HAL
open science

Function classification for the retro-engineering of malwares

Guillaume Bonfante, Julien Oury–Nogues

To cite this version:

Guillaume Bonfante, Julien Oury–Nogues. Function classification for the retro-engineering of malwares. 9th International Symposium Foundations and Practice of Security, Oct 2016, Quebec, Canada. hal-03178819

HAL Id: hal-03178819

https://inria.hal.science/hal-03178819v1

Submitted on 24 Mar 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Function classification for the retro-engineering of malwares

Guillaume Bonfante Julien Oury- -Nogues

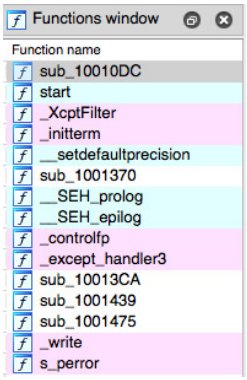
Carbone team
Lorraine University - CNRS - LORIA
Nancy - France

Abstract. In the past ten years, our team has developed a method called morphological analysis that deals with malware detection. Morphological analysis focuses on algorithms. Here, we want to identify programs through their functions, and more precisely with the intention of those functions. The intention is described as a vector in a high dimensional vector space in the spirit of compositional semantics. We show how to use the intention of functions for their clustering. In a last step, we describe some experiments showing the relevance of the clustering and some of some possible applications for malware identification.

1 Introduction

In this contribution, we are concerned with the retro-engineering of malware. In particular, we think to CERTs (Computer Emergency Response Team) where people must determine the content of some attacks; usually, they are asked to give answers quite quickly. Saving time is not just an option. This task is awfully complicate, and demands some high skills and high education. Indeed, the major part of questions are not computable, and thus only talented people may handle such issues. Since the number of attacks is arising [Sym16], we think there is a strong need to give some help, that is to provide automatic tools that output some insights on what's happening.

To get the outline of the behavior of a program, the import address table (IAT) provides a list of external functions that are called. Those include low level system calls. Naturally, those functions give good hints on the job of the program itself. As a justification of the fact, we recall that the (famous, if not the most) retro-engineering tool called IDA introduces this list on its front page as shown on the right. No doubt about it, we are aware that import tables may be hidden by malware or by packers, in particular in the case of code injection. But actually that enforces our argument: showing your import table is telling who you are.



In this paper, we put the focus on functions and we report some experiment that we made about classification. We show in a second

part how that can be related to malware classification. But before going further, let us come back to the context of this research. We have developed at the High Security Lab in Nancy a detection method called morphological analysis. Let us say few words about it. For deeper explanations, we refer the reader to [BKM09]. So, given some program—no matter if it is a malware or not—, we extract its control flow graph, either by a static analysis or a dynamic one. In the latter case, we use an instruction tracer that is made as stealthy as possible, in particular against anti-debugging techniques or anti-virtualization procedures. In the end, we transform the control flow graph to avoid basic obfuscations and we cut it into small pieces called sites. Sites are used as code signatures : they are identified up to some isomorphism. To sum up, the method focuses on *algorithms*. But algorithms may not reveal the *intention* behind them. Our slogan here is that *functions* can do it. There is nothing new at that point: functional models (recall SADT!) are central to software engineering (e.g. Ross [Ros77]).

Let us push a little bit further this idea of extracting the intention of a function. For the internal functions of a malware (and possibly for sane programs), there are no chances to extract it easily. Indeed, malware are awfully obfuscated, and thus two consequences: first, at the binary level, the code is completely blurred, typically, `call 0x12345678` is replaced by `push eax, jmp 0x12345678` where `eax` points to the current instruction. The replacement can be opposite, that is `jmp 0x12345678` can be replaced by `call 0x12345678, ..., pop eax` with `pop eax` pointing at position `0x12345678`. Second point, malware writters use tricks to hide functions. For instance, function call conventions are not followed: the signature `55 8B EC` corresponding to the standard sequence `push ebp, mov ebp, esp` is no longer operative. Thus, it becomes even difficult to identify correctly functions within a malware code. Finally, (and obviously, but for the sake of the argument, we *must* mention it!), the malware does not come with any documentation nor debugging hints. Thus, extracting the intention of internal functions of malware is difficult, and precisely this is the task of retro-analysts.

But, for the functions that occur within external calls, the situation is quite different. External calls refer to dynamically loaded libraries which—if correctly designed—provide well identified functionalities via functions. Moreover, libraries are—should be—well documented within their API. Furthermore, they are manually written, and for that reason, are one of the best sources of information for our purpose.

The goal of this paper is to show that one can extract meaningful informations from documentation in a retro-engineering perspective. We used MICROSOFT's API documents as a source. We used it because it is correctly[1] formatted and uniform, but the method may be used for other vendors. We also used it since many malware are written for WINDOWS and thus refer to MICROSOFT's system library such as `kernel32.dll` or `msvcr100.dll`.

There is one objection we want to address. One may argue that external function calls are usually heavily hidden by malware obfuscations, thus it would be

[1] but not fully!

difficult to identify those functions via (for instance) import address tables. Second point, malware authors include statically some functions within the malware code so that they are not imported. This contribution is not about function identification. For that, we have shown that morphological analysis is able to do it, even in a hostile code. For instance, we established the correspondence between functions of two famous malware REGIN and QWERTY in [BMS15]. We recall that REGIN has been considered as one of the most sophisticated malware, see the nice report by Kaczmarek [Kac15]. In conclusion, we think that the problem is solved by combining morphological analysis and function analysis.

Our contribution has three facets. First, we associate to each function its *intention*. To do that, we use the idea of Vector Space Semantics (VSS) coming from Natural Language Processing. Each function will be associated to some vector representing its semantics. We built an IDA-plugin that shows this mapping. Second, we propose some *clustering* algorithms. Indeed, depending on the depth of the analysis, the retro-analyst may not need[2] to distinguish functions such as `calloc` or `malloc`, both dealing with memory allocation. Thus the idea of clusters. We propose different versions of clustering procedures and we compare them. In the perspective of VSS, clusters correspond actually to the concepts that generalize the ones of their underlying inhabitants. Third, we show how to relate the function clustering to a *malware detection* procedure. The idea that two programs are close when they use same function is quite common, especially for the ANDROID OS (e.g. [PZ13]). We show that clusters are even better. Doing so, we justify the relevance of clusters, and we show them in action.

But, before we enter into technical details, we would like to make a reference to the paper by Teh and Stewart in [TS12] who mention that there are good malware detectors based on multi-layer perceptron whose inputs are features extracted from executable file. However, these tools work too much as black boxes and they do not bring human readable evidences. They conclude that these tools are not that good for retro-analysis. What we do is precisely to establish a human readable correspondence between programs and functionalities.

In Section 2, we present how we built an initial database. It contains informations coming from MICROSOFT's documentation. On these, we had to run some specific tools that extract the "meaning" of the function, that is a weighted vector of words. These are based on Natural Language Processing libraries. In Section 3, we present our clustering algorithm with three variations on function distances. We compare the three classification procedures with standard classification measures and we discuss pro and cons of each measure. In Section 4, we relate the clustering to some external evaluation. We use MICROSOFT's classification and some customized tests.

At the time of the conference, we will publish all python scripts that we used all along, and our rough databases are available on `https://github.com/JulienOuryNogues/DataBase-Function-Microsoft/`.

[2] or must not.

2 The semantics of functions

We describe the procedure that maps functions within libraries to their intention. This is done in three steps: first, we associate libraries to functions, then, we associate functions to documentation and finally, we extract the intention of the function, that is, its semantics.

2.1 Extracting function names from libraries

As stated in the introduction, we worked with DLL coming from microsoft libraries. Those are the most used, and certainly the most interesting since they are directly used for communications with the Operating System.

Our database is built from the 2481 DLL which are coming with the installation disc of our Windows distribution (Vista 64bits). A rapid comparison with other WINDOWS distribution shows that results/conclusions should not be very different. For each DLL, using the python library PEFILE, we get a table partly shown in Figure 1:

Ordinal	RVA	Name
1	0x0001E688h	DllCanUnloadNow
2	0x0001E7B8h	DllGetClassObject
3	0x0001E67Eh	DllInstall
4	0x0001E642h	DllMain
5	0x0004C9A9h	DllRegisterServer
6	0x0004C9A9h	DllUnregisterServer

Fig. 1: The first 6 functions within accessibilitycpl.dll

This table contains the list of functions exported by the DLL. Each function is given by its Ordinal Number, its Relative Virtual Address and its name as should be referred by calling executables. Actually, according to MICROSOFT's policy, names are optional, only Ordinals are mandatory. We keep only those lines with a Name. Second, some Names follow a mangling format corresponding to Visual C/C++. Typically, we read :

```
??0IndexOutOfRangeException@UnBCL@@QEAA@PEAVString@1@PEAVException@1@@Z
```

In which case, our heuristics is to take the first alphanumeric string occuring within the sentence (with a PYTHON filter of the shape `r'\d*(\w*)'`). In the end, we got 50306 distinct names out of the DLLs (34964 directly, 15342 via 'demangling').

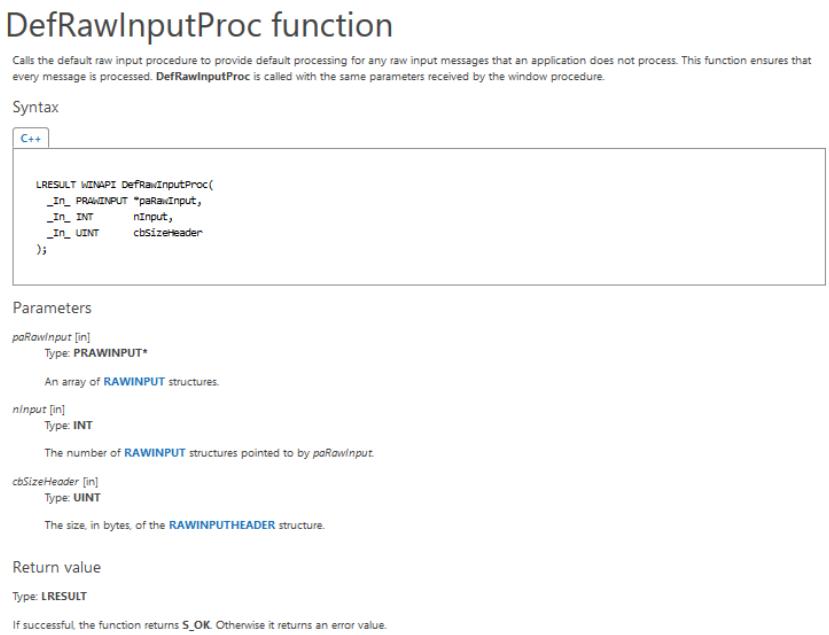
2.2 Extracting function documentation

The documentation of functions has been extracted from MICROSOFT website. We used two different ways to do it. First, we took the root of the WINDOWS API INDEX documentation that is stored at `https://msdn.microsoft.com/fr-fr/library/windows/desktop/ff818516(v=vs.85).aspx`. We visited the

site from this rooting node, and we filtered pages corresponding to function documentation. Doing so, we got a source page for 6155 functions.

Compared to the 50306 functions mentioned above, that is clearly not enough. To complete it, we did some requests of the shape `https://social.msdn.microsoft.com/search/en-US/windows?query=` on MICROSOFT search tool. On the 44151 remaining functions, we performed the requests. Sometimes, we got more than one answer, typically for functions with close names (`printf` vs `wprintf`), in which case, we took them all! In the end, we got 24149 inputs on the total. We denote by $\mathcal{F}$ the set of functions with documentation.

Now, let us come to the typical content of a page that is presented in Figure 2.



DefRawInputProc function

Calls the default raw input procedure to provide default processing for any raw input messages that an application does not process. This function ensures that every message is processed. **DefRawInputProc** is called with the same parameters received by the window procedure.

Syntax

C++

```
LRESULT WINAPI DefRawInputProc(
  _In_ PRAWINPUT *paRawInput,
  _In_ INT       nInput,
  _In_ UINT      cbSizeHeader
);
```

Parameters

paRawInput [in]
Type: **PRAWINPUT***

An array of **RAWINPUT** structures.

nInput [in]
Type: **INT**

The number of **RAWINPUT** structures pointed to by *paRawInput*.

cbSizeHeader [in]
Type: **UINT**

The size, in bytes, of the **RAWINPUTHEADER** structure.

Return value

Type: **LRESULT**

If successful, the function returns **S_OK**. Otherwise it returns an error value.

Fig. 2: DefRawInputProc's description

Thus, we get for each function,

- its name,
- its short description,
- its profile
- typed arguments and their description
- return value and its type.

The WINDOWS API tree describes 211 "categories/key words" among which 155 correspond to functions (versus structures). We appended within our database this category for those functions that could be properly situated within the tree.

We crossed these informations with those extracted from the PEFILE as seen in previous section, and, all in all, we have a table whose entries contain: a name,

a relative virtual address, a DLL name, a WINDOWS distribution name, a short description, a profile, arguments and return value description and a WINDOWS category. The database is accessible for research purposes on our website `https://github.com/JulienOuryNogues/DataBase-Function-Microsoft`.

2.3 Data preparation, some natural language tools

Once we have a complete database, we want to perform some function classification. Indeed, for instance, the reverse-engineer may work at some abstract level for which there are no good reasons to discriminate `wprintf` from `printf`, both dealing with printing. The clustering is supposed to cope that intuition.

Function descriptions are written in English/American English, and thus in a natural language. Our classification purpose algorithm relies on what is called vector space models of meaning [Sch98] (or more abstractly distributional semantics) that has shown to be very powerful those last years by Copestake and Herbelot [CH16] or by Abramsky and Sadrzadeh [AS14]. The rise of distributional semantics is due to the fact that the method requires large amount of textual data for its learning process, and these amounts are now available. The key idea behind the model can be summed up as follows: two words are close if they occur in same contexts. Dogs and cats eat, are stroked and sleep. In some way, the concept of a pet arises from this proximity. In that paradigm, the meaning of a word (and contexts, and clusters) is represented by a vector in a high dimensional vector space whose basis is built on words themselves.

The main problem with the approach is that some words blurs the co-occurence relation that is underlying. Typically, stop words which play a grammatical role without bringing some particular concepts: "with", "to", "the", and so on. We have a first stage that removes them. To do that we use the NLTK python library (Natural Language Tool Kit [Bir15]). In a first step, we apply a part-of-speech tagger that associate to each word its category (e.g. noun, verb, adjective, determinant, etc). Then, we keep only verbs, nouns and adjective.

For instance, `printf`'s description is that it *submits a custom shader message to the information queue*. Applying the Part-Of-Speech Tagging, we get:

```
[('submits', 'NNS'), ('a', 'DT'), ('custom', 'NN'), ('shader', 'NN'),
('message', 'NN'), ('to', 'TO'), ('the', 'DT'), ('information', 'NN'),
('queue', 'NN'), ('.', '.')]
```

And then,

```
[('submit', 'NNS'), ('custom', 'NN'), ('shader', 'NN'),
('message', 'NN'), ('information', 'NN'), ('queue', 'NN')]
```

Second step of our process deals with lemmatization. It is well known that some words have many inflectional forms: verbs (am, are, is, be) and nouns (singular or plural). For our purpose, there is no reasons to distinguish "submits" from "submit". Lemmatization aims to associate to each word in a sentence its *lemma*, that is the "core" word. The task is not that easy and we used the tool

provided by NLTK. Applying lemmatization on our example, we get for the verb "submits": `('submits', 'NNS', 'submit')`.

In a third step, we remove some specific words that are so common that they bring more noise to the discrimination procedure than they bring informations. The complete list is `["none", "be", "specify", "function", "DLL"]`.

We end the process by forgetting all decorations. The resulting vector on our current example is:

```
{'submits':1, 'custom':1, 'shader':1, 'message':1, 'information':1,
'queue':1}
```

The database is available on the website `https://github.com/JulienOuryNogues/DataBase-Function-Microsoft`.

3 Function classification

In this section, we present some clustering methods for functions. The rough idea is to avoid dubious distinctions, between several forms of `printf` for instance. In MICROSOFT documentation, there is already a clustering of functions. Actually, functions are gathered within a tree structure that can be used for our purpose. However, it is not sufficient, for at least two reasons. First, only a third of functions occur within the tree (around 6100 over 20000). And second, MICROSOFT's classification is not used by other libraries. We prefer to have a direct method.

Generally speaking, we apply a standard classification algorithm, the k-mean to function descriptions. The main point is then to define a proper distance between functions. We propose three definitions and we compare them.

Given a word w and a function $\mathtt{f} \in \mathcal{F}$, we denote by $\mathrm{n}(w, \mathtt{f})$ the number of occurrences of w within the (formatted as above) description of the function $\mathtt{f}$. Each function $\mathtt{f}$ is transformed into a vector $\mathbf{v}_{\mathtt{f}}$ as follows:

$$\mathbf{v}_{\mathtt{f}} = \sum_{w \in \mathcal{W}} \mathrm{n}(w, \mathtt{f}) \overrightarrow{w}$$

where $\mathcal{W}$ denotes the set of english words and the family $(\overrightarrow{w})_{w \in \mathcal{W}}$ defines the (orthogonal) basis of our vector space.

Definition 1 (μ-measure). *Given two functions* $\mathtt{f}$ *and* $\mathtt{g}$*, we define* $\mu(\mathtt{f}, \mathtt{g}) = \dfrac{|\mathbf{v}_{\mathtt{f}} - \mathbf{v}_{\mathtt{g}}|}{|\mathbf{v}_{\mathtt{f}}| + |\mathbf{v}_{\mathtt{g}}|}$ *where* $|\mathbf{v}|$ *denotes the euclidian norm of the vector* $\mathbf{v}$*.*

The second measure we use is known as Levensthein distance. Given two words u, v on some alphabet Σ, the Levensthein measure is the number of characters that must be removed or added to the first word to reach the second one. It is denoted by $\delta(u, v)$ in the sequel. For some function $\mathtt{f}$, $\mathrm{Name}_{\mathtt{f}}$ denotes its name (as to be opposed to its description).

Definition 2 (δ-measure). *Given two functions* $\mathtt{f}$ *and* $\mathtt{g}$*, we define* $\delta(\mathtt{f}, \mathtt{g}) = \delta(\mathrm{Name}_{\mathtt{f}}, \mathrm{Name}_{\mathtt{g}})$*.*

There is a slight abuse in notation, δ being used twice, but the context should be clear.

Finally, we introduce a variant of the μ-measure that is obtained by weighting words according to their relative frequency. This is known as TF-IDF (Term Frequency-Inverse Document Frequency). The intention of this measure is to decrease the weight of words that occur in a majority of documents. In the present case, each function description is considered to be a document and thus, $idf(w) = \log\left(\frac{|\mathcal{F}|}{|\{\mathtt{f} \in \mathcal{F} \mid \mathrm{n}(w, \mathtt{f}) \neq 0\}|}\right)$. The weight of a word $w \in \mathcal{W}$ in a function $\mathtt{f} \in \mathcal{F}$ is then defined as $\omega(w, \mathtt{f}) = \mathrm{n}(w, \mathtt{f}) \times idf(w)$. From that, we define the vector $\mathbf{v'}_{\mathtt{f}} = \sum_{w \in \mathcal{W}} \omega(w, \mathtt{f})\overrightarrow{w}$. And, correspondingly, we propose:

Definition 3 (μ'-measure). *Given two functions* $\mathtt{f}$ *and* $\mathtt{g}$*, we define* $\mu'(\mathtt{f}, \mathtt{g}) = \frac{|\mathbf{v'}_{\mathtt{f}} - \mathbf{v'}_{\mathtt{g}}|}{|\mathbf{v'}_{\mathtt{f}}| + |\mathbf{v'}_{\mathtt{g}}|}$.

3.1 Clustering

In this section, we compare the influence of the three different measures on function clustering. We denote by d any measure among $\{\mu, \delta, \mu'\}$. We write $d(\mathtt{f}, S) = \min\{d(\mathtt{f}, \mathtt{g}) \mid \mathtt{g} \in S\}$ given $\mathtt{f} \in \mathcal{F}$ and $S \subseteq \mathcal{F}$.

We use the standard k-mean algorithm. Nevertheless, we want to make three observations. So, the algorithm is as follow:

```
def cluster(F, k): #F is the set of functions, k is the parameter
  P = choose(F, k) # P chooses a list of k initial sets
  end = False
  while(not end): # (loop 1)
    nP = [set() for i in range(k)] # nP is the next P, k empty sets
    for f in F: # (loop 2)
      nP.add(argmin ([ d(f, P[i]) for i in range(k)]))
    end = nP == P
    P = nP
```

First, we begin with a choice of the first k representative based on a density argument. This choice is important since it will modify the number of times we perform loop 1. More technically, we compute $\overline{d}(\mathtt{f}) = \frac{1}{|\mathcal{F}|}\sum_{\mathtt{g} \in \mathcal{F}} d(\mathtt{f}, \mathtt{g})$ for each function $\mathtt{f}$, we order that list in decreasing order. Then, we choose the (an approximation of the) largest value m such that we can find k representative $\mathcal{F}_k = \{\mathtt{f}_1, \ldots, \mathtt{f}_k\}$ such that $d(\mathtt{f}, \mathcal{F}_k) \leq m$ for any function $\mathtt{f}$ and for all $1 \leq i \leq k$, $\overline{d}(\mathtt{f}_i) = \max\{\overline{d}(\mathtt{f}) \mid d(\mathtt{f}, \mathtt{f}_i) \leq m\}$.

In the algorithm, we have to compute the distance $\mathtt{d}(\mathtt{f}, \mathtt{P}[\mathtt{i}])$ from a function $\mathtt{f}$ to some cluster $\mathtt{P}$, that is $\overline{d}(\mathtt{f}) = \frac{\sum_{\mathtt{g} \in \mathtt{P}} d(\mathtt{f}, \mathtt{g})}{|\mathtt{P}|}$. For μ and μ', we compute first

the mean vector $\mathbf{v}_{\mathtt{P}} = \sum_{\mathtt{f}\in\mathtt{P}} \mathbf{v}_{\mathtt{f}}/|\mathtt{P}|$ so that loop (2) costs $k \times n$ with $n = |\mathcal{F}|$. For Levenshtein measure, there is no mean words, so that you have to compute each sum separately. The cost is then (of the order) n^2.

3.2 Levenshtein versus vectors

With a first observation and k set to 1600 (to be compared to the 150's of window), we observe a similarity between Levenshtein's clustering and μ-clustering. In Figure 3, we present a cluster obtained by μ-measure. Each line gives a function name followed by its corresponding vector. One observes that their names are close, it corresponds to a Levenshtein cluster!

```
waveOutMessage ,{'waveoutmessage': 1,'driver': 1,'send': 1,'device': 1,'output': 1,'message': ,'waveform-audio': 1}
waveOutGetPitch ,{'retrieve': 1,'current': 1,'setting': 1,'waveoutgetpitch': 1,'pitch': 1,'device': 1,'output': 1,'waveform-audio': 1}
waveOutSetPlaybackRate,{'set': 1,'device': 1,'rate': 1,'playback': 1,'waveoutsetplaybackrate': 1,'output': 1,'waveform-audio': 1}
waveOutRestart ,{'resume': 1,'paused': 1,'device': 1,'playback': 1,'waveoutrestart': 1, 'output': 1,'waveform-audio': 1}
waveOutPrepareHeader ,{'prepare': 1,'waveoutprepareheader': 1,'playback': 1,'waveform-audio': 1,'data': 1,'block': 1}
waveOutClose ,{'give': 1,'waveoutclose': 1,'output': 1,'device': 1,'close': 1,'waveform-audio': 1}
waveOutSetPitch ,{'set': 1,'device': 1,'pitch': 1,'waveoutsetpitch': 1,'output': 1,'waveform-audio': 1}
waveOutOpen ,{'waveoutopen': 1,'give': 1,'playback': 1,'device': 1,'output': 1,'waveform-audio': 1,'open': 1}
waveOutWrite ,{'give': 1,'send': 1,'waveoutwrite': 1,'device': 1,'output': 1,'waveform-audio': 1,'data': 1,'block': 1}
waveOutGetVolume ,{'volume': 1,'retrieve': 1,'level': 1,'current': 1,'device': 1,'output': 1,'waveform-audio': 1}
waveOutSetVolume ,{'waveoutsetvolume': 1,'set': 1,'level': 1,'volume': 1,'device': 1,'output': 1,'waveform-audio': 1}
waveOutGetPosition ,{'retrieve': 1,'give': 1,'current': 1,'playback': 1,'output': 1,'device': 1,'position': 1,'waveform-audio': 1}
```

Fig. 3: One of the categories, $k = 1600$

3.3 Experimental protocol

Let us validate this observation. We use three similarity indices. They show different aspects of the similarities, see [Qué12] for an in depth discussion. We suppose we are given two partitions (not necessarily with same value k), $(P_i)_{i=1..n}$ and $(Q_j)_{j=1..m}$ of the set $\mathcal{F}$ of size N. Set $n_{i,j} = |P_i \cap Q_j|$, $n_{i,.} = |P_i|$ and $n_{.,j} = |Q_j|$, then, we define: $a = \sum_{i,j} \binom{n_{i,j}}{2}$, $b = \sum_{i} \binom{n_{i,.}}{2} - \sum_{i,j} \binom{n_{i,j}}{2}$, $c = \sum_{j} \binom{n_{.,j}}{2} - \sum_{i,j} \binom{n_{i,j}}{2}$ and $d = \binom{N}{2} + \sum_{i,j} \binom{n_{i,j}}{2} - \sum_{j} \binom{n_{.,j}}{2} - \sum_{i} \binom{n_{i,.}}{2}$.

The three similarity measures we use are Rand index: $R = \dfrac{a+d}{a+b+c+d}$, Jaccard's index, $J = \dfrac{a}{a+b+c}$ and Dice's index $D = \dfrac{2a}{2a+b+c}$. Rand's index evaluates in which way two partitions agree for pairwise elements (whether they are similar or not), Jaccard's evaluates only similarities and Dice's strengthen similarities. These are the symmetric forms. If the partition $(P_i)_{i=1..n}$ is finer than $(Q_j)_{j=1..m}$, that is if $n < m$, one uses the non symmetric versions: $\tilde{R} = \dfrac{a+d+c}{a+b+c+d}$, $\tilde{J} = \dfrac{a+c}{a+b+c}$ and $\tilde{D} = \dfrac{2a+c}{2a+b+c}$ which avoids the fact that a partition in Q_i that would be perfectly split within $(P_j)_j$, that is $Q_i = \cup_{\ell=i_1,\ldots,i_k} P_\ell$, is wrongly evaluated.

4 External validation

Up to now, we worked without any references to other forms of evaluation. We do it in three different ways. The first objective is to evaluate the relevance of the clustering process as defined above. The second reason is that we want to justify the value of the parameter k.

4.1 Windows categorization

WINDOWS provides its own categories. We want to compare these with our own tool. We sum up our results on the following plot:

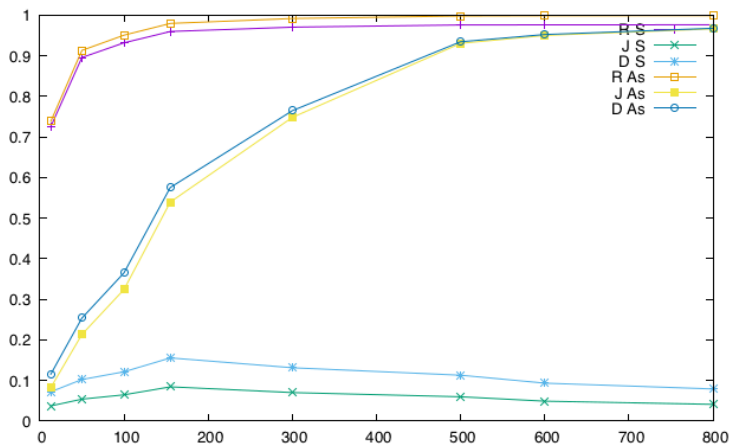


Fig. 4: Window vs μ

There are several observation that we want to make. First, Dice and Jaccard indices are closed. Thus, there are are no distance distortion.

The figure shows that the clustering reaches a maximum for the symmetric Jaccard distance when the two partitions have same parameter k and the value is not very high—close to 10%–, but significant (compared to some random clusters). Differences between WINDOW'S classification and our is due to the fact that some functions use specific vocabulary. For instance, `GetProcAddress` and `LoadLibrary` which are within the category "`DLL`" have no common words in their vectors:

```
GetProcAddress, {'library': 1, 'address': 1, 'retrieve': 1, 'dynamic-link': 1, 'specified': 1, 'variable': 1, 'dll': 1, 'export': 1}
LoadLibrary, {'cause': 1, 'specified': 2, 'call': 1, 'address': 1, 'load': 2, 'space': 1, 'module': 3, 'process': 1, 'other': 1}
```

Third, observe that we get a good refinement of WINDOW's classification. We reach a value of 96.6% for $k = 800$ for the asymmetric Jaccard distance. Finally, on our motivating example, we get the cluster (for k=1600): `fwprintf_s`, `wprintf_s`, `wscanf_s`, `sscanf_s`, `fscanf_s`, `fwscanf_s`, `swscanf_s`, `scanf_s`, `fprintf_s`, `printf_s` , `fread_s`, `scanf`, `wscanf`, `wprintf`, `printf`.

4.2 Do similar programs share similar functions?

We built a database from 25 programs "grouped" in 5 categories. The first four are video players, browsers, archivers and text editors, the last one is made of MICROSOFT OFFICE's main applications. For each category, we chose the most common softwares.

Given a function clustering, $\mathcal{F} = \cup_{i=1}^{k} \mathcal{F}_i$, we define the homomorphism $\phi : \mathbb{R}^{\mathcal{F}} \to \mathbb{R}^k$ by $\phi(\mathtt{f}) = e_j$ where $\{e_1, \ldots, e_k\}$ is an orthogonal basis[3] of $\mathbb{R}^k$ and j is the (unique) index such that $\mathtt{f} \in \mathcal{F}_j$. From the homomorphism, one defines the cluster distance between two vectors $\mathbf{v}$ and $\mathbf{v}'$ in $\mathbb{R}^{\mathcal{F}}$ to be $\Delta(\mathbf{v}, \mathbf{v}') = |\phi(\mathbf{v}) - \phi(\mathbf{v}')|$. In other words, applied to functions, this is the euclidian distance up to the clustering. Notice that given the definition of the homomorphism, given two functions $\mathtt{f}$ and $\mathtt{g}$, we have the inequality: $\Delta(\mathbf{v}_{\mathtt{f}}, \mathbf{v}_{\mathtt{g}}) \leq |\mathbf{v}_{\mathtt{f}} - \mathbf{v}_{\mathtt{g}}|$. With clusters, the world is smaller–in terms of dimensions– and more dense–functions being close.

From that definition, one can measure the distance between two programs:

Definition 4. *Given two programs* $\mathtt{p}_1$ *and* $\mathtt{p}_2$ *importing respectively functions* $\mathtt{f}_1, \ldots, \mathtt{f}_k$ *and* $\mathtt{g}_1, \ldots, \mathtt{g}_m$*, we define* $\mathbf{v}_{\mathtt{p}_1} = \mathbf{v}_{\mathtt{f}_1} + \cdots + \mathbf{v}_{\mathtt{f}_k}$ *and* $\mathbf{v}_{\mathtt{p}_2} = \mathbf{v}_{\mathtt{g}_1} + \cdots + \mathbf{v}_{\mathtt{g}_m}$*. Then, the cluster distance between programs is* $\Delta(\mathtt{p}_1, \mathtt{p}_2) = \Delta(\mathbf{v}_{\mathtt{p}_1}, \mathbf{v}_{\mathtt{p}_2})$*. And the normalized distance is* $\tilde{\Delta}(\mathtt{p}_1, \mathtt{p}_2) = \dfrac{\Delta(\mathtt{p}_1, \mathtt{p}_2)}{\Delta(\mathbf{v}_{\mathtt{p}_1}, \mathbf{0}) + \Delta(\mathbf{v}_{\mathtt{p}_2}, \mathbf{0})}$ *where* $\mathbf{0}$ *is the null vector within* $\mathbb{R}^k$*.*

nom	IZArc2Go	WinRAR	peazip	WINZIP32	7zFM	notepad++	notepad	gvim	EditPad	PSPa	iexplore	firefox	chrome	opera	Safari	QuickTi	iTunes	vlc	realplay	wmplayer	mspaint	WINWO	POWERP	EXCE	wordpad
IZArc2Go	0.0	0.16	0.13	0.18	0.27	0.18	0.35	0.21	0.08	0.07	0.64	0.74	0.39	0.38	0.75	0.73	0.58	0.62	0.45	0.61	0.26	0.74	0.79	0.27	0.25
WinRAR	0.16	0.0	0.18	0.22	0.24	0.15	0.31	0.17	0.19	0.19	0.58	0.7	0.38	0.36	0.71	0.7	0.49	0.6	0.41	0.58	0.25	0.69	0.74	0.27	0.25
peazip	0.13	0.18	0.0	0.23	0.29	0.19	0.41	0.23	0.14	0.13	0.65	0.71	0.42	0.38	0.73	0.75	0.59	0.65	0.47	0.62	0.27	0.73	0.78	0.33	0.27
WINZIP32	0.18	0.22	0.23	0.0	0.36	0.26	0.42	0.23	0.17	0.18	0.68	0.76	0.36	0.38	0.78	0.77	0.62	0.67	0.49	0.66	0.3	0.78	0.82	0.24	0.29
7zFM	0.27	0.24	0.29	0.36	0.0	0.29	0.32	0.32	0.3	0.29	0.45	0.6	0.45	0.4	0.65	0.62	0.52	0.47	0.41	0.52	0.33	0.61	0.66	0.37	0.31
notepad++	0.18	0.15	0.19	0.26	0.29	0.0	0.28	0.2	0.21	0.18	0.6	0.72	0.38	0.37	0.71	0.71	0.48	0.6	0.44	0.57	0.32	0.72	0.78	0.29	0.3
notepad	0.35	0.31	0.41	0.42	0.32	0.28	0.0	0.3	0.39	0.37	0.44	0.63	0.48	0.51	0.61	0.6	0.47	0.45	0.41	0.54	0.32	0.55	0.6	0.38	0.31
gvim	0.21	0.17	0.23	0.23	0.32	0.2	0.3	0.0	0.24	0.22	0.59	0.7	0.35	0.36	0.69	0.68	0.5	0.6	0.42	0.55	0.3	0.71	0.76	0.32	0.3
EditPadLite7	0.08	0.19	0.14	0.17	0.3	0.21	0.39	0.24	0.0	0.08	0.66	0.73	0.4	0.39	0.75	0.74	0.61	0.64	0.46	0.64	0.27	0.75	0.8	0.23	0.25
PSPad	0.07	0.19	0.13	0.18	0.29	0.18	0.37	0.22	0.08	0.0	0.65	0.74	0.39	0.37	0.76	0.73	0.6	0.62	0.45	0.62	0.25	0.74	0.79	0.25	0.25
iexplore	0.64	0.58	0.65	0.68	0.45	0.6	0.44	0.59	0.66	0.65	0.0	0.38	0.54	0.56	0.53	0.41	0.43	0.3	0.41	0.4	0.48	0.36	0.42	0.59	0.48
firefox	0.74	0.7	0.71	0.76	0.6	0.72	0.63	0.7	0.73	0.74	0.38	0.0	0.61	0.65	0.26	0.27	0.51	0.34	0.46	0.64	0.64	0.31	0.37	0.7	0.62
chrome	0.39	0.38	0.42	0.36	0.45	0.38	0.48	0.35	0.4	0.39	0.54	0.61	0.0	0.2	0.66	0.65	0.42	0.54	0.36	0.55	0.48	0.67	0.73	0.42	0.45
opera	0.38	0.36	0.38	0.38	0.4	0.37	0.51	0.36	0.39	0.37	0.56	0.65	0.2	0.0	0.65	0.64	0.39	0.57	0.35	0.51	0.47	0.67	0.73	0.46	0.45
Safari	0.75	0.71	0.73	0.78	0.65	0.71	0.61	0.69	0.75	0.76	0.53	0.26	0.66	0.65	0.0	0.22	0.47	0.42	0.5	0.63	0.7	0.3	0.29	0.75	0.65
QuickTimePlayer	0.73	0.7	0.75	0.77	0.62	0.71	0.6	0.68	0.74	0.73	0.41	0.27	0.65	0.64	0.22	0.0	0.45	0.42	0.52	0.55	0.66	0.21	0.34	0.73	0.64
iTunes	0.58	0.49	0.59	0.62	0.52	0.48	0.47	0.5	0.61	0.6	0.43	0.51	0.42	0.39	0.47	0.45	0.0	0.42	0.47	0.5	0.58	0.43	0.52	0.64	0.54
vlc	0.62	0.6	0.65	0.67	0.47	0.6	0.45	0.6	0.64	0.62	0.3	0.34	0.54	0.57	0.42	0.42	0.42	0.0	0.35	0.52	0.53	0.41	0.44	0.61	0.51
realplay	0.45	0.41	0.47	0.49	0.41	0.44	0.41	0.42	0.46	0.45	0.41	0.46	0.36	0.35	0.5	0.52	0.47	0.35	0.0	0.5	0.46	0.54	0.59	0.45	0.4
wmplayer	0.61	0.58	0.62	0.66	0.52	0.57	0.54	0.55	0.64	0.62	0.4	0.64	0.55	0.51	0.63	0.55	0.5	0.52	0.5	0.0	0.6	0.61	0.67	0.66	0.56
mspaint	0.26	0.25	0.27	0.3	0.33	0.32	0.32	0.3	0.27	0.25	0.48	0.64	0.48	0.47	0.7	0.66	0.58	0.53	0.46	0.6	0.0	0.64	0.69	0.27	0.12
WINWORD	0.74	0.69	0.73	0.78	0.61	0.72	0.55	0.71	0.75	0.74	0.36	0.31	0.67	0.67	0.3	0.21	0.43	0.41	0.54	0.61	0.64	0.0	0.1	0.72	0.63
POWERPNT	0.79	0.74	0.78	0.82	0.66	0.78	0.6	0.76	0.8	0.79	0.42	0.37	0.73	0.73	0.29	0.34	0.52	0.44	0.59	0.67	0.69	0.1	0.0	0.76	0.68
EXCEL	0.27	0.27	0.33	0.24	0.37	0.29	0.38	0.32	0.23	0.25	0.59	0.7	0.42	0.46	0.75	0.73	0.64	0.61	0.45	0.66	0.27	0.72	0.76	0.0	0.27
wordpad	0.25	0.25	0.27	0.29	0.31	0.3	0.31	0.3	0.25	0.25	0.48	0.62	0.45	0.45	0.65	0.64	0.54	0.51	0.4	0.56	0.12	0.63	0.68	0.27	0.0

Fig. 5: Program distance. Clustering with $k = 400$

In the table above, we computed the normalized distance between the applications of our database. Two observations. First, if one takes the closest programs (outside itself!), the result is not surprising: `firefox` is close to `safari`, `gvim` to `notepad` and `safari` to `quicktime`. These relationships differ, either applications have same purpose, or they share development.

In a second step, we use the correlation matrix above to perform some clustering for the distances in Figure 7. For that sake, we used the k-mean algorithm with $k = 6$, that is 5 categories plus one for trash. We get the following result:

[3]Being unique up to isomorphism, the definition does not depend on this choice.

Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6
izarc2go	winrar	chrome	firefox	iexplore	notepad
peazip	winzip32	opera	safari	winword	mspaint
notepad++	7zFM	iTunes	quicktime	powerpnt	wordpad
gvim	editpad7	realplay	vlc		
pspad	excel	wmplayer			

The table shows that proximity is explained either by close functionalities, or by a close (past or current) development process. However, we can conclude that a such a classification remains quite imprecise. For retro-engineering, this is not really problematic since the analyst would cope errors, but we could not use it for detection for which false positive ratio must be low.

Finally, to show the role of function clustering, we worked directly with the direct distance between programs $\Delta'(\mathtt{p}_1, \mathtt{p}_2) = |\mathbf{v}_{\mathtt{p}_1} - \mathbf{v}_{\mathtt{p}_2}|$. The program clustering is not as good as above.

Indeed, if we do not modify the convergence parameter (which corresponds to some cluster distance), the algorithm converges to only one category. So, we have to put a looser parameter (from 0.45 to 0.6) to get again some "reasonable" clusters. We get the following table. One observes a big category still emerges. It is actually the trash category (those who can't be compared to any others). Note also that the other categories are not as relevant as above.

Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6
winrar	izarc2go	firefox	quicktime	mspaint	iexplore
winzip32	peazip	safari	vlc	wordpad	wmplayer
7zFM	gvim	realplay			winword
notepad++	editpad7				powerpnt
notepad	pspad				
chrome	excel				
opera					
iTunes					

4.3 Packer identification

One may refine the preceding experiment as follows. The distance we define on programs does not take into account the order in which functions are used, or if they are used once or several times. The run distance defined below takes this into consideration. Characterizing programs using function sequence as signatures is known in the literature as behavioral detection (see for instance [JDF08]). Usually, results are not very good due to the high ambiguity (with respect to function sequence) of program behaviors (e.g. [BGM10]). But, what we do here is much more modest. We just want to identify some very particular behaviors, those coming from a small set of packers.

Given a program $\mathtt{p}$ calling some functions $\mathtt{f}_1, \ldots, \mathtt{f}_k$, let us run $\mathtt{p}$ on some inputs, one gets a sequence $w \in \{\mathtt{f}_1, \ldots, \mathtt{f}_k\}^*$ of the functions called along the

computation. We define $\mathcal{R}$ to be the set of all inputs, and for $r \in \mathcal{R}$, we define $w_{\mathtt{p},r}$ to be the sequence of the called functions of the program $\mathtt{p}$ on inputs r. For the sake of the argument, we will restrict runs to be finite, so are $w_{\mathtt{p},r}$ for all $r \in \mathcal{R}$.

Let us suppose given a clustering $\mathcal{F} = \cup_{i=1}^{k} \mathcal{F}_i$. for all $f \in \mathcal{F}$, $\gamma(\mathtt{f})$ denotes its cluster, that is a number within $\{1, \ldots, k\}$. The definition extends to sequences: $\gamma(\mathtt{f}_1, \ldots, \mathtt{f}_m) = \gamma(\mathtt{f}_1) \cdots \gamma(\mathtt{f}_m) \in \{1, \ldots, k\}^m$.

Definition 5. *Given two programs* $\mathtt{p}_1$ *and* $\mathtt{p}_2$*, we define their run distance to be* $\overline{\delta}(\mathtt{p}_1, \mathtt{p}_2) = E(r \mapsto \delta(\gamma(w_{\mathtt{p}_1,r}), \gamma(w_{\mathtt{p}_2,r})))$*, that is the expectation of the distances of the runs, the words in* $\{1, \ldots, k\}^*$ *being compared with respect to Levenshtein distance.*

There are infinitely many runs, so that it is hard to get the distance between two programs, actually undecidable. However, for packer identification, one may observe that packers are almost insensitive to inputs outside some self-protection mechanisms. This is for instance what is done by Calvet in [Cal10]. Thus, we will approximate run distance by the distance on one run.

	calc.exe	freecell.exe	mystic-calc.exe	mystic-freecell.exe	telock98-calc.exe	telock98-freecell.exe	telock982-calc.exe
calc.exe	0,00	0.46	0.98	0.35	0.98	0.98	0.12
freecell.exe	0.46	0,00	0.97	0.2	0.97	0.95	0.45
mystic-calc.exe	0.98	0.97	0,00	0.98	0.43	0.53	0.98
mystic-freecell.exe	0.35	0.2	0.98	0,00	0.98	0.96	0.34
telock98-calc.exe	0.98	0.97	0.43	0.98	0,00	0.33	0.98
telock98-freecell.exe	0.98	0.95	0.53	0.96	0.33	0,00	0.97
telock982-calc.exe	0.12	0.45	0.98	0.34	0.98	0.97	0,00

Fig. 6: Program distance

If one uses the rough run distance, one gets a correlation matrix that reveal the similarity between the original code and its packed form. Thus, to identify packers, one use the C-prefix run distance for some $C \in \mathbb{N}$, that is $\overline{\delta}_p(\mathtt{p}_1, \mathtt{p}_2) = \min_{i>C}(E(r \mapsto \delta(\gamma(w_{\mathtt{p}_1,r})[0..i], \gamma(w_{\mathtt{p}_2,r})[0..i])))$. We take $C = 10$. The result is:

nom	calc.exe	freecell.exe	mystic-calc.exe	mystic-freecell.exe	telock98-calc.exe	telock98-freecell.exe	telock982-calc.exe
calc.exe	0,00	0.05	0.36	0.43	0.22	0.25	0.22
freecell.exe	0.05	0,00	0.39	0.36	0.22	0.25	0.22
mystic-calc.exe	0.36	0.39	0,00	0.02	0.5	0.42	0.5
mystic-freecell.exe	0.43	0.36	0.02	0,00	0.5	0.42	0.5
telock98-calc.exe	0.22	0.22	0.5	0.5	0,00	0.07	0.04
telock98-freecell.exe	0.25	0.25	0.42	0.42	0.07	0,00	0.07
telock982-calc.exe	0.22	0.22	0.5	0.5	0.04	0.07	0,00

Fig. 7: Program distance

Then, packers are correctly identified. Notice that for `telock`, we used different options (`98` or `982`) and for the rough distance, the distance were high. Not anymore with the prefix distance.

4.4 Combining morphological analysis and Function clustering

Let us come back to our broad objective. In our research group, we are developing morphological analysis (MA) that is used for malware identification. We recall

that MA belongs to the branch of detecting method based on abstract control flow graph identification, see [BMM06] for an other example. In other words, it puts into light underlying algorithms, not functions. So, our question was the following: is there a relationship between functions and morphological analysis? Or to put it in a more precise setting: are there some specific algorithms before calling functions?

So, we came back to our application database and we applied morphological analysis. The learning process went well. To sum up, we had at least 73 specific sites (some remarkable graphs within applications). The main difference between applications is whether they are stand alone or if they are based on dynamic libraries. In the first case, we have lots of sites, in the latter one, much less.

Then, as we did above, we computed a correlation matrix that is displayed below.

	IZArc2Go	winrar	peazip	winzip32	7zfm	notepad++	notepad	gvim	editpad	pspad	iexplore	firefox	chrome	opera	Safari	quicktime	iTunes	vlc	realplay	wmplayer	mspaint	winword	powerpnt	excel	wordpad
IZArc2Go	0.00	9.89		9.57		9.04		10.96	4.48	2.45			9.94				6.74							9.86	
winrar	9.89	0.00		5.31		4.92		4.79		9.19			5.43	7.25			1.77							9.19	
peazip			0.00																						
winzip32	9.57	5.31		0.00		6.16		5.65		9.50			3.74	1.39			3.90			5.61	9.30			10.00	
7zFM					0.00																				
notepad++	9.04	4.92		6.16		0.00		2.45		9.26			3.91	5.42			5.35							9.55	
notepad							0.00				3.14										1.05				
gvim	10.96	4.79		5.65		2.45		0.00		10.60			3.79	4.73			5.35			5.61				10.70	
editpad	4.48								0.00	6.06														7.24	
pspad	2.45	9.19		9.50		9.26		10.60	6.06	0.00			9.25				6.04							10.20	
iexplore							3.14				0.00										2.33				
firefox												0.00			2.34	2.63			3.23						
chrome	9.94	5.43		3.74		3.91		3.79		9.25			0.00	3.26			3.05				9.30			9.25	
opera		7.25		1.39		5.42		4.73					3.26	0.00			4.03								
Safari												2.34			0.00	1.89			2.34						
quicktime												2.63			1.89	0.00			2.18						
iTunes	6.74	1.77		3.90		5.35		5.35		6.04			3.05	4.03			0.00							6.04	
vlc																		0.00							
realplay												3.23			2.34	2.18			0.00						
wmplayer				5.61				5.61												0.00					
mspaint				9.30			1.05				2.33		9.30								0.00				3.28
winword																						0.00	0.97	1.05	
powerpnt																						0.97	0.00	0.97	
excel	9.86	9.19		10.00		9.55		10.70	7.24	10.20			9.25				6.04					1.05	0.97	0.00	
wordpad																					3.28				0.00

Some clusters occur. They are:

```
[{'winword', 'vlc', 'powerpnt', '7zFM', 'excel', 'peazip'}, {'notepad', 'wordpad'},
{'realplay', 'Safari', 'firefox', 'Quicktime'}, {'iexplore', 'mspaint'},
{'iTunes', 'chrome', 'IZArc2Go', 'pspad', 'winzip32', 'wmplayer', 'editpad',
 'gvim', 'opera', 'notepad++', 'winrar'}]
```

that brings back some of the clusters that we have seen above. For instance, we see the link between `firefox`, `safari` and `quicktime`. No surprises, there are differences since we are using a very different mechanism. But, a manual verification on common sites showed us that algorithmic relationships correspond to function relationship (around one third of the sites). In other words, the function analysis could be used to enrich our own methodology.

5 Conclusion

We propose a mapping from functions to their intentions via a vector of words within a natural language. It is based presently on the function documentation provided by MICROSOFT, but it could be extended to any vendors.

In a second step, we discuss the question of function clustering, the idea being to avoid dubious distinction. The clustering may be performed at different levels, depending on the expected precision.

In a third step, we relate the function clustering to other issues. We compare it with respect to MICROSOFT's own clustering. Then, we work on program identification and packer identification. In a last step we compare it to our morphological analysis.

Finally, we provide on our web-page a plugins for IDA that maps functions to their vectors, or alternatively to the url of each function. The plugins is available on our git repository.

As a perspective, we would like to explore a little bit further the natural language aspect of our approach. For instance, we did not relate words one to an other with respect to their own semantics. We think that this could strengthen the semantics of functions even more. An other idea is to look for informations in a much broader way: there are tons of tutorials, technical explanations and code samples on the web. Machine Learning techniques could be applied to these data (that could be inspired by the work of Lakhotia et al [LL15] or Tawbi et al. [SSM+16]).

Acknowledgment. The authors would like to thank Jean-Yves Marion and Mizuhito Ogawa for early discussions and Fabrice Sabatier and Alexis Lartigue for discussions and some experiments.

References

AS14. Samson Abramsky and Mehrnoosh Sadrzadeh. Semantic unification - A sheaf theoretic approach to natural language. In Claudia Casadio, Bob Coecke, Michael Moortgat, and Philip Scott, editors, *Categories and Types in Logic, Language, and Physics - Essays Dedicated to Jim Lambek on the Occasion of His 90th Birthday*, volume 8222 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2014.

BGM10. Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. *Behavior Abstraction in Malware Analysis*, pages 168–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

Bir15. Steven Bird. NLTK Documentation. 2015.

BKM09. Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Architecture of a Morphological Malware Detector. *Journal in Computer Virology*, 5(3):263–270, 2009.

BMM06. Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In Roland Büschkes and Pavel Laskov, editors, *Detection of Intrusions and Malware & Vulnerability Assessment: Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006. Proceedings*, pages 129–143, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

BMS15. Guillaume Bonfante, Jean-Yves Marion, and Fabrice Sabatier. Gorille sniffs code similarities, the case study of Qwerty versus Regin. In Fernando Colon Osorio, editor, *Malware Conference*, page 8, Fajardo, Puerto Rico, October 2015. IEEE.

Cal10. Joan Calvet. Tripoux: Reverse-engineering of malware packers for dummies. In *DeepSec 2010*, 2010.

CH16. Ann Copestake and Aurélie Herbelot. Lexicalised compositionality, 2016.

JDF08. Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4(3):251–266, 2008.

Kac15. Matthieu Kaczmarek. Malware Instrumentation Application to Regin Analysis. In Eric Freyssinet, editor, *Malware Conference*, page 16, Paris, France, November 2015.

LL15. Charles LeDoux and Arun Lakhotia. Malware and machine learning. In *Intelligent Methods for Cyber Warfare*, volume 563 of *Studies in Computational Intelligence*, pages 1–42. Springer, 2015.

PZ13. Naser Peiravian and Xingquan Zhu. Machine learning for android malware detection using permission and api calls. In *Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, ICTAI '13, pages 300–305, Washington, DC, USA, 2013. IEEE Computer Society.

Qué12. Romain Quéré. *Some proposals for comparison of soft partitions*. Phd, Université de La Rochelle, December 2012.

Ros77. Douglas T. Ross. Structured Analysis (SA): A language for communicating ideas. *IEEE Transactions on Software Engineering*, 3(16), 1977.

Sch98. H. Schuetze. Automatic word sense discrimination. *Computational Linguistics*, 1(24):97–123, 1998.

SSM+16. Mina Sheikhalishahi, Andrea Saracino, Mohamed Mejri, Nadia Tawbi, and Fabio Martinelli. Fast and effective clustering of spam emails based on structural similarity. In Joaquin Garcia-Alfaro, Evangelos Kranakis, and Guillaume Bonfante, editors, *Foundations and Practice of Security: 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015*, pages 195–211. Springer International Publishing, 2016.

Sym16. Symantec. 2016 Internet Security Threat Report, 2016.

TS12. Anselm Teh and Arran Stewart. Human-Readable Real-Time Classifications of Malicious Executables. In *10th Australian Information Security Management Conference*, 2012.