions on Information Theory*, vol. 49, pp. 280–284, 2003.

[16] M. Christodorescu and S. Jha, "Testing malware detectors," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 34–44, ACM Press, 2004.

[17] S. Josse, "How to assess the effectiveness of your anti-virus?," *Journal in Computer Virology*, vol. 2, no. 1, EICAR 2006 Special Issue, V. Broucek Ed., pp. 51–65, 2006.

[18] E. Filiol, G. Jacob, and M. L. Liard, "Evaluation methodology and theoretical model for antiviral behavioural detection strategies," *Journal in Computer Virology*, vol. 3, no. 1, WTCV'06 Special Issue, G. Bonfante and J-Y. Marion Eds., pp. 23–37, 2007.

[19] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," in *Proceedings of the European Symposium on Research in Computer Security*, pp. 326–343, 2003.

[20] G. Hoglund and J. Butler, *Rootkits, Subverting the Windows kernel.* Addison-Wesley Professional, 2006, ISBN:0-321-29431-9.

[21] A. D. Vivanco, "Comprehensive non-intrusive protection with data-restoration: A proactive approach against malicious mobile code," Master's thesis, Florida Institute of Technology, 2002.

[22] M. E. Wagner, "Behavior oriented detection of malicious code at run-time," Master's thesis, Florida Institute of Technology, 2004.

[23] "Norman′s sandbox malware analyzer." Norman ASA, url=`www.norman.com/microsites/malwareanalyzer/fr/`.

[24] "Cwsandbox." Sunbelt Software, url=`www.cwsandbox.org`.

[25] U. Bayer, C. Kruegel, and E. Kirda, "Ttanalyze: A tool for analyzing malware," in *Proceedings of EICAR*, 2006.

[26] J. Rutkowska, "Red pill... or how to detect vmm using (almost) one cpu instruction," 2005, url=`http://invisiblethings.org/papers/redpill.html`.

[27] P. Ferrie, "Attacks on virtual machine emulators," in *Proceedings of the AVAR conference*, 2006.

[28] S. Josse, "Secure and advanced unpacking using computer emulation," in *Proceedings of the AVAR conference*, 2006.

[29] M. D. Preda, M. Christodorescu, S. Jha, and S.Debray, "A semantic-based approach to malware detection," in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[30] M. Debbabi, "Dynamic monitoring of malicious activity in software systems," in *Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS)*, 2001.

[31] C. Nachenberg, "Behavior blocking: The next step in anti-virus protection," SecurityFocus, 2002, url=`www.securityfocus.com/infocus/1557`.

[32] M. Schmall, *Classification and Identification of Malicious Code Based on Heuristic Techniques Utilizing Meta-languages.* PhD thesis, University of Hamburg, 2002.

[33] M. Schmall, "Heuristic techniques in av solutions: An overview," SecurityFocus, 2002, url=`www.securityfocus.com/infocus/1542`.

[34] F. Veldman, "Heuristic anti-virus technology," in *Proceedings of the International Virus Protection and Information Security Council*, 1994.

[35] R. Zwienenberg, "Heuristics scanners: Artificial intelligence?," in *Proceedings of the Virus Bulletin Conference*, pp. 203–210, 1994.

[36] "Understanding heuristics: Symantec bloodhound technology," tech. rep., Symantec White Paper Series / Volume XXXIV, 1997.

[37] F. W. Glover and G. A. Kochenberger, *Handbook of Metaheuristics.* Springer, 2003, ISBN:1-402-07263-5.

[38] B. L. Charlier, A. Mounji, and M. Swimmer, "Dynamic detection and classification of computer viruses using general behaviour patterns," in *Proceedings of the Virus Bulletin Conference*, 1995.

[39] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati, "A fast automaton-based approach for detecting anomalous program behaviors," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 144–155, 2001.

[40] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages and Computation, Second Edition.* Addison Wesley, 1995, ISBN:0-201-44124-1.

[41] G. Mazeroff, V. D. Cerqueira, J. Gregor, and M. G. Thomason, "Probabilistic trees and automata for application behavior modeling," in *Proceedings of the 43rd ACM Southeast Conference*, 2003.

[42] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantic-aware malware detection," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 32–46, 2005.

[43] D. Bruschi, L. Martignoni, and M. Monga, "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.

[44] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[45] F. Periot, "Defeating polymorphism through code optimization," in *Proceedings of the Virus Bulletin Conference*, pp. 142–159, 2003.

[46] D. Bruschi, L. Martignoni, and M. Monga, "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.

[47] M. Webster, "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.

[48] M. Webster and G. Malcolm, "Detection of metamorphic computer viruses using algebraic specification," *Journal in Computer Virology*, vol. 2, no. 3, pp. 149–161, 2006.

[49] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs," in *Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS)*, 2001.

[50] P. Singh and A. Lakhotia, "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.

[51] E. Clark, O. Grumberg, and D. Long, *Model Checking.* MIT Press, 1999, ISBN:0-262-03270-8.

[52] P. Schnoebelen, "The complexity of temporal logic model checking," *Advances in Modal Logic*, vol. 4, pp. 393–436, 2003.

[53] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," *Lecture Notes in Computer Science*, vol. 3548, pp. 174–187, 2005.

[54] R. Perdisci, D. Dagon, P. W. L. Fogla, and M. Sharif, "Misleading worm signature generators using deliberate noise injection," in *Proceedings of IEEE Symposium on Security and Privacy*, 2006.

[55] W. Lee, S. Stolfo, and P. Chan, "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, 1997.

[56] M. G. Schultz, E. Eskin, and E. Zadok, "Data mining methods for detection of new malicious executables," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 38–49, 2001.

[57] J.-H. Wang, P. S. Deng, Y.-S. Fan, L.-J. Jaw, and Y.-C. Liu, "Virus detection using data mining techniques," in *Proceedings of IEEE on Security Technology*, pp. 71–76, 2003.

[58] J. Kolter and M. Maloof, “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, 2004.

[59] T. Lee and J. Mody, “Behavioral classification,” in *Proceedings of EICAR*, 2006.

[60] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, “Behavior-based spyware detection,” in *Proceedings of the 15th USENIX Security Symposium*, 2006.

[61] Frost&Sullivan, “Protection en temps réel contre toutes les menaces,” tech. rep., White Paper Eset.

[62] “Avg anti-virus.” Grisoft, url=`www.grisoft.com/doc/39/lng/fr/tpl/tpl01`.

[63] “Viguard.” Softed, url=`www.viguard.com/detail_163_logiciel_antivirus_viguard-platinium#`.

[64] “Bitdefender antivirus technology,” tech. rep., BitDefender White Paper.

[65] “Host and network intrusion prevention, competitors or partners?,” tech. rep., Mc Affee White Paper, 2004.

[66] “Safe′n′sec antivirus.” Safen Soft, url=`www.safensoft.com/technology/`.

[67] “Truprevent.” Panda Software, url=`www.pandasoftware.com/products/truprevent_tec.htm?sitepanda=particulares`.

[68] “Virus keeper.” AxBa, url=`www.viruskeeper.com/fr/faq.htm`.