We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Chapter 1

Proactive Detection of Unknown Binary Executable Malware

Eric Filiol

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/67775

Abstract

To detect unknown malware, heuristic methods or more generally statistical approaches are the most promising research trends nowadays, but their computing and detection performances are generally not compatible with what users do accept. Hence, most commercial AV products still heavily rely on signature-based detection (opcodes, control flow graph, and so on). This implies that frequent and prior updates must be performed. May their analysis techniques be fully static of dynamic (using sandboxing or virtual machines), commercial AVs do not capture what defines malware compared to benign files: their intrinsic actions. In this chapter, we focus on binary executables and we describe how to effectively synthetize these actions and what are the differences between malware and nonmalicious files. We extract and analyze two tables that are present in executable files: the import address table (IAT) and export address table (EAT). These tables summarize the different interactions of the executable with the operating system. We show how this information can be used in supervised learning to provide effective detection algorithms, which have proven to be very accurate and proactive with respect to unknown malware detection.

Keywords: malware detection, program behavior, MZ-PE format, combinatorial methods, learning theory

1. Introduction

To detect unknown malware (or at least malware that are unknown in the antivirus database), heuristic methods or more generally statistical approaches are the most promising research trends nowadays. However, innovative detection algorithms cannot be included in antivirus software due to performance requirements. Among them, we generally face a relatively high false-positive rate, a significant analysis time for a given sample or have memory limit constraints. Having a too high false-positive rate may be a critical issue regarding executable files

which are essential for the operating system kernel, for instance. Reducing the risk of false-positive detection by limiting the scope of efficient heuristic methods is still possible but it does not constitute a realistic solution.

Most of commercial AV products rely on signature-based detection or equivalent techniques. They all use the same scheme to detect malware while dealing with the above-mentioned limitations. The classification about malware signatures by antivirus company can be the following:

- **Object file header attribute** in this case, the header of a portable executable is used to detect whether the file is a malware or not, using combination of the different parts of the file structure. Despite the fact that packers may be used, their identification is relatively straightforward. A similar technique has been proposed in [28] by hashing object file feature. The key advantage of this technique lies on the fact that the result is efficient. Malware belonging to the same family (and written by the same programmer) are easy to detect. On the other hand, if the malware has some modifications while compiled or linked, due to compiler options, the header information may change.

- **Byte level approaches** There are three main possibilities about the byte level:

  - File hashing: the concept is to obtain a hash of whole or part of the malware. This a very common techniques which is quite systematically implemented in antivirus software, especially because it is easy to implement and it does not require a lot of computing resources with respect to the detection process. However, the major drawback comes from the fact that any modification of the binary code will result in a totally different hash value.

  - Character String signatures: a static character string present in the binary code of all the malware of the same family is used to detect the complete family. Griffin, Schneider, Hu and Chiueh [14] had proposed a way to automatically extract strings signatures from a set of malware.

  - Code normalization: the most common approach consist in rewriting some parts of the code using optimization techniques [1]. Junk code, dead code, and one-branch tests are removed while expressions with algebraic identities are simplified. The final code is a normal form that can be easily compared to other malware codes under the same form.

- **Instructions distributions**: the detection here is based on the distribution of the binary executable opcodes [2, 10]. A statistical scheme can be created and used to detect a whole family. Another way is to use N-gram analysis using the method given by McBoost [22].

- **Basic blocks**: the main technique with basic blocks deals with the description of the number of insertions, deletions, and substitutions to mutate a string into another one [3, 12]. To classify a malware from that, it is disassembled statistically and all its basic blocks are extracted. They are then compared to other malware blocks in order to get the smallest differences from one block to another.

- **API calls**: this technique consists into disassembling a full malware to extract the API call sequence. This sequence is compared to that of other malware. The SAVE system [26] is using this method.

Even when heuristics are supposedly used, they do not capture and synthetize enough information to be able to detect unknown malware accurately and proactively. This implies that frequent and prior updates must be performed. May their analysis techniques be fully static of dynamic (using sandboxing or virtual machines), commercial AVs do not capture what defines malware compared to benign files: their intrinsic actions.

In this chapter, we describe how effectively synthetize the essential differences (behaviors, structure, internal primitives) between benign files (or goodware) and malware. Aside a few features about the MZ-PE file header [7], we extract and analyze two tables that are present in executable files: the import address table (IAT) and export address table (EAT). These tables summarize the different interactions of the executable with the operating system. We show how this information, once it has been extracted, can be used in supervised learning (Sections 2 and 3) to provide an effective detection algorithm which has proven to be very accurate and proactive with respect to unknown malware detection.

As a main result, we achieve a very high detection rate with a low false-positive rate while our database has not been updated since 2014. All the techniques presented in this chapter have been implemented in the French antivirus project called DAVFI and presented in Section 4.

Because most of the malware are targeting Windows systems, our techniques are mostly designed for this operating system family. However, our approach has been similarly extended and applied to UNIX systems in the same way (up to the technical differences between ELF executables and MZ-PE executables). Even if we implemented our algorithms to be able to detect UNIX malware specifically as well, without loss of generality we will not present them in this chapter since it would be redundant with what has been made for Windows.

The chapter is organized as follows. Section 2 explains which information to extract from the binary code IAT/EAT and how to use it to capture the essential differences between malware and benign files with respect to their intrinsic behaviors. From that, a very efficient and accurate detection algorithm is designed. To improve further the description of binary executable behaviors, we consider the correlation of order 2 or of order 3 between the different functions involved in the IAT. By considering generic combinatorial structure, we derived a second detection algorithm in Section 3. In Section 4, we present the practical implementation of the algorithms of Sections 2 and 3 in the French antivirus project denoted DAVFI. We conclude in Section 5 and explore the possible evolutions for the results presented in this chapter.

2. Heuristic and proactive IAT/EAT detection

2.1. Technical background: import address table (IAT) and export address table (EAT)

2.1.1. Introduction to IAT and EAT

Any executable file contains a lot of information in the MZ-PE header [21] but some information can be considered more relevant than the others. Tables like import address table (IAT) and export address table (EAT) are, in our case, enough to describe what a program should do or is supposed to do. The IAT is a list of functions required from the operating system by the program. Technically there are two possibilities of importing functions on Windows. The first

one is made explicitly through the IAT during the loading phase of the process before running it, and or during the running phase with the use of the *LoadLibrary* and *GetProcAddress* functions [19, 20]. The second possibility is used by a lot of malware to hide their real functionalities by loading them without referencing them in their IAT. Nonetheless, the functions are used to load libraries and to retrieve functions during runtime and therefore constitute some unavoidable points of passage which can be referenced. In most of the cases, malware, or packers have enough significant IAT to be detectable.

All executable files need an IAT. Without IAT—if this one is empty—it would mean that the targeted program would have no interaction with the operation system. In other words, it is not able to display text or any information at screen, it is not able to access any file on the system and it cannot allocate any segment of memory. Except consuming CPU time — with no result exploitable — it is not supposed to do anything else. Such useless program can be considered as suspicious (since it is suspicious to launch useless programs) or as malware in the most common case. If executable files need IAT, dynamic linked library (Dll) can also provide an EAT. This table describes which functions are exported by a Dll (and which are importable by an executable). Dll generally contains IAT and EAT — except for specific libraries which only export functions or objects. An executable can contain both an IAT and an EAT (the kernel of Windows `ntoskrnl.exe` is a good example). The use of EAT and IAT is a good combination to discriminate most of the libraries since the export and import is quite unique.

However, there are some limits to this system. One lies on the fact that this system only uses and trusts function, executable or library names. If a malware is designed to change every name of function to unknown ones, the system will not be able to give any reliable information any more. In addition, samples which imitate IAT and EAT from real benign files are able to bypass this type of test. Of course, it is a true limit of our model but, surprisingly, in most operational case, such a situation is not common. Most of the packers which are used on malware provides reliable IAT and EAT based on the executable file packed or on the packer itself (which helps to discriminate which packer is used). This observation is extensible to setup programs which are sort of packers.

2.1.2. IAT and EAT extraction

Before we can extract the IAT and EAT, it is necessary to find whether they are present or not. For this purpose it is necessary to analyze the entries of each table in the DataDirectory array of the IMAGE_OPTIONAL_HEADER (or IMAGE_OPTIONAL_HEADER64 in x64) structure. These entries (whose type is IMAGE_DATA_DIRECTORY) are DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT], and DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].

For the IAT and EAT to be present, it is necessary that the VirtualAddress and Size fields in the associated structures are nonzero.

Upon confirmation of the presence of an IAT, it must then be read. Each DLL is stored as a structure of type IMAGE_IMPORT_DESCRIPTOR. From this structure we extract the *Name* field first. It contains the name of the DLL, then the *OriginalFirstThunk* field containing the

http://dx.doi.org/10.5772/67775

address where is stored the primary function, the other being stored in sequence. Each function is stored in a structure of type IMAGE_THUNK_DATA, in which the field *AddressOfData* (whose type is IMAGE_IMPORT_BY_NAME) contains:

- the *hint* value (or Hint field). This 16-bit value is an index to the loader that can be the ordinal of the imported function [24],
- and the function name, if present (*Name* field), i.e., if the function has not been imported by ordinal (see further in Section 2.1.3). In the case of imports by ordinal only, it is the *Ordinal* field of IMAGE_THUNK_DATA that contains the ordinal of the function (if the most significant bit is equal to 1 then it means that the least significant 16 bits are the ordinal of the function [16]).

After getting the name of the function, a pair *dll_name/function_name* (*function_name* is the name of the function or its ordinal otherwise) is formed and stored, and the next function is played until all the functions of the DLL are read, and so on for each imported DLL. On output, a set of pairs *dll_name/function_name* is obtained, which will go through a formatting phase (see Section 2.1.4).

The format of the EAT, although also representing a DLL and all of its functions, is different from that of the IAT. All of the EAT is contained in a structure of IMAGE_EXPORT_-DIRECTORY type. From this structure are obtained the name of the DLL (which may be different in the case of renaming) using the *Name* field, the number of functions contained in the EAT (*NumberOfFunctions* field) and the number of named functions among them (since some functions can be exported by ordinal only) (*NumberOfNames* field).

Then we recover the functions and their name/ordinal. For the named functions, we just have to read two arrays in parallel, whose addresses are *AddressOfNames* and *AddressOfNameOrdinals*: at equal index, one contains the name of a function, and the other, its ordinal. For nonnamed functions, we must then retain all ordinals of named functions and then recover in the table with address *AddressOfFunctions* — which is indexed according to the ordinals of the functions it contains — all the functions whose ordinal has not been retained. After obtaining the set of functions/ordinals, in a similar way to that for the IAT, a set of pairs *dll_name/function_name* is built and then formatted (Section 2.1.4).

2.1.3. Miscellanous data

Let us now detail a few technical points that are interesting to understand IAT and EAT in depth. Microsoft's documentation [18] explains how to export functions by ordinal in a DLL: ordinals inside a DLL MUST be from 1 to *N*, where *N* is the number of functions exported by the DLL. This is interesting and leads us to think that maybe some malicious files do not respect this rule. To go further, it is likely that this also applies to the hint of functions, although no documentation about it could be found. However, the analysis of a few Windows system DLL export tables like *kernel32.dll* and *user32.dll* shows that they comply to this rule. After conducting tests on malicious files and benign files, it turns out that only one "healthy" file (*sptd.sys*, a driver from alcohol120%) does not follow this rule, while a number of malicious files do the same.

2.1.4. Generation of IAT and EAT vectors

After getting all the *dll_name/function_name* pairs from a file, two vectors are created (one for the IAT and one for the EAT). These vectors will the base object for our detection algorithm. In order to generate those vectors, we must build a database containing all the known pairs. A unique ID is associated to each unique pair. This database is populated by a base set of files with a known classification (malicious or benign). The population process is the following:

1. EAT and IAT pairs are extracted from files.
2. For each pair, a unique ID is constructed. This ID is a 64-bit number with the 20 most significant bits representing the DLL and the remaining 44 bits representing the function.
3. For the DLL ID: if the DLL is known, its ID is used. In the other case, a new ID is used, corresponding to the number of currently known DLLs (the first is 0).
4. The function ID follows the same process with known functions.

This population process is only executed manually whenever we would update the database; it is not run during file analysis. The two vectors are created according to this database. For each pairs, its ID is recovered from the database. If it does not exist in the database, the pair is discarded. All the 64-bit numbers are then sorted and stored in a file.

2.2. The detection algorithm

In this section we are now presenting our supervised detection algorithm which works on the vectors built with the data extracted and presented in the previous section. Usually [17, 25, 29] the database of known samples (training sets) must be built before writing the detection algorithm, as far as supervised algorithms are concerned. Such a procedure is led by the knowledge and the learning of what to detect (malware) and what not to detect (benign files). So the training set contains two subsets summarizing the essence of what malware and benign files really are.

2.2.1. How to build the algorithm

Our solution is quite different. Indeed, if we know beforehand which data to use to perform detection, we did not know how to build the database to make it reliable and accurate enough for our algorithm. Which data to select among a set of millions of malware samples and of benign files, in order to get a representative picture of what a malware is (or is not) for the algorithm, is a complex problem in itself. Our approach has privileged the operational point of view. We have designed the algorithm as formal as possible and we have applied it on sets of malware and on a set of benign files to allow it to learn by itself, building the database after the creation of the algorithm. In other words, the algorithm is designed to use a minimal database of malware and of benign files at the beginning and this one is able to perform minimal detection helping to develop the database with samples undetected to improve results. We thus consider an iterative learning process, somehow similar to boosting procedure [15, 29].

Such an approach privileges experimental results and design of algorithms to detect unknown malware. Indeed, the algorithm uses subsets of malware samples which are the most representative of their families. Derivatives and parts of known malware (or variants) can be recognized since they have been learned previously. "Unknown" malware uses most of the time old fashion technologies, with the same base behaviors, and hence our algorithm is able to detect a lot of them with such a design and approach. For sake of clarity, the description of our algorithm starts with building the detection databases (training sets). To help the reader, we suppose in this part that we (already) have a known detection algorithm which is presented right after in the chapter (refer to Section 2.2.3).

2.2.2. Building the detection database (training set)

The heuristic algorithm we have designed uses a database of knowledge to help it to make decisions. Of course, algorithm databases are built with the two different types of files it is supposed to process and decide on: benign files and malware. The use of a combination of samples from malware and benign files gives the best results since they are suitably chosen. The way the database is built is the key step of our heuristic algorithm, since it affects directly the results we obtained. However, we must stress on the fact that we would obtain the same results for different malware/benign files subsets, as long as those sets are representative enough of their respective family. Somehow, this step can be seen as a probabilistic algorithm.

From a simple observation, more than the number of samples we could set in the database, the diversity of samples helps better to get the widest possible spectrum of detection. Smaller and more diverse the database is, faster and better are the results obtained. Indeed, if the database is too big, searching inside will be too much time-consuming, thus resulting in the impossibility to use it in real time. Only the most representative malware of a family must be included in the database (and similarly for the benign files).

First, we need a detection function which is the one used by our algorithm. At the beginning, the database used by this function is composed only with a small set of malware arbitrarily selected (denoted M) to be representative of the family we want to include. Such a detection function can be defined as follows. From any sample S we want to analyze, we have a prior detection function D_M which is of the form

$$D_M(S) = \begin{cases} 0 & \text{if } S \text{ is a non malicious} \\ 1 & \text{if } S \text{ is a malware} \end{cases} \quad (1)$$

It is not required that function D_M exhibits huge and optimal detection performances. So a known and initial malware (respectively benign file) sample set is enough to initiate the process. To expand the databases (malware and benign files), Algorithm 1 is used. This approach is more or less similar to boosting methods such as Ada-Boost [11, 15].

Algorithm 1 Database creation algorithm (training set)

Require: A set of files S_f to analyze (which has n files) and a maximal error detection rate ϵ.

Ensure: Database files S_d (malware) and S_{ud} (benign files).

while $\frac{|S_f|}{n} < \epsilon$ **do**

for $\{s\} \in S_f$ **do**

if $D_M(s) == 1$ **then**

$S_d \leftarrow \{s\}$

else

$S_{ud} \leftarrow \{s\}$

end if

end for

$M = M \cup S_{ud}$

if $|S_d| == 0$ **then**

break

end if

end while

Algorithm 1 also enables to control the error detection rate ε for a given malware family (with $\epsilon \in [0,1] \subset \mathbb{R}$). Indeed, if ε is chosen too small, the algorithm can include all the files from S_f. Of course, the representativeness of files in S_f is a key point to use the algorithm. Working with several different samples of the same family is, most of the time, the best approach. Another possibility of control is to use the rate of detected files such as $\frac{|S_d|}{S_{ud}} < \epsilon$ with ϵ close to zero.

The building of database is performed family per family (of malware). It is possible to make it faster mixing multiple relevant samples from different families in one set. For example, to build the benign file database, one can choose files among those coming from `C:\ Windows`. In fact, the initial choice of incoming files defines the relevance and the diversity of the database. Starting from a small set of these files, we launch Algorithm 1 on the remaining files until we have got enough file detected by the database created on the fly.

One key advantage of this principle lies in the fact that we can increase the size of the database in the future without prior knowledge of a malware family. At the first time we created the database, if the diversity of malware families was enough good, it is possible to include new samples of malware without knowing its type/family. In fact, malware share strong IAT and EAT correspondences and similarities with many other families, in most of the cases. It means that malware can be detected by the database previously built even if we never included any sample from its family. In other words, we can use this property to increase the size of the database by adding undetected malware coming from different families into the current database. Taking a file defined as malware (which could be given by any trusted source or by a prior manual analysis), if this one is not detected by our algorithm, we can include it in our

database in order to improve the detection of its family. It is a simple way to improve the accuracy of the detection.

2.2.3. The detection procedure: the K-nn algorithm

Once the structural analysis is achieved and the database (training sets) has been built, then the detection tests occur by using the IAT and EAT vectors which have previously generated. This is the second part our module is in charge of, and which aims at deciding the nature of a file.

Detection tests are split into two sets: the IAT comparison test and the EAT comparison test. The principle of those tests is: the unknown file's IAT (or EAT) is compared to each element of the base of benign files and to each element of the base of malicious files. The $k = 2p + 1$ files that are closest to the unknown file are kept with their respective label (malware or benign file). A decision is then made based on these k files to decide which label to give to the file under analysis. This test thus uses the method of k-Nearest Neighbors [15, 29], which has been modified for the occasion. In both cases, the input consists of the k closest training examples in the feature space.

2.2.3.1. Vector format limits

While this format allows an optimized storage of the IAT/EAT, it faces several constraints that limit its use. The first constraint is a space constraint, which actually is not an intractable problem. Our encoding limits to 2^{20} possible DLLs and to 2^{44} functions per DLL. Today, this is more than enough, but we must keep in mind that this limit exists, and could be a problem in a (very far) future.

The second constraint lies in the fact that our vectors do not have a fixed length. It is a problem if we want to use standard distance functions, like the Euclidean distance. We could have used a similar vector format in which each possible couple was given a 0 or 1 number depending on whether it was present in the file or not. But the length would have been around 106 (about the current size of the database) instead of around 103 (for large files) with the current format. It would have a bad impact on the performances of real-time analysis, and hence it would have increased the time of analysis by too a high factor. In order to optimize the computation time, all the vectors in the bases and generated during analysis are sorted.

2.2.3.2. The similarity measure

In order to determine the nearest neighbors, we need a function to compare two IAT/EAT vectors of different sizes. The format prevents the use of standard distances (because to use a standard distance, the IAT/EAT vectors should have the same size, i.e., always the same number of imported/exported functions in each file, which is quite never the case). It was therefore necessary to find a function fulfilling this role and to apply it our format. Let us adopt a few notations:

- An IAT/EAT vector of size n is written as $\sigma = \sigma_1\sigma_2\ldots\sigma_n$ where $\sigma_i \in \{0,1\}^{64}$ (64-bit integers). The set of such vectors is denoted Σ_U.

- The inverse indicator function $I : E,F \rightarrow \{0,1\}$ is defined such that $\forall x \in E, I_F(x) = 0$ if $x \in F$ and 1 otherwise.
- If v is an IAT/EAT vector, $E_v = \{\sigma_i\}$ (this notation describes the fact that vectors are implemented as lists of 64-bit integers).

The function we use to compute the degree of similarity between IAT or EAT vectors is then defined by:

$$\forall a \in \Sigma_U, \forall b \in \Sigma_U,\ f(a,b) = \frac{1}{|a| + |b|} \left(\sum_{i=1}^{|a|} I_{E_b}(a_i) + \sum_{j=1}^{|b|} I_{E_a}(b_j) \right) \quad (2)$$

It is easy to prove that this function satisfies the separation, the symmetry and the coincidence axioms as any similarity measure has to.

2.2.3.3. The decision algorithm

The detection algorithm to decide the nature of a file (malware or benign) is given by Algorithm 2. It is composed of two parts in order first to reflect the importance of similarity optimally and second to eliminate some neighbors who are there only due to the lack of data.

The first part consists in filtering the set of neighbors that the k-NN algorithm returns to refine the best decision based on the neighbors that are really close. For this purpose, a threshold is set (50% for now) and only neighbors with a higher degree of similarity (i.e., that the function f returns a value less than 0.5) are kept. Then classical decision is applied to this new set: the file is considered closer to the base with the most representative among the neighbors.

The second part is used in the case when an equal number of representatives in each base, is returned (situation of indecision). All the neighbors are again considered, and again the file is considered closer to the base with the most representatives among the neighbors. If k is odd, it helps to avoid indecision (majority decision rule). It was therefore decided that all k are used odd in order not to fall in the case of indecision.

Algorithm 2 Algorithm used to classify a file

Require: A vector X representing a file to analyze, a malware vector base B_M and a benign vector base B_B.

Ensure: A Boolean value indicating whether the file is malicious.

$i \leftarrow 0$

for $\{b\} \in B_B$ **do**

d = f(X, b)

if $d == 0$ **then**

Return(false)

Else

neighbors[*i*] += (*d, brnidn*)

i++

end if

end for

for $\{m\} \in B_M$ **do**

d = f(X, m)

if $d == 0$ **then**

Return(false)

else

neighbors[*i*] += (*d, malicious*)

i++

end if

end for

if MaxNeighbors(neighbors) == malicious **then**

Return(true)

else

Return(false)

end if

2.3. Detection and performances results

In order to test and to tune up our algorithm, we have defined many tests. On the one hand, we have tested the modification of the number of neighbors' parameter in the *k*-nn algorithm. This test is made in order to observe for how many neighbors the test is the most efficient. Then, on the other hand, we performed tests on databases to measure results of the algorithm. Of course, the detection algorithm is used with the most efficient number of neighbors obtained in the first test.

Increasing the number of neighbors by more than 9 does not change the results significantly. In fact, keeping the number of neighbors as minimal as possible is a better choice since it has an impact on the response time of the algorithm — a key point when we used it in real-time detection conditions. The results about this test are displayed in **Figure 1**. For the final test, we have put the algorithm to the proof with two sample sets. One is composed of 10,000 malware (extracted from different families and unknown from our databases) and one composed of legitimate files composed of executable files extracted from a clean Microsoft Windows operating

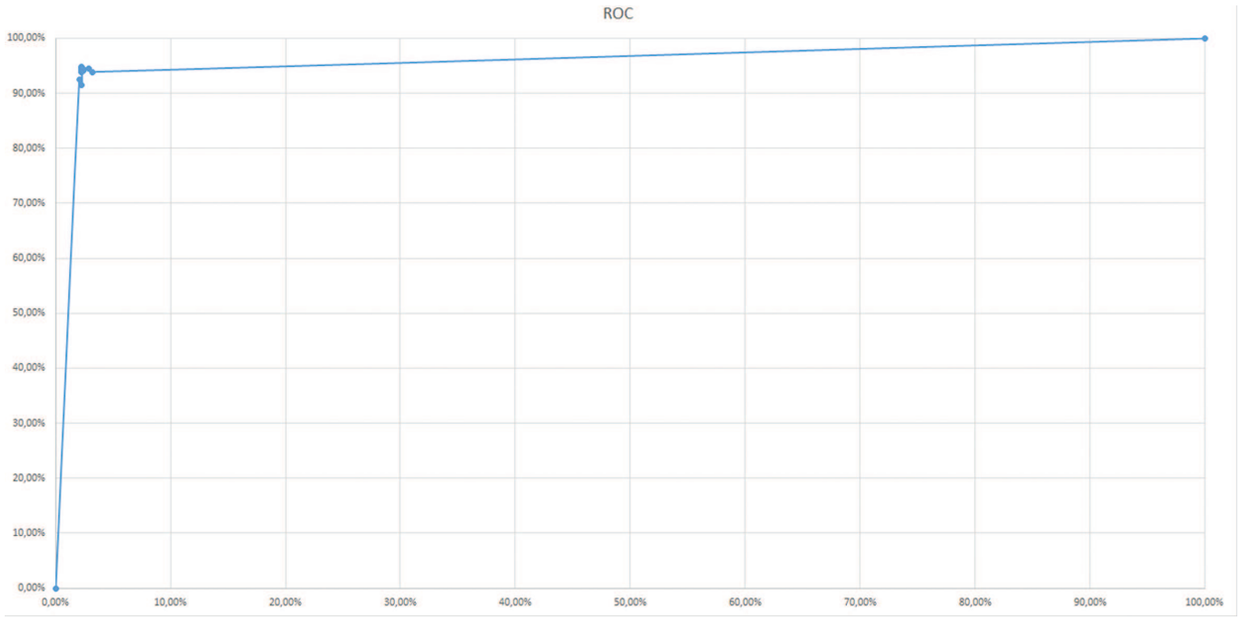

Figure 1. ROC summarizing the detection algorithm performances.

	Malware set	**Benign files set**
Detected as malware	95.028%	4.972%
Detected as benign file	2.053%	97.947%

Table 1. Algorithm performances results.

system (around 131,000 files). The results are given in **Table 1** These results show that the algorithm is quite efficient to detect similarities between different executable files. Nonetheless, it is not enough to use it for detection in real time only since the rate of false-positive detection is too high to be acceptable. To prevent such a case, our algorithm in module 5.2 is chained with other techniques (see Section 4). This is the most efficient approach since we succeeded in making the residual false-positive rate tends toward 0.

3. Combinatorial detection of malware by IAT discrimination

As we did in the previous section, we now consider a mix between the object file header and the API call. We are orienting our research toward the Import Address Table (IAT) and especially the correlation between IAT functions that are used either by malware or by benign files or used by both.

For this purpose, we use supervised learning techniques. The training models aims at building vectors that capture the combined use of specific IAT functions. We have observed that the subsets of specific functions significantly differ depending on the nature of the executable file —

malware and benign files. Then, the testing phase enables to detect codes, even when unknown, with a very good true positive rate while keeping the false-positive rate very low.

3.1. The IAT functions correlation model

To build our model (our training sets), we have to extract the specific IAT functions and to build specific vectors that describe their combined use by malware, benign files and blacklisted IAT functions. We thus build two vector sets, one set which models malware, the second the benign files. The (unique) blacklist vector set describes specific IAT functions which must be considered as used systematically by malware only (see further).

Each of our vectors is implemented as a multiprecision integer by using GMP [13]. Each bit of this integer represents the presence or absence of a predefined (specific) function in its Import Address Table. This implementation approach allows to perform vectorized computation with simple bitwise logical operators. The predefined functions are derived from the extraction of all the Export Address Table from the dynamic-linked library in the operating system. For example, **Table 2** summarized the occurrences of the predefined functions in both malware and legitimate files.

The vector for the malware is: 001 0011 $\Rightarrow$ 19, the vector for the benign file 1 is: 101 0101 $\Rightarrow$ 85 and the vector for the benign file 2 is: 111 0100 $\Rightarrow$ 116. In this way, we can easily and quickly detect which function from which dynamic-linked library is used by malware or benign files. The Dll name:function name indices are arbitrarily ordered, provided the chosen ordering remains the same for all vectors.

3.1.1. Creation of initial vectors

The different sets containing the vectors we generate are essential components in our detection engine. We used a fresh install of Windows 7 professional with all update at January 1st, 2015. Three vector sets are created: for benign files, malware and for the blacklisted functions. In order to obtain a list of all functions, we extracted all of them in each dynamic link library which was present in the Windows system. We obtained a total of 76,669 functions in 1568 dynamic link libraries.

Dll name	Function name	Malware	Benign file 1	Benign file 2
dll_i	F1	1	1	0
	F2	1	0	0
	F3	0	1	1
dll_j	F1	0	0	0
	F2	1	1	1
	F3	0	0	1
dll_k	F1	0	1	1

Table 2. Vectors creation table (drawn from [8]).

3.1.1.1. Malware vector set

The vectors are created by extracting the import address table from a set of 3567 malware. This set covers 95% of the different families for the two last years. After analysis and cleaning steps (especially for discarding duplicated vectors), we have obtained more than 1381 vectors. Let us remind that many malware use packers to delay the analysis or to make it less straightforward. Whenever a benign file packs its code a packer that is generally used by malware, we then decide it as malicious.

3.1.1.2. Goodware vectors

Goodware vectors are created from the executable files on a clean installation of Windows 7. We have obtained a set of 985 vectors.

3.1.1.3. Blacklist vector

This blacklist vector set is created by considering all undocumented functions contained by the Microsoft dynamic-link library on a native Windows 7 professional, as well as a few functions used by malware only. Development standards now make nowadays compulsory not to use undocumented functions (may them be Windows functions or not). As a consequence, it is a key point to keep in mind that there is no real reason for a legit program to rely on or to use undocumented functions from the Microsoft dynamic-link library. Those functions can become deprecated at any time without explanations from Microsoft. As a consequence, any legit program does not have to use them. In order to add some other functions, we also take the PeStudio blacklist [23] into account. The blacklist vector references around 47,040 blacklisted functions.

3.1.2. Correlation between functions and function subsets

In order to improve the detection scheme presented in Section 2, we decided to use the correlations between functions. Indeed, program behaviors can be described by a set of functions, which are generally indexed by time (in order words, the order according to which functions are called, matters). We thus intend to use the information describing the simultaneous occurrence of subsets of functions. Since a few years, compilers do no longer preserve the time ordering of functions in the IAT. To retrieve this information either we have to reverse the binaries and analyze the code or to perform a dynamical analysis from execution traces. Hence, subsets can be considered in place of vectors (ordered subsets). To model this, we are going to use all subsets of size 2 (pairs) or of size 3 (triplets). In other words, we intend to capture more closely the behaviors by considering the call of any two (resp. three) possible functions.

From the initial vectors of size n we then build pair-vectors or triplet-vectors.

Pair-vectors have size of $\binom{n}{2}$ while triplet-vectors have a size of $\binom{n}{3}$. For an easy implementation, we will keep on representing these vectors as GMP integers.

All possible function pairs and function triplets are ordered according to some arbitrary ordering, for example, $(1;2)$, $(1;3)$, …$(1;n)$, $(2;3)$, …, $(n-1;n)$. For example, when considering data given in **Table 2**, we produce the data given in **Table 3** (due to lack of space, we show only pairs that are effectively present in the binary code of at least one of the files). For ease of writing we call *binomial sets* the function subsets of size 2 and *trinomial sets* the function subsets of size 3. We thus produce three new vectors sets.

3.1.2.1. Binomial set vectors

From the previous initial vectors produced in Section 3.1.1, we generate binomial set vectors for both benign files and malware. For each vector and for any function binomial set, we check whether this set is present (the corresponding vector bit is set to 1) in the executable or not (the bit is set to 0). If again we consider the result of **Table 3** drawn from [8], the malware file is defined by the following binomial sets: (1;2), (1;5) and (2;5). Then the resulting binomial set vector is 00000000000100001001 where the binomial (1;2) is the least significant bit and the binomial (n – 1, n) is the most significant bit. Goodware are then similarly defined by the two followings vectors: 01000010100000010101O and 11100011100000000000. There is only one pair, (1;5), in common. **Table 4** summarizes the number of subsets for each category.

Bit 1	Bit 2	Malware	Benign file 1	Benign file 2
1	2	1	0	0
1	3	0	1	0
1	5	1	1	0
1	7	0	1	0
2	5	1	0	0
3	5	0	1	1
3	6	0	0	1
3	7	0	1	1
5	6	0	0	1
5	7	0	1	1
6	7	0	0	1

Table 3. IAT function pairs (example drawn from [8]).

	Count
Goodware	1,753,640
Malware	2,036,311
Common	433,119

Table 4. Details of count in binomial sets.

3.1.2.2. Trinomial set vectors

In the same way we did for binomial set vectors, we have produced three sets for the trinomial sets (see **Table 5**).

3.1.2.3. Common sets

In order to make the analysis more accurate, we removed all the common sets for both the binomial and trinomial sets. Since there are present at the same time both in malware and benign files, they do not provide meaningful information. As an additional advantage, we also reduce the size of the database and we spare time and memory (see **Tables 4** and **5**) [8].

3.2. The detection algorithm

We use a variant the K -nn algorithm [17] whose aim is to compute the distance of a given vector (the file to analyze) to the sets of the training database. We then label the vector with respect to the set which is at the shortest distance. In practice, to classify an executable as a malware or a benign file, the detection algorithm consists in five tests. Three of them use directly the initial vectors extracted from its Import Address Table. The last two tests use the binomial and trinomial set vectors.

The detection algorithm is summarized in Algorithm 3 and implements several steps:

- The first test is a comparison with the blacklist vector. A simple bitwise AND is performed between both vectors. If the result is different from zero (characteristic malware functions are indeed shared by both vectors), then the executable is considered as a malware.
- The second test consists in performing a bitwise XOR between the file vector to classify and all vectors from the malware and legitimate file sets. The label (malware or benign file) will be determined by the shortest distance. We only keep the $2p + 1$ best values (usually $p = 15$) and apply a majority voting. Moreover, we also analyze whether there is gap in these $2p + 1$ distances. If we notice such a gap, we consider that the label for the file must be the same than that of the family of the vector for the gap. For example if the best value is 3 with malware label, and the second is 27 with the nonmalicious label, the file is considered as a malware (since $27 – 3 = 24$ is far greater that generally observed).
- In the third test, we compare vectors with a bitwise AND test. The classification label is determined by the largest distance: the bigger the result, the closer is the vector to the corresponding vector set. In the same way we do the XOR test, we use gap criteria to discriminate a family in case of uncertainty. It is worth noticing that the AND and XOR test

	Count
Goodware	373,026,049
Malware	336,423,103
Common	283,4537

Table 5. Details of count in trinomial sets.

are not the same. While the XOR test enlightens the dissimilarities, the AND test favors similarities. In fact, both tests are complementary to each other.

- The two last tests are based on the binomial and trinomial vector sets, we calculate which set yields the most common matches and hence we decide the label accordingly.

Algorithm 3 IAT-based combinatorial detection algorithm (vectors and files are represented as GMP integers; binary operators are computed bitwise over GMP integers)

Require: File f to analyze. Blacklit vector $\mathcal{B}$, malware vector set $\mathcal{M}$ and benign file vector set $\mathcal{G}$, malware binomial set vectors $\mathcal{MBS}$, malware trinomial sets vectors $\mathcal{MTS}$, benign file binomial set vectors $\mathcal{MBS}$, benign file trinomial sets vectors $\mathcal{MTS}$.

Ensure: File label (malware [1] or nonmalicious [0]).

type $\leftarrow 0$

compute $v = \mathcal{B}\ AND\ f$

if $v \not= 0$ **then**

type++

end if

compute the XOR distance of f with vectors in $\mathcal{M}$ and $\mathcal{G}$

keep the 31 best vectors with their distance from f and their label (malware or benign file)

if Malware labels are the most represented **then**

type++

end if

compute the AND distance of f with vectors in $\mathcal{M}$ and $\mathcal{G}$

keep the 31 best vectors with their distance from f and their label (malware or benign file)

if Malware labels are the most represented **then**

type++

end if

compute $d_{\mathcal{MBS}}$ and $d_{\mathcal{GBS}}$ (resp.)the distance of f with vectors in $\mathcal{MBS}$ and $\mathcal{GBS}$)

if $d_{\mathcal{MBS}} > d_{\mathcal{GBS}}$ **then**

type++

end if

compute $d_{\mathcal{MTS}}$ and $d_{\mathcal{GTS}}$ (resp.)the distance of f with vectors in $\mathcal{MTS}$ and $\mathcal{GTS}$)

if $d_{\mathcal{MTS}} > d_{\mathcal{GTS}}$ **then**

type++

if type ≥ 2 **then**

return 1 (malware)

else

return 0 (nonmalicious)

end if

3.3. Results and performances

In this section, we now detail the results of those different steps of the detection algorithm. With the initial sets only and without learning phase, we have a detection rate more than 98% and a really small false-positive rate (less than 3%). The false positive is mostly due to legitimate software, which uses packers that we can wrongly label as malware. However, by combining with white listing techniques (as we did in the French AV project DAVFI, see Section 4), the false-positive rate systematically tends toward zero. As explained before, only a very few legitimate software are using code packing as malware usually do.

3.3.1. Blacklisted function vector

The result for this test is generally zero (in more than 97% of the cases). But whenever this result is different (nonnull) we are certain that the file is a malware. This indicator about undocumented functions from the Windows API is discriminant only if the executable uses one of these functions.

3.3.2. XOR & AND tests

The tests for bitwise XOR and AND were the two first tests implemented (**Tables 6** and **7**). With a rather small database for each set (less than 30 Mb), we detect 99% of malware correctly. The following tables show the results using a part of the database only [8]. The aim is to determine whether a reduced database would provide significantly similar results thus enabling to spare memory.

To create the partial database, we keep only the most significant vectors in terms of information contained. This is directly connected to the sparsity of vectors. Another way to select the vectors to keep consists in computing their respective *Information Gain* [17]. Let us consider a vector v. Its information gain is given by the formula:

$$IG(v) = \sum_{v_j \in \{0,1\}} \sum_{C \in \{C_i\}} P(v_j, C) \log\left(\frac{P(v_j,C)}{P(v_j).P(C)}\right), \tag{3}$$

where C is the class (malware or benign file), v_j is the value of the j-th attribute, $P(v_j, C)$ is the probability that it has value v_j in class C, $P(v_j)$, is the probability that it takes value v_j in the

% of original base	Size on disk	Detection rate	Time
100	39 Mb	99	2 s
90	35 Mb	93	2 s
80	31 Mb	86	2 s
75	29 Mb	81	2 s

Table 6. Results for the AND test.

% of original base	Size on disk	Detection rate	Time
100	39 Mb	98	2 s
90	35 Mb	97	2 s
80	31 Mb	87	2 s
75	29 Mb	80	2 s

Table 7. Results for the XOR test.

whole training set (database) and $P(C)$ is the probability for the class C. With only 75% of the whole database, we detect 80% of the malware, while the rate of false positive is close to 0.

3.4. Binomial and trinomial set vectors tests

With the binomial and trinomial set vectors we have built in the previous part, we detect 99% of malware containing an Import Address Table. Whenever an executable file has no IAT, it is strongly suspected to be a malware. Consequently it is labeled as such. However the size of database is relatively big: 121 Mb for the binomial set vectors and 34 Gb for the trinomial set vectors. To reduce the database sizes, once again we keep only the most significant vectors in each set. In this way, we reduce the time to analyze a file and the size of database. **Tables 8** and **9** give the best ratio to keep.

3.4.1. General results

The efficient approach consists in combining and chaining all the tests using different possible decision rules (one of the most efficient is the maximum-likelihood rule). The detection rate is then more than 99% while the false-positive rate is very close to 0 (without additional white listing techniques). **Tables 10** and **11** show the detection rate depending on the size of the

% of original base	Size on disk	Detection rate	Time
100	121 Mb	98	67 s
90	109 Mb	97	53 s
80	96 Mb	90	47 s
50	60 Mb	80	30 s

Table 8. Results for binomial set vectors.

% of original base	Size on disk	Detection rate	Time
100	34 Gb	99	287 s
90	30 Gb	98	240 s
80	27 Gb	93	223 s
50	17 Gb	82	153 s

Table 9. Results for trinomial set vector.

% of original base	100	90	80	70	60	50
% of detection	99	98	94	89	87	84

Table 10. Detection rate.

% of original base	100	90	80	70	60	50
% of false positive	1	3	7	12	17	24

Table 11. False-positive rates.

initial database. The following table indicates us the rate of false positives on all our tests depending of the database size. As we can see, the rate of false positive is very good. False positive can be explained as follows:

- Software installers generally embed compressors and packers. Hence we observe the presence of a small IAT with many compression imports.
- The DotNet environment is developing more and more. DotNet files have really a small IAT. An optimization would be to analyze the internal imports.
- Update only programs. These programs are generally really near of webdownloaders (a functionality shared with malware), because they basically only try to connect on specific websites in order to check whether any new version is online.

In all three cases, white listing techniques and/or additional analysis routines (such as those presented in Section ??) will make the false-positive rate tends toward 0.

4. The DAVFI project

4.1. Presentation of the project

The DAVFI project [5] (standing for *Démonstrateur d'Antivirus Français et International* or *French and International Antiviral Demonstrator*) was a 2-year project (from October 2012 to September 2014) partially funded by the French Government (*National Fund for the Digital Society*). The objective of this project was to design, to implement and to test a proof-of-concept for a new generation, sovereign, multi-platform (Android, Linux, and Windows) open antivirus software.

http://dx.doi.org/10.5772/67775

The final proof-of-concept has been delivered in September 2014 and is based on a strongly multithreaded architecture. The latter is made of several modules which are chained and operate within two main resources: a resident notification pilot and an antiviral analysis service. The latter embeds two analysis streams, one for binaries and executable files, the other to process documents (and malware documents) specifically. In 2015, after a technical and operational validation by the French Directorate General of Armaments has been transferred to the private sector for the industrialization process. By now this project equips the French National Gendarmerie's computers (Linux version).

The DAVFI's general structure (we will focus on the Windows version) is summarized in **Figure 2**. The detailed internal structure of the executable analysis chain is depicted in **Figure 3**.

DAVFI/OpenDAVFI's detection architecture is based on several modules. Whenever a relevant file is accessed, antivirus' kernel drivers notify the analysis service for the file analysis. Then many possibilities are considered. First, the file may be already known by the analysis system to be a nonmalicious file. Such a file can be defined as part of the system or already scanned by the antivirus and therefore has not to be detected as malicious (**Figure 2**). For this purpose, dynamic white-listing and black-listing modules have been designed and implemented (modules 1.1, 1.2 and 1.3). Second, the file is a document file and must be analyzed by a specific module (module 4 in **Figure 3**) [9]. Third, if we deal with a script file, it must be analyzed by another specific module. In the last case of a binary executable file, the analysis involves the module 5. This module is in fact a chained sequence of sub-modules designed to filter the detection of a binary file (note that other modules are composed in the same way) as depicted in **Figure 3**.

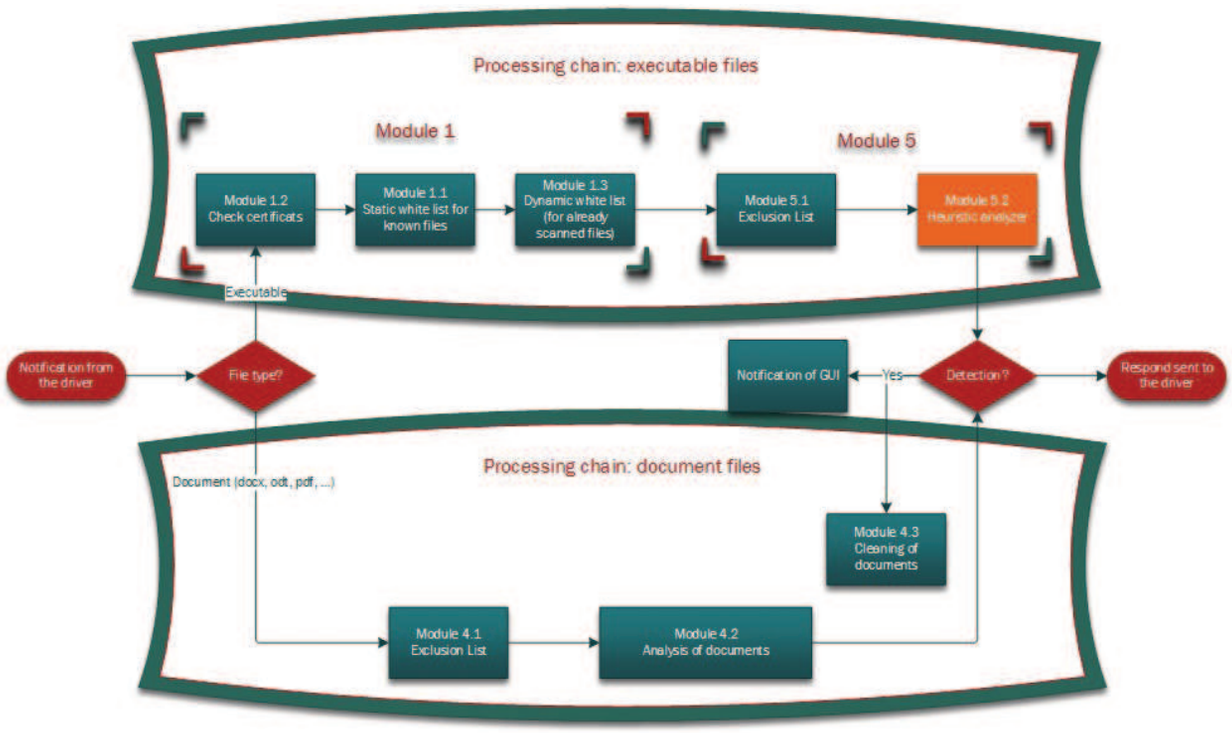

Figure 2. Overall structure of the windows DAVFI/OpenDAVFI application.

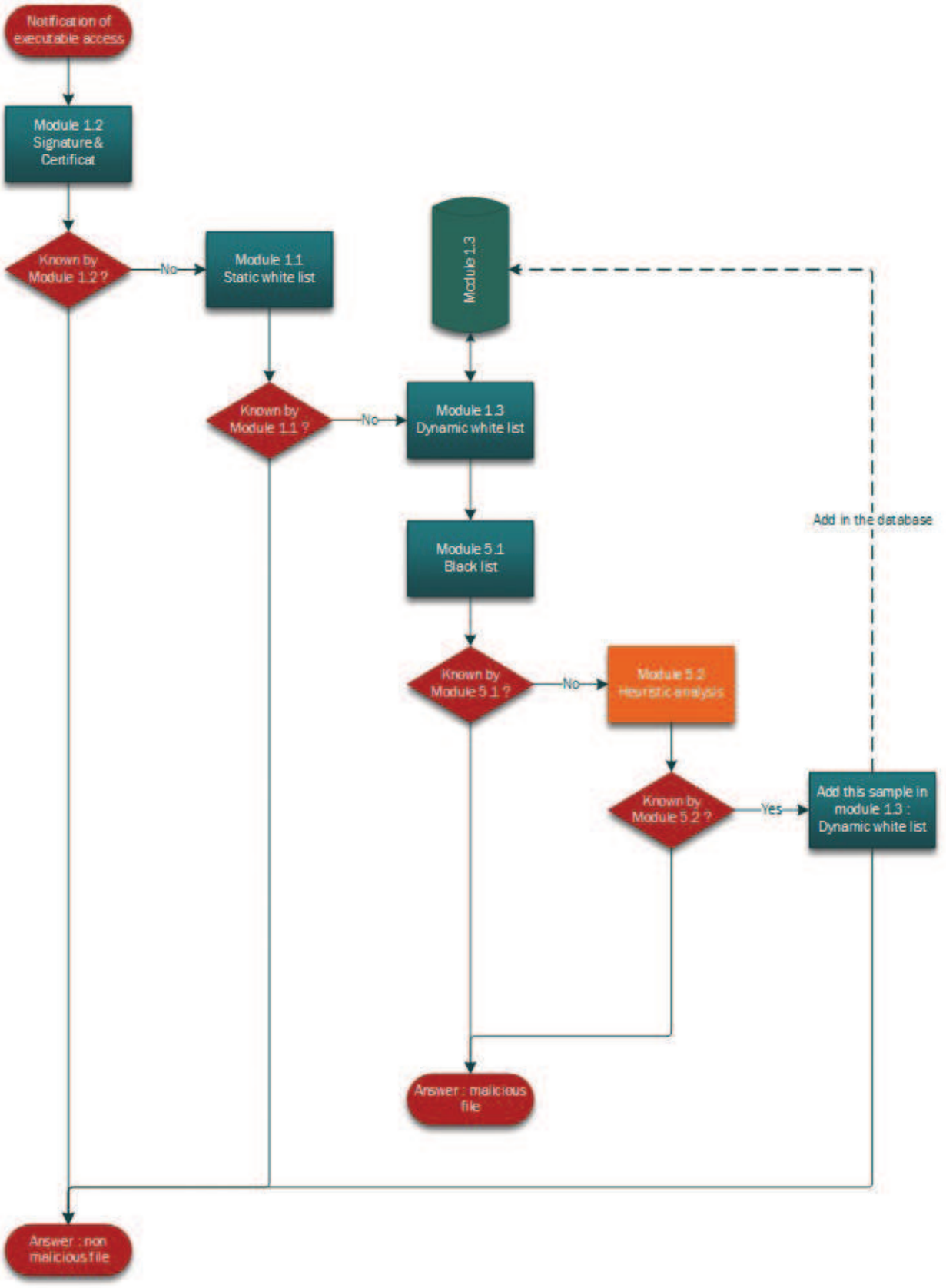

Figure 3. Overall structure of the windows DAVFI/OpenDAVFI executable file analysis module.

Whenever the module 5 starts, it checks with SEClamAV antivirus engine whether the file is a well-known malware or not. SEClamAv is used in our case for performance purposes to notify the heuristic detection module (module 5.2 which implements the detection algorithm presented in Section 2) with unknown files only. The heuristic module, which has been developed, is designed to detect both unknown malware and known malware. It is made of three parts: a header structural analysis [7] and two heuristic submodules which process the information contained in the *Import Address Table* (IAT) and in the *Export Address Table* (EAT) [6, 8]. It is also worth mentioning that the first filter in the DAVFI's analysis system is able to discard from detection all legitimate Windows kernel files (white-listing approach) or well-known benign files. This greatly reduces the false-positive detection rate.

Since heuristic detection is generally time-consuming, module 5.2 embeds a structural analysis chain which operates first [7]. Most AV software still uses detection techniques (either static or dynamic), which are however mostly based on the general concept of (static or heuristic) signature. However, we have observed that many malware do not comply to the Microsoft specifications with respect to the MZ-PE format [21]. Indeed, implementing malware techniques and tricks to fool a number of protection, detection or analysis techniques requires for the malware writer to take liberties with the file format specification. Consequently, a simple structural analysis with respect to this file format allows to identify executable that are indeed surely malware. As a consequence we avoid useless, time-consuming processing with the subsequent heuristic module.

4.2. Testing and technical evaluation

This project has been tested many times intensively during the two years of the project and then by the Directorate General of Armaments. A users committee (French Banks, French DoD, Prime Minister Office, and so on) has also been built for the DAVFI project. The aim was to involve end users, to have their operational feedback regarding antivirus software and to make them test a few modules in real-life conditions. Moreover, they feed us with unknown malware (usually manually detected in their respective CERT during the very first hours of the attack), most of them being not detected by commercial AV software (we use the VirusTotal [27] website for checking this point). Most of the samples provided related to targeted attacks. Final testing were organized as blind testing (we did not know which files were malware or benign files).

The performance results are very good and can be summarized as follows:

- The overall detection rate (true positive) is more than 97% while false-positive rate equal to 0.
- These overall results include unknown malware at the time of testing (the malware nature has been confirmed by manual malware forensics analysis). It is worth mentioning that the initial databases (presented in Sections 2 and 3 were not updated during the different testing phases).
- New tests in mid-2016 confirmed the previous results without database updating.

5. Conclusion and future work

In this chapter we have presented a different supervised detection algorithms working on data extracted from the IAT and EAT of binary executable files (Windows and Unices) and more broadly from their header. These particular pieces of information do not only describe the executable in a static way more precisely (use of far more complex and rich signatures) but also they capture the information related to program behaviors.

The overall performances which we have achieved show that is possible to detect unknown malware proactively and accurately. This yields enhanced detection capabilities while requiring far less database update. Beyond the experimental analysis, operational testing of those techniques has been performed on malware coming from the real world in real conditions. The results which have been observed fully satisfy the operational constraints and specifications with respect to unknown malware detection.

Future work will address the combinatorial modeling and processing of information contained in IAT/EAT. While we have considered mostly statistical aspects and initiated their combinatorial analysis in Section 3, it is possible to have a far more precise processing of this information when using combinatorial structures to synthetize the concept of behaviors and hence base a more accurate detection on the dynamical information contained in the code.

We also intend to extend the information used for detection. The study of data section or opcodes sections is a possible option in order to increase the number of detection criteria. These sections can provide correlations with the features we already consider.

As far as combinatorial techniques are concerned (Section 3), they can still somehow be time-consuming depending on the malware code to classify. The need for an important storage space when working with binomial and trinomial set vectors may also have an impact of the detection engine performances (mainly the computing time required for analysis). In case of a desktop computer, the user may not accept to wait more than a few seconds before he can access his data or resources. It is then better to use it upstream on a gateway, which would be dedicated to malware analysis and would check all the flow of incoming data.

As far as the size of the database is concerned, we can mitigate this point by considering that computer hard drives can store nowadays huge amounts of bytes. They also are large live memory (RAM) size. But it is still always unpleasant for the final user to let his antivirus software to be too much resource-consuming. Future work will consequently aim at reducing the database size by using suitable combinatorial designs [4]. The key approach lies in the ability to concentrate the information inside combinatorial design blocks while exhibiting correlation between IAT functions at a far higher order. We estimate that it is thus possible to reduce the database size at least by 75% without lessening the final detection performances.

Another future work deals with optimizing the detection with respect to binomial and trinomial vector-based detection. By adding or removing a few well-known combinations, it is possible to reduce the size of the database and the computing time.

The last work is to consider a limit for the Import Address Table size. In a few cases, malware writers are trying to fool the work performed by antivirus engines. As an example, they try to increase the malware size by loading and using too much external functions. It should then be rather easy to classify malware that are using more than a limited number of external functions but which actually only need and use less.

In the other hand, they may use a few stealth techniques to load and to use external functions without linking them in the Import Address Table. The improvement would then be to consider a file with a too small or too big Import Address Table as a malware.

Author details

Eric Filiol

Address all correspondence to: filiol@esiea.fr

ESIEA – Operational Cryptology and Virology Lab (C+V), Laval, France

References

[1] Bruschi D, Martignoni L, Monga M. Using code normalization for fighting self-mutating malware. IEEE Security and Privacy. 2007. 5:2:46–54. DOI: 10.1109/MSP.2007.31.

[2] Bilar D. Opcodes as predicator for malware. International Journal of Electronic Security and Digital Forensics. 2007;1:2:156–168. DOI: 10.1504/IJESDF.2007.016865.

[3] Borello J M. Study of computer virus metamorphism: modelling, design and detection [thesis]. Rennes: Université de Rennes; 2011.

[4] Colbourn C J, Dinitz J H. Handbook of combonatorial designs. 2nd ed. New York: Chapman et Hall/CRC Press; 2006. 1016 p. ISBN: 1-58488-506-8.

[5] DAVFI projet homepage [Internet]. 2012. Available from: http://davfi.fr/index_en.html [Accessed 2017-01-20].

[6] David B, Filiol E, Gallienne K. Heuristic and proactive IAT/EAT-based detection module of unknown malware. In: Proceedings of the European Conference on Information Warfare and Security (ECCWS) 2016, 7–8 July 2016; Reading. UK: ACPI; 2016. pp. 84–93.

[7] David B, Filiol E, Gallienne K. Structural analysis of binary executable headers for malware detection optimization. Journal in Computer Virology and Hacking Techniques. DOI: 10.1007/s11416-016-0274-2.

[8] Ferrand O, Filiol E. Combinatorial detection of malware by IAT discrimination. Journal in Computer Virology and Hacking Techniques, Special Issue on Knowledge-based System and Security, Roy Park Editor. 2016; 12:131. DOI: 10.1007/s11416-015-0257-8.

[9] Dechaux J, Filiol E. Proactive defense against malicious documents. Formalization, implementation and case studies. Journal in Computer Virology and Hacking Techniques, Special Issue on Knowledge-based System and Security, Roy Park Editor. 2016; 12:191. DOI: 10.1007/s11416-015-0259-6.

[10] Filiol E. Computer viruses: from theory to applications. New York: Springer; 2005. 424 p. DOI 10.1007/2-287-28099-5.

[11] Freund Y, Schapire, R E. A decision-theoretic generalization of on-line learning and an application to boosting. Journal of computer and Systems. 1997; 55:119. DOI: 10.1006/jcss.1997.1504.

[12] Gheorghescu M. An automated virus classification system. In: Proceedings of Virus Bulletin Conference (VB'05). 5–7 October 2005; Dublin. Abington: Virus Bulletin; 2005. pp. 294–300.

[13] GNU Foundation. The GNU multiprecision library [Internet]. Available from: https://gmplib.org [Accessed: 2017-01-12].

[14] Griffin K, Schneider S, Hu X, Chiueh T. Automatic generation of string signatures for malware detection. In: Proceedings of the Recent Advances in Intrusion Detection (RAID'09); 23–25 September 2009; Saint-Malo, France. New York: Springer; 2009; pp. 101–120.

[15] Hastie T, Tibshirani R, Friedman S. The elements of statistical learning: data mining, inference and prediction. New York: Springer Verlag; 2009. 745 p. DOI: 10.1007/978-0-387-84858-7.

[16] Iczelion. Win32 assembly. Tutorial 6: import table [Internet]. Available from: http://win32assembly.programminghorizon.com/pe-tut6.html [Accessed: 2017-01-08].

[17] Maloof M A. Machine learning and data mining for computer security. New York: Springer Verlag; 2006. 210 p. DOI: 10.1007/1-84628-253-5.

[18] Microsoft MSDN [Internet]. Exporting from a DLL using DEF files [Internet]. Available from: http://msdn.microsoft.com/fr-fr/library/d91k01sh.aspx [Accessed: 2017-01-08].

[19] Microsoft MSDN. Loadlibrary function. [Internet]. Available from: https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx [Accessed: 2017-01-08].

[20] Microsoft MSDN [Internet]. Getprocaddress function. Available from: https://msdn.microsoft.com/en-us/library/windows/desktop/ms683212(v=vs.85).aspx [Accessed: 2017-01-08].

[21] Microsoft.Microsoft PE and COFF Specification [Internet]. Available from: https://fr.scribd.com/document/54172253/PE-COFF-Specification-v8-2 [Accessed: 2017-01-08].

[22] Perdisci R, Lanzi A, Lee, W. McBoost, Boosting scalability in malware collection and analysis using statistical classification of executables. In: IEEE Annual Computer Security Applications Conference (ACSAC); 8–12 December 2008; Anaheim. New York: IEEE;2008; pp. 301–310.

[23] peStudio: malware initial assessment. Available from: https://winitor.com/[Accessed: 2017-1-13].

[24] Pietrek M. An in-depth look into the Win32 portable executable file format, part 2 [Internet]. Available from: http://msdn.microsoft.com/en-us/magazine/cc301808.aspx [Accessed: 2017-01-08].

[25] Rajaraman A, Ullman J D. Mining of massive datasets. Cambridge: Cambridge Univeristy Press. 476 p. DOI: 10.1017/cbo9781139058452.

[26] Sung A H, Xu J, Chavez P, Mukkamala S. Static analyzer of vicious executables. In: IEEE Annual Computer Security Applications Conference (ACSAC); 6–10 December 2004; Tucson. New York: IEEE; 2004; pp. 326–334.

[27] Virus total. Available from: https://www.virustotal.com/[Accessed: 2017-01-06].

[28] Wicherski G. peHash: a novel approach to fast malware clustering. In: Proceedings of the 2nd Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET'09); 21 April 2009; Boston. Berkeley: Usenix Association. pp. 1–1.

[29] Williams G J, Simoff S J, editors. Data mining—Theory, methodology, techniques and applications, 2nd edition. New York: Springer Verlag. 275 p. DOI: 10.1007/11677437.

IntechOpen