Malware Behavioral Detection by Attribute-Automata Using Abstraction from Platform and Language

Grégoire Jacob[1,2,⋆], Hervé Debar[1,⋆], and Eric Filiol[2]

[1] France Télécom R&D, Caen, France
{gregoire.jacob,herve.debar}@orange-ftgroup.com
[2] Operational Virology and Cryptology Lab., ESIEA Laval, France
filiol@esiea.fr

Abstract. Most behavioral detectors of malware remain specific to a given language and platform, mostly executables for Windows. The objective of this paper is to define a generic approach for behavioral detection based on two layers respectively responsible for abstraction and detection. The abstraction layer is specific to a platform and a language. It interprets the collected instructions, API calls and arguments and classifies these operations, as well as the objects involved, according to their purpose in the malware lifecycle. The detection layer remains generic and interoperable with different abstraction components. It relies on parallel automata parsing attribute-grammars where semantic rules are used for object typing (object classification) and object binding (data-flow). Theoretical results are first given with respect to the grammatical constraints weighting on the signature construction as well as to the resulting complexity of the detection. For experimentation purposes, two abstraction components have then been developed: one processing system call traces and the other processing the VBScript interpreted language. Experimentations have provided promising detection rates, in particular for scripts (89%), with almost no false positives. In the case of process traces, the detection rate remains significant (51%) but could be increased by sophisticated collection tools.

Keywords: Malware, Behaviors, Attribute-Grammars, Interpretation.

1 Introduction

Malware behavioral detection should theoretically be able to detect, if not innovative malware, at least unknown malware reusing variations of known techniques. However, most current behavioral detectors rely on specific characteristics, allowing evasion through simple functional modifications. This article aims to provide generic grammars modeling malicious behaviors in order

⋆ This work has been partially supported by the European Commissions through project FP7-ICT-216026-WOMBAT funded by the 7th framework program. The opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the European Commission.

to build efficient and resilient detection automata. Deterministic finite automata are attractive because their linear complexity remains acceptable for operational deployment. In 1995, [1] already used automata to describe the alternative sequences of operations making up malicious behaviors. Since then, researches focusing on the notion of data flow has led to the apparition of tainting techniques to detect malicious uses of data [2]. Control of the data flow has exhibited significant successes and is now broadly used, in intrusion detection [3] or malware behavior extraction [4]. These articles use automata to model the sequences of system calls constituting respectively attacks and behaviors. The data flow is then captured by analysis of the parameters collected along the calls. On this principle, [5] focuses on self-reproduction as the discriminating behavior for detection.

Similarly, our approach of behavioral detection combines automata and data flow control. The model easily supports multiple behaviors. In fact, malicious behaviors are described by attribute-grammars. Syntactic rules describe the possible combinations of operations making up the behavior, whereas, semantic rules both control the data flow between the involved objects, and associate them with a potential purpose in the malware lifecycle (installation, communication, execution). The detection process is finally achieved by parsing execution traces to check for the satisfaction of the grammatical behavior descriptions.

Abstraction is needed to translate observed traces into the behavioral model for detection. By a layered architecture, [6] addresses the semantic gap existing between the system call traces, understandable by OS specialists, and high-level behaviors. Similarly, our abstraction layer provides generic descriptions where the processed data get detached from the specificities of the platform and the programming language. In fact, the graph-based formalism in [6] is in many ways equivalent to the grammatical formalism provided here. In effect, AND/OR graphs may be expressed by the semantic rules of attribute-grammars. Relying on a well-established formalism, these grammars provide theoretical results in terms of complexity which also hold for the approach from [6]. In addition, the present article provides different behaviors, assessed on larger test pools.

With regards to the operations for language abstraction, the identification of the system objects with a potential use for malware and the generation of the grammatical behavior descriptions, they all require an initial configuration step as described in Fig.1. Contrary to other methods, the configuration focuses both on critical objects, which remain enumerable in a standard environment, and innovative malware, which are scarce among the numerous variants of known malware. In a few words, this paper introduces the following contributions:

- A model of malicious behaviors using attribute-grammars with semantic for object binding (data flow control) and typing (object purposes for malware).
- An abstraction layer to translate observed traces into the model, detaching detection from the specificities of platforms and programming languages, with two proofs of concept to analyze executable traces and scripts.
- Some generic automata for behavior detection with an assessment from perspectives theoretical (complexity) and operational (coverage, performance).

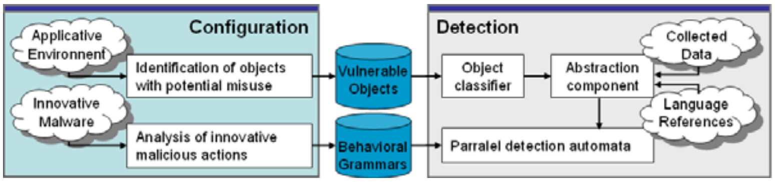


Fig. 1. Configuration and detection processes

The article is articulated as follows. Section 2 introduces the behavioral model based on attribute-grammars. Section 3 presents the abstraction process from the collected data to the model. Section 4 describes the detection process. An implementation is given in Section 5 whose results are commented in Section 6.

2 Grammatical Formalization of Behaviors

From a theoretical perspective, an attribute-grammar (Definition 1) is a Context-Free Grammar (CFG) enriched with semantic attributes and rules [7]. In the formalism, each start symbols begins the description of a new malicious behavior. The terminal symbols of the grammar then correspond to the basic operations making up the behavior whereas the production rules describe their different combinations to achieve the behavior. As stated in [8,9], basic operations eventually refer to data collected through the abstraction layer (instructions, API calls, parameters). These common principles are kept along the formalization.

Definition 1. *An attribute-grammar* G_A *is a triplet* $<G, D, E>$ *where:*

- G is originally a context-free grammar $<V, \Sigma, S, P>$,
- $att : X \in \{V \cup \Sigma\} \rightarrow att(X) \in Att^*$ *is an assignment function for attributes and* $D = \cup_{\alpha \in Att} D_\alpha$ *their set of values,*
- E is a set of semantic rules such as for any production of P, there is at most one rule per variable of the form $Y.\alpha = f(Y_1.\alpha_1 ... Y_n.\alpha_n)$ *with* $f : D_{\alpha_1} \times ... \times D_{\alpha_n} \rightarrow D_\alpha$.

2.1 Malicious Behavior Language

A generic programming language is required to describe malicious behaviors: the Malicious Behavior Language (MBL) has been designed to this purpose. Its syntax and operational semantics are given in [8]. Most malicious behaviors can be described by sub-grammars of the MBL generative grammar. The language principles are object-oriented according to the encapsulation in Fig.2. It provides internal operations: arithmetic and control operations guaranteeing Turing completeness, as well as interactions to interface with external objects: commands (open, create, close, delete, execute) or inputs/outputs (send, receive).

On top of the syntax for operations and interactions, a type system has been provided for the external objects. These objects are typed according to

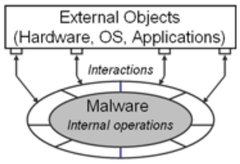


Fig. 2. Malware object encapsulation

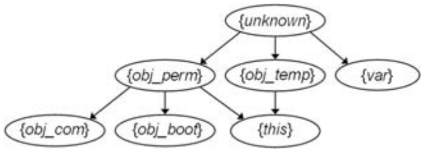


Fig. 3. Object type poset

their potential use in the malware lifecycle: permanent objects (*obj_perm*), temporary objects (*obj_temp*), booting objects (*obj_boot*), communicating objects (*obj_com*), self-reference (*this*). A partial order has been defined on these types according to their subset inclusion, as shown in the Hasse Diagram of Fig.3. These inclusions correspond to object specializations. This type system can be deployed thanks to the semantic attributes enriching attribute-grammars. In fact, semantic attributes and rules can have several purposes:

Object binding: Object binding identifies the different object instances, and guarantees they are coherently used. It is achieved by affecting specific attributes called identifiers to the terminal symbols representing objects (denoted $objId$ where Id is an abbreviation for Identifier). Considering interactions, the binding constrains the data-flow between objects. The data flow is critical in behaviors such as duplication where data transfers are involved.

Object typing: A type attribute can also be affected to a given object (denoted $objTp$ where Tp is an abbreviation for Type). Types are attached to objects according to their potential use. They are critical to distinguish certain malicious purposes such as booting objects in the case of residency or communicating objects in the case of propagation. Additional characterization of the objects can be achieved through additional attributes. For example, an attribute can store the object nature (denoted $objNat$): variable, file, registry key, network socket, mail, etc.

2.2 Descriptions of Malicious Behaviors

Four behaviors are examined: duplication, propagation, residency (automatic start) and overinfection tests (avoiding reinfection of an infected system). Because their whole descriptions would be too tedious, only two extracts of the most prevalent ones are covered: duplication and propagation. Their descriptions, as well as additional behaviors, had been generated in [8], by manual analysis of a malware pool. Since these descriptions convey the most generic features of the malicious behaviors, manual generation can be considered more easily than for the binary signatures of scanners.

Duplication. Duplication is achieved by copying code from the self-reference to a permanent object. It is described below by syntactic production rules (grey) and their related semantic rules (white). The syntactic derivations correspond to different duplication techniques: only single-block read/write is described above, but the complete description also supports interleaved read/write and direct copy. The semantic rules guarantee the data-flow through a same variable between read and write interactions (Binding: <*Write*>.varId = <*Read*>.varId). They also guarantee the behavior maliciousness by constraining read interactions to refer to the self-reference (Typing: <*Duplicate*>.srcTp = *this*).

(*i*) <*Duplicate*>	::=	<*Create*><*Open*> <*Read*><*Write*>
	|	<*Open*><*Create*> <*Read*><*Write*>
	|	<*Open*><*Read*> <*Create*><*Write*>
{ <*Duplicate*>.srcId	=	<*Open*>.objId
<*Duplicate*>.srcTp	=	*this*
<*Duplicate*>.targId	=	<*Create*>.objId
<*Duplicate*>.targTp	=	*obj_perm*
<*Open*>.objTp	=	<*Duplicate*>.srcTp
<*Create*>.objTp	=	<*Duplicate*>.targTp
<*Read*>.objId	=	<*Duplicate*>.srcId
<*Read*>.objTp	=	<*Duplicate*>.srcTp
<*Write*>.objId	=	<*Duplicate*>.targId
<*Write*>.objTp	=	<*Duplicate*>.targTp
<*Write*>.varId	=	<*Read*>.varId }

(*ii*) <*Create*>	::=	*create object*;
{ <*Create*>.objId	=	*object*.objId
object.objTp	=	<*Create*>.objTp }
(*iii*) <*Open*>	::=	*open object*;
{ <*Open*>.objId	=	*object*.objId
object.objTp	=	<*Open*>.objTp }
(*iv*) <*Read*>	::=	*receive object*1 $\leftarrow$ *object*2;
{ <*Read*>.varId	=	*object*1.objId
*object*1.objTp	=	*var*
*object*2.objId	=	<*Read*>.objId
*object*2.objTp	=	<*Read*>.objTp }
(*v*) <*Write*>	::=	*send object*1 $\rightarrow$ *object*2;
{ <*Write*>.varId	=	*object*1.objId
*object*1.objTp	=	*var*
*object*2.objId	=	<*Write*>.objId
*object*2.objTp	=	<*Write*>.objTp }

Propagation. Propagation differs from duplication by a different target. The malware code is copied from the self-reference to a communicating object. Consequently, it shows syntactic similarities with duplication, except adjustments to insert a format process. The main differences thus lie in adaptations of the semantic rules. Illustrating typing, the permanent type of the target is replaced by the communicating type (<*Propagate*>.targTp = *obj_com*). A communicating object can either be a network connection, a mail or a shared file. The second modification specifies, by a disjunction of semantic equations, that the propagation source can be either the self-reference or the result of a previous duplication (<*Propagate*>.srcTp = *this* or <*Propagate*>.srcId = <*Duplicate*>.targId).

(*i*) <*Propagate*>	::=	<*Open*><*Read*><*Transmit*>
	|	<*Read*><*Open*><*Transmit*>
{ <*Propagate*>.srcId	=	<*Read*>.objId
(<*Propagate*>.srcTp = *this*	$\vee$	<*Propagate*>.srcId = <*Duplication*>.targId)
<*Propagate*>.targId	=	<*Open*>.objId
<*Propagate*>.targTp	=	*obj_com*
...		}
(*ii*) <*Transmit*>	::=	<*Format*><*Write*> | <*Write*>

3 Model Translation by Abstraction

In the context of behavioral detection, a trace conveying the actions of the monitored program is statically or dynamically collected. Depending on the collection mechanism, completeness of the data and its nature vary greatly, from simple instructions to system calls along with their parameters. The trace remains specific to a given platform and to the language in which the program has been coded (native, interpreted, macros). An abstraction layer is thus required for translation into the behavioral language from Section 2. Translation of basic instructions, either arithmetic (move, addition, subtraction...) or control related (conditional, jump...), into operations of the language is an obvious mapping, not requiring further explanation. On the opposite, translation of API calls and their parameters into interactions and objects from the language is detailed thereafter.

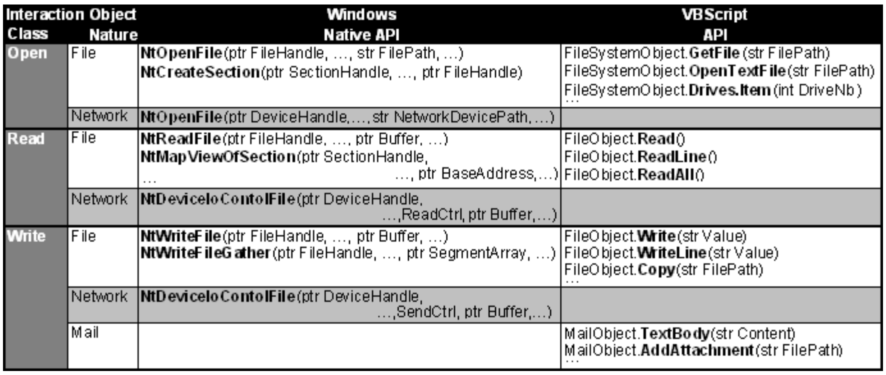
3.1 API Calls Translation

For a program to access services and resources, the Application Programming Interfaces (APIs) constitute a mandatory point enforcing security and consistency [10]. API calls are also denoted system calls when accessing services from the operating system. For each programming language, the set of available APIs can be classified into distinct interaction classes. This set being finite and supposedly stable, the translation is defined as a mapping over the interaction classes, the completeness of the process being guaranteed. Table 1 provides a mapping for APIs subsets from Windows [11] and VBScript. The table is refined according to the nature of the manipulated objects. The API name, on its own, is not always sufficient to determine its interaction class. For example, network devices and files use common APIs; the distinction is made on their path (`\device\Afd\Endpoint`). Sending, receiving packets then depends on control codes transmitted to `NtDeviceIoControlFile` (`IOCTL_AFD_RECV`, `IOCTL_AFD_SEND`). If required, specific call parameters constitute additional mapping inputs:
$\{\text{API name}\} \times (\{\text{Parameters}\} \cup \{\epsilon\}) \rightarrow \{\text{Interaction class}\}$.

3.2 Parameters Interpretation

In the context of interactions, parameters are important factors to identify the involved objects and assess their criticality through typing. Parameters interpretation thus complements the initial abstraction from the platform and language obtained through API translation. Due to their varying nature, parameters can

Table 1. Mapping Windows Native and VBScript APIs to interaction classes



Interaction Class	Object Nature	Windows Native API	VBScript API
Open	File	**NtOpenFile**(ptr FileHandle, ..., str FilePath, ...) **NtCreateSection**(ptr SectionHandle, ..., ptr FileHandle)	FileSystemObject.**GetFile** (str FilePath) FileSystemObject.**OpenTextFile**(str FilePath) FileSystemObject.**Drives.Item** (int DriveNb) ...
	Network	**NtOpenFile**(ptr DeviceHandle,...,str NetworkDevicePath,...)	
Read	File	**NtReadFile**(ptr FileHandle, ..., ptr Buffer, ...) **NtMapViewOfSection**(ptr SectionHandle, ..., ptr BaseAddress,...) ...	FileObject.**Read**() FileObject.**ReadLine**() FileObject.**ReadAll**()
	Network	**NtDeviceIoContolFile**(ptr DeviceHandle, ...,ReadCtrl, ptr Buffer,...)	
Write	File	**NtWriteFile**(ptr FileHandle, ..., ptr Buffer, ...) **NtWriteFileGather**(ptr FileHandle, ..., ptr SegmentArray, ...)	FileObject.**Write**(str Value) FileObject.**WriteLine**(str Value) FileObject.**Copy**(str FilePath) ...
	Network	**NtDeviceIoContolFile**(ptr DeviceHandle, ...,SendCtrl, ptr Buffer,...)	
	Mail		MailObject.**TextBody**(str Content) MailObject.**AddAttachment**(str FilePath) ...

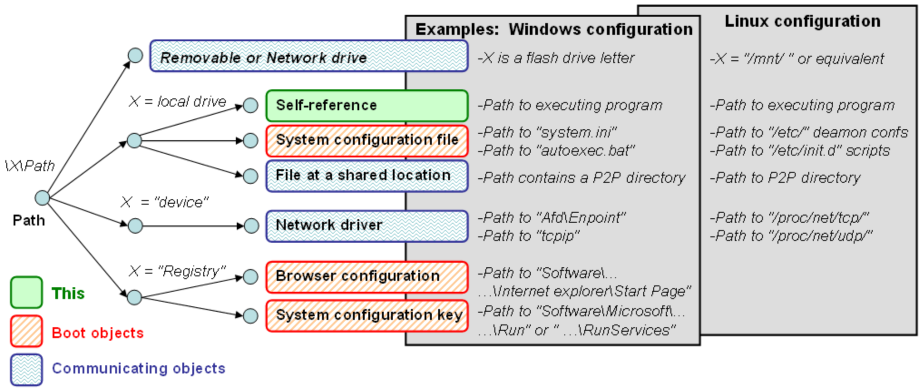

Examples: Windows configuration
Linux configuration
\X\Path
Path
Removable or Network drive
-X is a flash drive letter
-X = "/mnt/ " or equivalent
X = local drive
Self-reference
-Path to executing program
-Path to executing program
System configuration file
-Path to "system.ini"
-Path to "autoexec.bat"
-Path to "/etc/" deamon confs
-Path to "/etc/init.d" scripts
File at a shared location
-Path contains a P2P directory
-Path to P2P directory
X = "device"
Network driver
-Path to "Afd\Enpoint"
-Path to "tcpip"
-Path to "/proc/net/tcp/"
-Path to "/proc/net/udp/"
X = "Registry"
Browser configuration
-Path to "Software\...
...\Internet explorer\Start Page"
System configuration key
-Path to "Software\Microsoft\...
...\Run" or " ...\RunServices"
This
Boot objects
Communicating objects

Fig. 4. Character strings interpretation

not be translated by a simple mapping. Decision trees are more adaptive tools, capable of interpreting parameters according to their representation:

Simple integers: Integer attributes are mainly constants specific to an associated API. They may condition the interpretation of its interaction class.

Address and Handles: Addresses and handles identify the different objects appearing in the collected traces. They are particularly useful to study the data flow between objects. Considering a variable, it is represented by its address a_v and its size s_v. Every address a such as $a_v \leq a \leq a_v + s_v$ will refer to the same variable. Certain addresses with important properties may be refined by typing: import tables, services table, entry points. These specific addresses may be interpreted by decision trees partitioning the address space.

Character strings: String parameters contain the richer information. Most of these parameters are paths satisfying a hierarchical structure where every element is important: from the root identifying drives, drivers and registry, passing by the intermediate directories providing object localization, until the real name of the object. This hierarchical structure is well adapted for a progressive analysis embedded in a decision tree. A progressive interpretation of the path elements is shown in Fig.4 with basic examples for Windows and Linux platforms.

3.3 Decision Trees Generation

Building decision trees requires a precise identification of the critical resources of a system. Our methodology proceeds by successive layers: hardware, operating system and applications. For each layer, we define a scope encompassing the significant components; the resources involved either in the installation, the configuration or the use of these components are monitored for potential misuse:

Hardware layer: For the hardware layer, the scope can be restricted to the interfaces open to external locations (Network, CD, USB). The key resources to monitor are the drivers used to communicate with these interfaces as well as additional configuration files (e.g. `Autorun.inf` files impacting booting).

Operating system layer: OS configuration is critical but unfortunately dispersed in various locations (e.g. files, registry, structures in memory). However, most of the critical resources are already well identified, such as the boot sequence or the intermediate structures used to access the provided services and resources (e.g. file system, process table, system call table).

Applicative layer: It is obviously impossible to consider all existing applications. To restrict the scope, observing malware propagation and interoperability constraints, the analysis is limited to connected and widely deployed applications (web browsers, messaging, mail, peer-to-peer, IRC clients). Again are considered resources involved in communication (connections, transit locations) as well as in configuration (application launch).

Identification of the critical resources potentially used by malware is a manual, but necessary, configuration step. We believe however that it is less cumbersome than analyzing the thousands of malware discovered every day, for the following reasons. First, critical resources of a given platform are known and limited; they can thus be enumerated. Their name and location can then be retrieved in a partially automated way (e.g. listing connected drives, recovering peer-to-peer clients and their shared folders). In fact, full automation of the parameter interpretation may be hard to achieve. In [12], an attempt was made to fully automate their analysis for anomaly-based intrusion detection. The interpretation relied on deviations from a legitimate model based on string length, character distribution and structural inference. These factors are significant for intrusions which mostly use misformatted parameters to infiltrate through vulnerabilities. It may prove less efficient with malware since they can use legitimate parameters, at least in appearance. Moreover, the real purpose of these parameters would still be unexplained; an additional analysis would be required for type affectation. Thus, interpretation by decision trees with automated configuration seems a good trade off between automation and beforehand manual analysis.

4 Detection Using Parsing Automata

Detecting malicious behaviors may be reduced to parsing their grammatical descriptions. To achieve syntactic parsing and attribute evaluation in a single pass, the attribute-grammars must be both LL grammars and L-attribute grammars: attribute dependency is only allowed from left to right in the production rules. These properties are not necessarily satisfied by the MBL generative grammar but they prove true for the sub-grammars describing the malicious behaviors. Therefore, detection can be implemented by LL-parsers, capable of building, from top to down, the annotated leftmost-derivation trees. Basically, LL-parsers are pushdown automata enhanced with attribute evaluation (Definition 2).

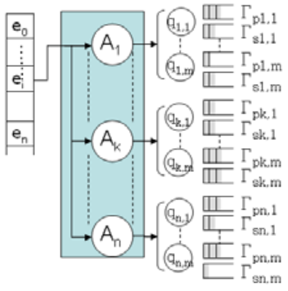
Definition 2. *A LL-parser is a particular pushdown automaton A that can be built as a ten-tuple* $<Q, \Sigma, D, \Gamma_p, \Gamma_s, \delta, q_0, Z_{p,0}, Z_{s,0}, F>$ *where:*

- Q is the finite set of states, and $F \subset Q$ is the subset of accepting states,
- Σ is the alphabet of input symbols and D is the set of values for attributes,
- Γ_p / Γ_s are the parsing / semantic stack alphabets,
- $q_0 \in Q$ is the initial state and $Z_{p,0}$ / $Z_{s,0}$ are the stacks start symbols,
- δ is the transition function defining the production rules and semantic routines, of the form: $Q \times (\{\Sigma \cup \epsilon\}, D^*) \times (\Gamma_p, \Gamma_s) \rightarrow Q \times (\{\Gamma_p \cup \epsilon\}, \Gamma_s)$.

Several behaviors are monitored in parallel by dedicated automata. Each automaton A_k parses several instances of the behavior, storing its progress in independent derivations (triple made up of a state q_k and parsing and semantic stacks Γ_{pk}, Γ_{sk}). For each collected events e_i containing input symbols and semantic values, all the parsing parallel automata progress along their derivations. When an irrelevant input is read (an operation interleaved inside the behavior for example), this input is dropped instead of causing an error state. The global procedure is defined in the Algorithms 1 and 2 with an explicative figure.

Algorithm 1. A.ll-parse(e,Q,Γ_p,Γ_s)

```
 1: if $e$, $Q$, $\Gamma_p$, $\Gamma_s$ match a transition $T \in \delta_A$ then
 2:    if $e$ introduces a possible ambiguity then
 3:       duplicate state and stack triple $(Q, \Gamma_p, \Gamma_s)$.
 4:    end if
 5:    Compute transition $T$ to update $(Q, \Gamma_p, \Gamma_s)$.
 6:    if $Q$ is an accepting state $Q \in F_A$ then
 7:       Malicious behavior detected.
 8:    else
 9:       ignore $e$.
10:    end if
11: end if
```



Algorithm 2. BehaviorDetection(e_1,...,e_t)

```
Require: events $e_i$ are couples of symbol and semantic values: $(\{\Sigma \cup \epsilon\}, D^*)$.
 1: for all collected events $e_i$ do
 2:    for all the automata $A_k$ such as $1 \leq k \leq n$ do
 3:       $m_k$ = current number of parallel derivations handled by $A_k$.
 4:       for all state and stack triple $(Q_{k,j}, \Gamma_{pk,j}, \Gamma_{sk,j})$ such as $1 \leq j \leq m_k$ do
 5:          $A_k$.ll-parse$(e_i, Q_{k,j}, \Gamma_{pk,j}, \Gamma_{sk,j})$).
 6:       end for
 7:    end for
 8: end for
```

4.1 Semantic Prerequisites and Consequences

The present detection method can be related to scenario recognition in intrusion detection. An intrusion scenario is defined as a sequence of dependent attacks [13]. For each attack to occur, a set of prerequisites or preconditions must be satisfied. Once completed, new consequences are introduced, also called postconditions. In [14], isolated alerts are correlated into scenarios by parsing attribute-grammars annoted with semantic rules to guarantee the flow between related alerts. Similarly, a malicious behavior is a sequence where each operation prepares for the next one. In a formalization by attribute grammars, the sequence order is led by the syntax whereas prerequisites and consequences are led by semantic rules of the form $Y_i.\alpha = f(Y_1.\alpha_1 ... Y_n.\alpha_n)$ (Definition 1).

Checking prerequisites: Prerequisites are defined by specific semantic rules where the left-side attributes of the equations are attached to terminal symbols ($Y_i \in \Sigma$). During parsing, semantic values are collected along input symbols. These values are compared to values computed using inherited and already synthesized attributes. This comparison corresponds to the matching step performed on the semantic stack Γ_s during transitions from δ.

Evaluating consequences: When the left-side attribute is attached to a non-terminal ($Y_i \in V$) and right-side attributes are valued, the attribute is evaluated. During transitions from δ, the evaluation corresponds to the reduction step where the computed value is pushed on the semantic stack Γ_s.

4.2 Ambiguity Support

All events are fed to the behavior automata. However, some of them may be unrelated to the behavior or unuseful to its completion. Unrelated events do not match any transition and are simply dropped. This is insufficient for unuseful events raising ambiguities: they may be related to the behavior but parsing them makes the derivation fail unpredictably. Let us take an explicit example for duplication. After opening the self-reference, two files are consecutively created. If duplication is achieved between the self-reference and the first file, parsing succeeds. If duplication is achieved with the second one, parsing fails because the automaton has progressed beyond the state of accepting a second creation. Similar ambiguities may be observed along the variable affectations which alter the data-flow. The algorithm should thus be able to manage the different objects and variables combinations. Ambiguities are handled by the detection algorithm using derivation duplicates. Before transition reduction, if the operation is potentially ambiguous, the current derivation is copied in a new triple containing the current state and the parsing and semantic stacks. This solution handles the combinations of events without backtracking. To come back and forth in the derivation trees would have proved too cumbersome for real-time detection.

4.3 Time and Space Complexity

LL-parsing is linear in function of the number of symbols. Parallelism and ambiguities increase the complexity of the detection algorithm. Let us consider calls to the parsing procedure as the reference operation. This procedure is decomposed in three steps: matching, reduction and accept (two comparisons and a computation). In the worst case scenario, all events are related to the behavior automata and all these events introduce ambiguities. In the best case scenario, no ambiguity is raised. Resulting complexities are given in Proposition 1.

Proposition 1. *In the worst case, behavioral detection using attributed automata has a time complexity in $\vartheta(k(2^n - 1))$ and a space complexity in $\vartheta(k2^n(2s))$ where k is the number of automata, n is the number of input symbol and s is the maximum stack size. In the best case, time complexity drops to linear time $\vartheta(kn)$ and space complexity becomes independent from the number of inputs $\vartheta(k2s)$.*

The worst case complexity is important but it quickly drops as the number of ambiguous events decreases. The experimentations in Section 6 show that the ratios of ambiguous events are limited and the algorithm offers satisfactory performances. Based on these ratios, a new assessment of the average practical complexity is provided. Besides, these experimentations also show that important ratios of ambiguous events are already a sign of malicious activity.

Proof. In a best case scenario, the number of derivation for each automaton remains constant. Considering the worst case scenario, all events are potentially ambiguous for all the current derivations. Technically, ambiguities multiply by two the number of derivations at each iteration of the main loop. Consequently, each automaton handles 2^{i-1} different derivations at the i^{th} iteration. The time complexity is then equivalent to the number of calls to the parsing procedure:

(1) $k + 2k + ... + 2^{n-1}k = k(1 + 2 + ... + 2^{n-1}) = k(2^n - 1)$

The maximum number of derivations is reached after the last iteration where all automata manage 2^n parallel derivations. Each derivation is stored in two stacks of size s. This moment coincide with the maximum memory occupation:

(2) $k2^n(2s)$.

5 Prototype Implementation

The prototype includes the aforementioned two layers: a specific collection and abstraction layer and a generic detection layer. The overall architecture is described in Fig.5. Components of the abstraction layer interpret the specificities of the languages whereas the common object classifier interprets the specificities of the platform. As a proof of concept, abstraction components have been implemented for two languages: native code of PE executables and interpreted Visual Basic Script. Above abstraction, the detection layer based on parallel behavioral automata parses the interpreted traces independently from their original source.

5.1 Analyzer of Process Traces

Process traces provide useful information about the system activity of an executable. The detection method could be deployed in real-time but for a greater easiness, the experimentations were led off-line. The process traces were thus collected beforehand inside a virtual environment to avoid any risk of infection. The

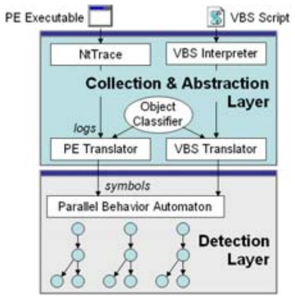


Fig. 5. Multi-layer architecture

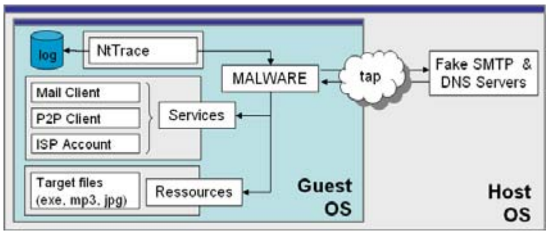


Fig. 6. Collection environment for API calls

prototype deploys an existing tool called NtTrace to collect Windows native calls, their arguments and returned values [15]. The main point with dynamic collection mechanisms (real-time or emulation based) is that most behaviors are conditioned by external objects and events: available target for infection or listening servers for network propagation. In order to increase the mechanism coverage and collect conditioned behaviors, the virtual environment from Fig.6 has been deployed over Qemu [16]. Windows XP was installed on the drive image and useful services and resources were configured: system time, ISP account, Mail and P2P clients, potential targets (.exe, .jpg, .html). Outside the virtual machine, emulations of DNS and SMTP servers have been deployed to establish connections and capture a network activity at the system call level. The weight of the platform and its configuration may seem important but notice that simple call interception would be sufficient in a real-time deployment without any need for a virtual environment.

Translation is then deployed line by line on the collected traces. It directly implements the results from Section 3 for API call translation and parameter interpretation. Only a selection of APIs is classified by mappings; the others are ignored until their future integration. An object classifier, embedding decision trees specifically crafted for a Windows configuration as in Fig.4, is then called on the parameters. During the process, sequences of identical or combined calls are detected and formatted into loops in order to compress the resulting logs. Looking specifically at creation and opening interactions, when resolved, a correspondence is established between the names of objects and their references (addresses, handles). Following interactions check for these references for interpretation. Conversely, on deletion or closing, this correspondence is destroyed for the remainder of the analysis. Names and identifiers must be unlinked since a same address or handle number could be reused for a different object.

5.2 Analyzer of Visual Basic Scripts

No collection tool similar to NtTrace was available for VBScript. We have thus developed our own collection tool, directly embedding the abstraction layer. VBScript being an interpreted language, its static analysis is simpler than native code because of the visibility of the source but also because of some integrated safety properties: no direct code rewriting during execution and no arbitrary transfer of the control flow [17]. For these reasons, path exploration becomes conceivable. The interest of the static approach with respect to the dynamic one used for process traces lies in the coverage of the collected data. In effect, the different potential actions corresponding to the different execution paths will be monitored. In addition, the visibility over the internal data flow will be increased likewise. By comparison, the results of the experimentations will eventually be a good indicator of the impact of the collection mechanism on detection.

Basically, the VBScript analyzer is a partial interpreter using static analysis for path exploration. The analyzer is divided into three parts:

1) Static analyzer: The static analyzer heavily depends on the syntactic specifications of the VBScript language [18]. The script is first parsed to localize the main, the local functions and procedures, as well as to retrieve their signature. Its structure is then parsed by blocks to recover information about the declared variables and instantiated managers (file system, shell, network, mail). In addition, the analyzer also deploys code normalization to remove the syntactic shortcuts provided by VBScript, but most critically to thwart obfuscation. By normalization, the current version can handle certain categories of obfuscation such as integer encoding, string splitting or string encryption.

2) Dynamic interpreter: A partial interpreter has been defined to explore the different execution paths. It is only partial in the sense that the script code is not really executed. Only significant operations and dependencies are collected. To support path exploration, the analyzer handles conditional and loop structures, but also calls to local functions and procedures. Inside these different blocks, each line is processed to retrieve the monitored API calls manipulating files, registry keys, network connections or mails. Calls interpretation is deployed by mapping as previously defined. Affectations, impacting the data-flow, are thereby also monitored. Additional analysis is then deployed to process the expressions used as call arguments, or affected values. In order to control the data-flow, object references and aliases must be followed up through the processing of expressions:
- Local function/procedure calls: linking signature with the passed parameters,
- Monitored API calls: creating objects or updating their type and references,
- Variable affectations: linking variables with values,
- Calls to execute: evaluating expressions as code.

3) Object classifier: The previous classifier has been reused, as in Fig.5. Scripts being based on strings, the address classifier part is unused. The string classifier has been extended to best fit the script particularities, with new constants for the self-reference for example (`"Wscript.ScriptName"`,`"ScriptFullName"`).

5.3 Detection Automata

The transitions corresponding to the different grammar production rules have directly been coded in a prototype similarly to the algorithms from Section 4. Only two enhancements have been brought to the algorithm in order to increase the performance. A first mechanism avoids duplicate derivations. Coexisting identical derivations artificially increase the number of iterations without identifying other behaviors than the ones already detected. The second enhancement is related to the close and delete interactions. Once again, in order to decrease the number of iterations, the derivations where no interaction intervene between the opening/creation and the closing/deletion of an object, are destroyed. These two mechanisms have proved helpful in regulating the number of parallel derivations.

6 Experimentation and Discussions

For experimentation, hundreds of samples have been gathered, the pool being divided into two categories: Portable Executables and Visual Basic Scripts. For each category, about 50 legitimate samples and 200 malware were considered. According to the repartition in Fig.7, different types of legitimate applications, selected from an healthy system, and malware, downloaded from repositories [19,20], have been considered.

1) Coverage: The experimentation has provided significant detection rates with 51% for PE executables and up to 89% for VB Scripts. Results, behavior by behavior, are described in Tables 2 and 3. Duplication is the most significant malicious behavior. However the additional behaviors, and in particular residency, have detected additional malware where duplication was missed. False positives are almost inexistent according to Tables 4 and 5. The only false positive, related to residency, can be easily explained: the script was a malware cleaner reinitializing the browser start page to clear the infection. On the opposite, important false negative spikes can be localized in the PE results (Table 2): the low detection rates for duplication of Viruses and propagation of Net/Mail Worms are explained by limitations in the collection mechanisms that are assessed in 2).

Comparing VB scripts and PE traces, the false negatives are fewer for VB scripts. Path exploration and affectation monitoring implemented in the analyzer provide a greater coverage. The remaining false negatives are explained by the encryption of the whole malware body which is not supported yet and the cohabitation in a same script of JavaScript and VBScript which makes the syntactic analysis fail. Code localization mechanism could solve the problem. For the analyzer of process traces, the detection rates observed for duplication are consistent with existing works [5]. The real enhancements are twofolds: the

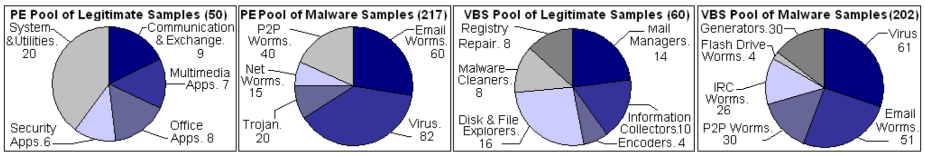


Fig. 7. Repartition of the test pool

Table 2. PE Malware detection. (EmW = Email Worms, P2PW = P2P Worms, V = Virii, NtW = Net Worms, Trj = Trojans, Eng = Functional Polymorphic Engine)

Behaviors	EmW	P2PW	V	NtW	Trj	Global	Eng
Duplication	41(68,33%)	31(77,5%)	15(18,29%)	8(53,33%)	6(30%)	46,54%	30(100%)
direct copy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	8(26,67%)
single read/write	41(68,33%)	30(75%)	14(17,07%)	8(53,33%)	6(30%)	45,63%	12(40%)
interleaved r/w	9(15%)	3(7,5%)	3(3,66%)	3(0,2%)	0(0%)	8,29%	10(33,3%)
Propagation	4(6,67%)	19(47,5%)	3(3,66%)	1(6,67%)	0(0%)	12,44%	17(56,7%)
direct copy	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	0(0%)
single read/write	4(6,67%)	19(47,5%)	3(3,66%)	1(6,67%)	0(0%)	12,44%	17(56,7%)
interleaved r/w	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	0(0%)
Residency	36(60%)	22(55%)	5(60,98%)	6(40%)	9(45%)	35,94%	30(100%)
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	0(0%)
conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	0(0%)
inverse conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%	0(0%)
Global detection	43(71,67%)	33(82,50%)	16(19,51%)	8(53,33%)	11(55,00%)	51,15%	30(100%)

Table 3. VBS Malware detection. (EmW = Email Worms, FdW = Flash Drive Worms, IrcW = IRC Worms, P2PW = P2P Worms, V = Viruses, Gen = Generators variants)

Behaviors	EmW	FdW	IrcW	P2PW	V	Gen	Global
Nb string ciphered	1/51	0/4	1/26	0/30	3/61	10/30	15/202
Nb body ciphered	4/51	0/4	0/26	1/30	2/61	0/30	7/202
String encryption	1(100%)	0	0	0(0%)	2(66,67%)	10(100%)	86,67%
Duplication	43(84,31%)	4(100%)	20(76,96%)	22(73,33%)	44(72,13%)	30(100%)	80,70%
direct copy	41(80,39%)	4(100%)	20(76,96%)	22(73,33%)	25(40,98%)	30(100%)	70,30%
single read/write	8(15,69%)	0(0%)	4(15,38%)	3(10%)	21(34,43%)	0(0%)	17,82%
interleaved r/w	1(1,96%)	0(0%)	0(0%)	0(0%)	8(13,11%)	0(0%)	4,46%
Propagation	33(64,71%)	3(75%)	5(19,23%)	25(83,33%)	5(8,20%)	30(100%)	49,99%
direct copy	33(64,71%)	3(75%)	4(15,38%)	25(83,33%)	3(4,92%)	30(100%)	48,52%
single read/write	3(5,88%)	0(0%)	2(7,69%)	1(3,33%)	2(3,28%)	0(0%)	3,96%
interleaved r/w	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	32(62,75%)	4(100%)	20(76,92%)	18(60,00%)	20(32,79%)	30(100%)	61,39%
Overinfection test	4(7,84%)	1(25%)	1(3,85%)	0(0%)	0(0%)	0(0%)	2,97%
conditional	4(7,84%)	1(25%)	1(3,85%)	0(0%)	0(0%)	0(0%)	2,97%
inverse conditional	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	46(90,20%)	4(100%)	25(96,15%)	27(90,00%)	50(81,97%)	30(100%)	90,09%

Table 4. PE Legitimate Samples. (Com=Communication & Exchange Applications, MM=Multimedia Apps, Off=Office Apps, Sec=Security Tools, SysU=System & Utilities)

Behaviors	PE ComE	PE MM	PE Off	PE Sec	PE SysU	PE Global
Duplication	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Propagation	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%

Table 5. VBS Legitimate Samples. (EmM=Email Managers, InfC=Information Collectors, Enc=Encoders, DfE=Disk & File Explorers, MwC=Malware Cleaners, RegR=Registry Repairs)

Behaviors	VBS EmM	VBS InfC	VBS Enc	VBS DfE	VBS MwC	VBS RegR	VBS Global
Duplication	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Propagation	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Residency	0(0%)	0(0%)	0(0%)	0(0%)	1(12,50%)	0(0%)	1,67%
Overinfection test	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0,00%
Global detection	0(0%)	0(0%)	0(0%)	0(0%)	1(12,5%)	0(0%)	1,67%

parallel detection of additional behaviors described in the same language, and the possibility to feed detection with traces from other sources such as those coming from the script analyzer. Additional tests in Table 2 have been led using the functional polymorphic engine from [9]. For comparison with actual antivirus products, confronted to the engine, the detection rates were between 15% for most of them, up to 90% for others, but with numerous false positives.

2) Limitations in trace collection: A significant part of the missed behaviors are due to limitations in the collection coverage. However, thanks to the layer-based approach, collection and abstraction can be improved for a given platform or language without modifying the upper detection layer.

With regards to dynamic analysis (PE traces), the first reason for missed detections is related to the configuration of the simulated environment. The simulation must seem as real as possible to satisfy the execution conditions of malware. Problems can reside in the software configuration. 65% of the tested Viruses (53/82) did not execute properly: invalid PE, access violations, exceptions. These failures may be explained by the detection of virtualization or anti-debug techniques thwarting dynamic analysis. Problems can also come from the simulated network. Considering worms, their propagation is conditioned by the network configuration. 75% of the Mail Worms (45/60) did not show any SMTP activity because of unreachable servers. Likewise, Net Worms propagate through vulnerabilities only if a vulnerable target is reachable, explaining that 93% of them did not propagate (14/15). All actions conditioned by the simulation configuration are difficult to observe: a potential solution could be forced branching. Notice that this discussion makes sense for off-line analysis but is less of a problem in real-time conditions where we are only interested in the malicious actions effectively performed.

Beyond configuration, the level of the collection can also explain the failures. With a low level collection mechanism, the visibility over the performed actions and the data flow is increased. All flow-sensitive behaviors such as duplication can be missed because of breakdowns in the data flow. Such breakdowns can find their origin sometimes in non monitored system calls and for the most part in the intervention of intermediate buffers where all operations are executed in memory. These buffers are often used in code mutation (polymorphism, metamorphism). 12% of additional virus duplications (10/82) were missed because of data flow breakdowns. The problem is identical with Mail Worms where 8% of the propagations (5/60) were missed because of intermediate buffers intervening in the Base64 encoding. These problems do not come from the behavioral descriptions but from NtTrace which does not capture processor instructions. More complete collection tools either collecting instructions [21] or deploying tainting techniques [22] could avoid these breakdowns in the data flow.

With regards to static analysis (VB scripts), the interpreted language implies a different context where branching exploration is feasible and the whole data flow is observable. Implemented in the script analyzer, these features compensate for the drawbacks of NtTrace and eventually result in better detection rates. However, contrary to the restricted number of system calls, VBScript offers numerous services. A same operation can be achieved using different managers or interfacing with different Microsoft applications. Additional features could be monitored for a greater coverage: accesses to Messenger, support of the Windows Management Instrumentation (WMI). Moreover, like any other static analysis, script analysis is hindered by encryption and obfuscation. The current version of the analyzer only partially handles these techniques; code encryption is missing for example. Static analysis of scripts is nevertheless easier to consider because no prior disassembly is required and some security locks ease the analysis.

3) Behavior relevance: In addition to data collection, the behavioral model itself must be assessed. The relevance of each behavior must be individually assessed by checking the coverage of its grammatical model. Some behaviors such as duplication, propagation and residency are obviously characteristic to malware. Duplication and propagation are discriminating enough for detection. On the other hand, residency is likely to occur in legitimate programs, during installations for example. To avoid certain false positives, its description could be refined , using additional constraints on the value written to the booting object: the value should refer to the program itself or to one of its duplicated versions. On the other hand, the overinfection model does not seem completely relevant. The problem comes from a description that includes too many restraints limiting its detection. In particular, the conditional structure intervening in the model can not be detected in system call traces. Its generalization could increase detection but the risk of confusion with legitimate error handling would also increase.

4) Performance: Table 6 measures the performances of the different prototype components. Considering abstraction, the analysis of PE traces is the most time consuming. The analyzer uses lots of string comparisons which could be avoided by replacing the off-line analysis by hooking in rel-time for immediate translation. On the other hand, the VBScript analyzer offers satisfying performances. With regards to the detection automata, the performances are also satisfying compared with the worst case complexity. The detection speed remains far below a half second in more than 90% of the cases; the remaining 10% were all malware. The implementation has also revealed that the required space for the derivation stacks was very low, with a maximal stack size of 7. In addition, the number of ambiguities has been measured. If n_e denotes the number of events and n_a the number of ambiguities, in the worst case, we would have $n_a = 2^{n_e}$. But by experience: $n_a << 2^{n_e} \quad and \quad n_a << n_e^2 \quad and \quad n_a \approx \alpha n_e$.

This approximation provides a practical complexity in $\vartheta(k\alpha(\frac{n^2+n}{2}))$ which is more worth considering. Moreover, the algorithm can easily be parallelized in multi-core architectures. Figures 8 and 9 provide graphs of the collected α ratios. **It can be observed that above a certain threshold, an important ambiguity ratio α is already a sign of malicious activity**.

Table 6. Compared performances on mono and multi-core architectures

NtTrace Analyzer	
Data reduction from PE traces to logs	
Total size: 351Mo Reduced logs: 11Mo	Average: 1,3Mo/Trace Reduction ratio: 29
Execution speed	
Core M 1,4GHz	Dual core 2,6GHz
1,48 s/trace	0,34 s/trace
VB Script Analyzer	
Data reduction from VB scripts to logs	
Total size: 1842Ko Reduced logs: 298Ko	Average: 7Ko/Script Reduction ratio: 6
Execution speed	
Core M 1,4GHz	Dual core 2,6GHz
0,042 s/script +0,50 s/ciphered line	0,016 s/script +0,21 s/ciphered line
Detection Automata	
Execution speed	
Core M 1,4GHz	Dual core 2,6GHz
NT: 0,44 s/log VBS: 0,002 s/log	NT: 0,14 s/log VBS: <0,001 s/log

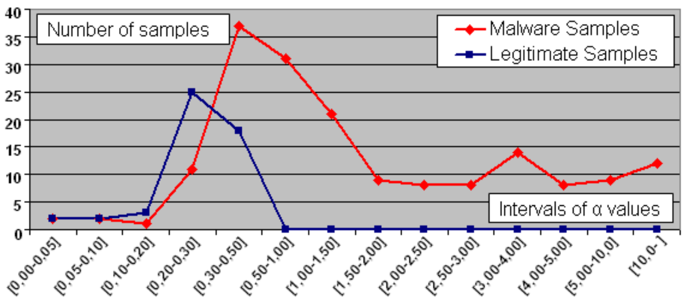


Fig. 8. Ambiguity ratios (α) for PE samples

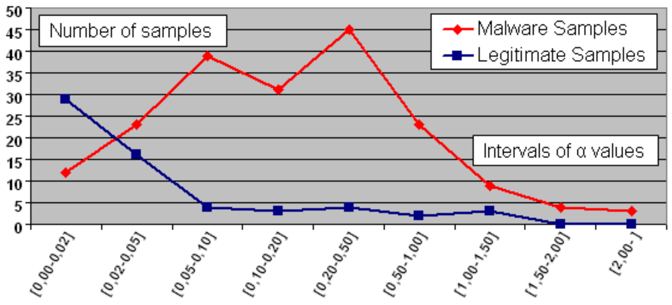


Fig. 9. Ambiguity ratios (α) for VB scripts

7 Conclusions

Detection by attribute automata provides a good coverage of malware using known techniques with 51% of detected PE malware and 89% of VB Scripts malware. The grammatical approach offers a synthetic vision of malicious behaviors. Indeed, only four generic, human-readable, behavioral descriptions have resulted in these detection rates. Unknown malware using variations from these known behaviors should be detected thanks to the abstraction process. In case of innovative techniques, this approach eases the update process. The segmentation between abstraction and detection enables independent updates: in the grammatical descriptions for generic procedures (infrequent), or in the abstraction components for vulnerable objects and APIs. Up until now, the generation of the behavioral descriptions is still manual but the process could be combined with the identification of malicious behaviors by differential analysis proposed by Christodorescu et al. [4]. The experimentations have also stressed the importance of data collection in the detection process. Collection mechanisms are already an active research field and future work can be testing more adapted collection tools deploying tainting.

References

1. Charlier, B.L., Mounji, A., Swimmer, M.: Dynamic detection and classification of computer viruses using general behaviour patterns. Virus Bulletin (1995)
2. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proc. of the Network and Distributed System Security Symposium, NDSS (2005)
3. Bhatkar, S., Chaturvedi, A., Sekar, R.: Dataflow anomaly detection. In: Proc. of the IEEE Symposium on Security and Privacy (SSP), pp. 48–62 (2006)
4. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behaviour. In: Proc. of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineeering, pp. 5–14 (2007)
5. Morales, J.A., Clarke, P.J., Deng, Y.: Identification of file infecting viruses through detection of self-reference replication. Journal in Computer Virology Online (2008)
6. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.C.: A layered architecture for detecting malicious behaviors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 78–97. Springer, Heidelberg (2008)
7. Knuth, D.E.: Semantics of context-free grammars. Theory of Computing Systems 2, 127–145 (1968)
8. Jacob, G., Filiol, E., Debar, H.: Malwares as interactive machines: A new framework for behavior modelling. Journal in Computer Virology 4(3), 235–250 (2008)
9. Jacob, G., Filiol, E., Debar, H.: Functional polymorphic engines: Formalisation, implementation and use cases. Journal in Computer Virology Online (2008)
10. US Department of Defense: "Orange Book" - Trusted Computer System Evaluation Criteria. Rainbow Series (1983)
11. NTInternals: The undocumented functions microsoft windows nt/2k/xp/2003, `http://undocumented.ntinternals.net`

12. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 326–343. Springer, Heidelberg (2003)
13. Cuppens, F., Miège, A.: Alert correlation in a cooperative intrusion detection framework. In: Proc. of the IEEE Symposium on Security and Privacy (SSP), p. 202 (2002)
14. Al-Mamory, S.O., Zhang, H.: Ids alerts correlation using grammar-based approach. Journal in Computer Virology Online (2008)
15. NtTrace: Native api tracing for windows, `http://www.howzatt.demon.co.uk/NtTrace/`
16. QEMU: Processor emulator, `http://fabrice.bellard.free.fr/qemu/`
17. Marion, J.Y., Reynaud-Plantey, D.: Practical obfuscation by interpretation. In: 3rd Workshop on the Theory of Computer Viruses, WTCV (2008)
18. MSDN: Vbscript language reference, `http://msdn.microsoft.com/en-us/library/d1wf56tt.aspx`
19. VXHeaven: Repository, `http://vx.netlux.org/`
20. OffensiveComputing: Repository, `http://www.offensivecomputing.net/`
21. Carrera, E.: Malware - behavior, tools, scripting and advanced analysis. In: HITB-Sec Conf. (2008)
22. Anubis: Analyzing unknown malware, `http://anubis.iseclab.org/`