ApiScout: Robust Windows API Usage Recovery for Malware Characterization and Similarity Analysis

Daniel Plohmann[1], Steffen Enders[2], Elmar Padilla[1]
[1] Fraunhofer FKIE, [2] TU Dortmund

Abstract

Given today's masses of malware there is a need for fast analysis and comparison of samples. System API usage has been proven to be a very valuable source of information for this e.g. shown by Rieck et al. [1]. However, the majority of malware samples is shipped packed, making it hard to get accurate information on their payload's API usage. Today's state of the art to get this information from packed samples is by unpacking them or dumping memory with subsequent reconstruction of imports using tools like ImpREC and Scylla. This has several drawbacks since it is a manual procedure requiring a live process environment and suffers from inaccuracy due to missed dynamic imports.
In this paper, we present ApiScout, a fully automated method to recover API usage information from memory dumps. It does not require a live process environment and is capable of handling dynamic imports leading to more accurate results compared to existing approaches. ApiScout is a two-staged approach. The first stage is a preparation step creating a database of candidate offsets for API functions. In the second step we crawl through a given memory dump of a process and match all possible DWORDs and QWORDs against this database yielding us API reference candidates. We filter and enrich candidates using different procedures leading us to the desired API usage information.
Based on this information, our second contribution in this paper is a concept called ApiVectors. It efficiently stores the information extracted by ApiScout. This enables fast assessment of a malware's potential capabilities and allows similarity analysis of API usage across samples. For the latter the methods imphash and impfuzzy are the *de facto* standard. However, they both suffer from inaccuracy due to exclusively relying on the import table and non-recoverability of input data. In our approach we use Jaccard and Tanimoto similarity to compare ApiVectors, leading to a much higher accuracy.
Our third contribution is an extensive analysis of API usage across 589 malware families of the Malpedia dataset. The families combined use only about 4500 APIs that can be grouped into 12 semantic groups. The analysis further proves the functionality of ApiScout and shows that ApiVectors clearly outperform imphash and impfuzzy.

Keywords: malware analysis, malware classification, Windows API, visualization

1 Introduction

Even with continuous advances and growing tool support, in-depth malware analysis and reverse engineering in general remain a tedious, primarily manually executed task. Among the most important steps during analysis are the rapid identification of the family for a given malware sample, e.g. to allow incorporation to existing analysis results, and the localization and semantic annotation of relevant regions in the binary that most likely contain points of interest, such as code related to persistence, communication, or specific malicious capabilities. Especially for the latter, an analyst typically depends on orientation along prominent cornerstones, which are first and foremost interactions with the operating system, e.g. with the Windows API or system calls on unix-based systems.

Therefore, it is highly important during the preprocessing for in-depth analysis (i.e. typically during unpacking or directly after) to annotate and recover this system API information to consequently allow effective analysis of the program's interactions with this system interface. As generic unpacking is still far from being solved [2], dumping memory is a common practical take to circumvent packers in an approximating way. However, this oftentimes implies rebuilding a binary from a memory dump of a mapped process image, ensuring correct section boundaries and applying special care for its import table. Recent research [3] has shown that almost 50% of 382 investigated individual Windows malware families make use of dynamic imports that are not reflected by the given PE header's structures. Because these dynamic imports are often not in proximity of the regular import address table or are even stored in custom structures, they are regularly missed by available state of the art tools for import reconstruction such as ImpREC and Scylla [4]. Additionally, these tools require that the process from which the memory dump was taken and for which the import information shall be reconstructed is still alive and running, making their application in automated analysis complicated.

We propose a generalization to the API recovery method presented by Sharif et al. [5] in context of their Eureka framework that we call ApiScout. ApiScout (after a one-time setup) works completely independent from a running system and is especially suited for the automated post-processing of memory dumps. It is capable of recovering both import table based and cached dynamic Windows API references, and transferring this information directly into popular analysis tools such as IDA Pro [6], effectively removing the need for accurately rebuilding binaries from memory dumps.

In addition, we propose a second concept called ApiVectors which efficiently stores the results extracted with ApiScout, i.e. presence of references to Windows API functions. As this information may allow an analyst to derive a first assessment of a malware's potential capabilities, we also provide a compact visualization method for these vectors to support this activity. We furthermore introduce a method to compare ApiVectors in order to measure the similarity of API usage across binaries. Currently, there are two *de facto* standard methods for import table similarity analysis: `ImpHash` [7] and `ImpFuzzy` [8]. We extensively evaluate our method ApiVectors and show that it outperforms both of these methods.

In summary, we make the following contributions:

- We present ApiScout, a method for robustly recovering Windows API usage information from memory dumps that can capture both import table based and cached dynamic imports, requiring no running system or disassembly.
- We conduct an extensive analysis of Windows API usage patterns across 589 malware families, analyzing the occurrence frequency and grouping around 4,500 Windows API functions into semantic context groups.
- We introduce ApiVectors, a method to efficiently store and compare ApiScout results, and show that the matching capabilities of ApiVectors outperform both of the *de facto* standards `ImpHash` and `ImpFuzzy`.
- We publish a production-ready library that allows easy integration of ApiScout and ApiVectors into existing malware analysis workflows and provide a reference set of ApiVectors for more than 700 malware families to be used for future malware classification.

The remainder of this document is structured as follows. In Section 2, we describe the concepts for both ApiScout and ApiVectors in detail. In the course, we provide information about the evolution of the Windows API. Section 3 provides a detailed evaluation of these concepts using the Malepdia data set. We derive the best parameter combination for ApiVectors and investigate the general capabilities and limitations of Windows API usage information in the context of similarity analysis and classification of malware. Section 4 gives an overview of related work while Section 5 concludes the paper.

2 Approach

The Windows API is a programmer's primary interface for interacting with Windows on system level. This holds true for malicious software in the same way as for benign software. Because it is hard if not almost impossible to avoid using the Windows API for most invasive tasks, it bears high relevance for program analysis. In fact, it is one of the essential cornerstones for orientation in reverse engineering and thus for inferring the behavior of code. Luckily, the Windows API is a well-documented and stable structure tied to the NT kernel, which is organized in layers of abstraction. Programming using the Windows API will typically lead to the creation of an Import and/or Delay Import Table (IT) in the compiled binary program's PE header. These tables are effectively the blueprint for the Import Address Table (IAT) which ultimately accumulates the references to the Windows API.

Apart from using packers, malware often tries to conceal its usage of the Windows API [3]. Besides easily reconstructable static imports (using the IAT), cached dynamic imports resolved during runtime appear as frequently as in 45% of analyzed malware families, while another 5% of families make use of fully obfuscated API usage. Currently available tools require a running system environment and the process information it provides and yet are barely able to reliably recover all information (especially cached dynamic imports), which highlights the need for new methods.

In this section, we describe our approach ApiScout in detail. ApiScout is a technique that makes use of trivially obtainable domain knowledge in order to accurately extract references to the Windows API as found in arbitrary mapped program images such as memory dumps. Our method is a significant generalization of the approach presented by Sharif et al. [5] in the context of their unpacking framework Eureka. Opposite to their work, our method is capable of locating API references in absence of other analysis methods such as disassembly and call analysis, which potentially leads to better coverage while being easily applicable in isolation. In a second part, we introduce ApiVectors, a concept to efficiently store the data extracted using ApiScout. Apart from outlining how ApiVectors are composed, we also explain how they can be compared to measure similarity.

2.1 ApiScout: Extraction of Windows API References

ApiScout aims at easing the task of import reconstruction when dealing with memory dumps. This is a typical situation analysts find themselves in when unpacking malware. The application of this technique is divided into two stages, a mandatory preparation phase which usually only has to be performed once for a given patch state of the operating system and the actual application in the second step.

The first stage is a necessary preparation step in which a database of all exported functions of executables and most importantly Dynamic Link Libraries (DLLs) is created. This is performed by parsing the PE header of files for their preferred ImageBase address as well as all of the exported functions they provide in order to obtain their Relative Virtual Addresses (RVA). Assuming DLLs are mapped at their desired memory location (ImageBase), the expected Virtual Address (VA) of API calls is simply calculated as ImageBase + RVA. Performing system-wide indexation ensures that all information potentially required is available for the second stage.

Although not explicitly explained, we assume that Sharif et al. [5] used a similar technique to dynamically create a database for all DLLs in a given process, recording ImageBase and RVA pairs for all Windows API functions. Should the operating system use Address Space Layout Randomization (ASLR), every DLL is also loaded once in order to initialize their respective randomized global load offsets, which is recorded along the other data. Note that in this case, ASLR load offsets have to be derived after every reboot of the operating system.

However, we found that in typical analysis use cases (manual dynamic analysis or sandboxing) it is sufficient to build the database just once as virtualization and snapshots are frequently used. In our case, the indexation procedure took between 2 and 5 minutes on a virtual machine typically configured for dynamic malware analysis (2 CPUs, 4 GB RAM). To provide an overview of typical database sizes, Table 1 gives an overview of the number of available Windows DLLs and API functions per operating system version (vanilla installation of Professional Edition each). The column "All" lists database results when the whole hard disk is crawled. We noticed that a significant number of DLLs is stored redundantly, which is why we additionally provide the "Unique" column, in which each DLL-API name pair is counted only once. Note that 64bit OS versions of Windows will also feature 32bit versions of most system DLLs to enable the WOW64 compatibility mode, which explains the bigger gap of versions Windows 7 and above versus Windows XP. The total number of unique pairs of DLL and exported function across all of these combined is at 323,851, which means that there are seemingly entries that are unique to specific Windows versions.

		All		Unique	
Name	Version/Build	APIs	DLLs	APIs	DLLs
Win XP	NT5.1/2600	128,408	1,597	101,710	1,584
Win 7	NT6.1/7601	251,186	3,828	168,176	2,215
Win 8.1	NT6.3/9600	282,802	5,154	183,424	3,024
Win 10	NT10.0/17134	338,456	5,971	234,528	3,751
Total				323,851	5,686

Table 1: Number of DLLs and exported API functions found in different vanilla installations of Windows. Windows XP is 32bit, all others are 64bit versions.

The second stage is the actual recovery of Windows API references from memory dumps. We first calculate the expected Virtual Address for every exported function by adding the DLL's ImageBase and export RVA offset, while accounting for a potential additional offset introduced by ASLR.

Having calculated all values, we can cache them in a database against which queries can be performed. Using the Virtual Address as key, it is obviously possible that exported functions from two or more DLLs may map to the same address. We analyzed these potential collisions for all operating system versions shown in Table 1.

For Windows XP we found a single collision and for Windows 7 we identified 178 collisions, with none of them being in critical system DLLs. However to our surprise, we noticed 55,181 and 115,022 collisions for Windows 8.1 and Windows 10 respectively. Our investigation of this circumstance lead to the explanation that the majority of DLLs in these Windows versions is compiled with an ImageBase of 0x10000000 (32bit) or 0x180000000 (64bit) and the system fully relies on dynamic rebasing and ASLR to ensure conflict-free mapping into the memory space. In these cases, the API database has to be built for a specific initialized running instance of the given Windows system (with ASLR per DLL being identical across processes because of Windows' shared memory concept).

Now, in order to actually recover Windows API references for a given memory dump of a process, we now crawl through the whole memory dump and query every possible DWORD and QWORD (to account for 32bit and 64bit systems) in the given memory dump against the database and collect hits as candidates. These candidates are then (optionally) filtered and enriched with information using the following procedures.

First, it is common that references to the Windows API appear in clusters, similar to their native representation (IAT). We can exploit this to remove any hits that do not have another hit in their immediate neighborhood, e.g. using a window of 32 or 64 bytes.

Second, in case the memory dump contains a PE header, we can additionally use the Import and Delay Import Table to check if API references are part of the Import Address Table. On the one hand, we can use this information to confirm crawled candidates as actual hits. On the other hand, we can afterwards compare the API entries described by these structures against all other confirmed candidates in order to derive whether or not the given program makes extensive use of cached dynamic imports.

Third, we can derive an estimate of how many times an API reference is itself used in a `call` or `jmp` instruction. We are aware that full disassembly (e.g. as used in [5]) would enable the entire stack of in-depth analysis methods such as data flow analysis which could provide us with accurate references. However, these methods are typically very expensive in terms of analysis time, hence we decided to instead approximate these code references by limiting ourselves to the two instructions that are most commonly used to interact with Windows API references: `call dword ptr <offset>` and `jmp dword ptr <offset>`.

For x86 machine code, the Opcode bytes for these instructions are `FF15<offset>` and `FF25<offset>` respectively, with `<offset>` being an absolute memory address. Given a memory dump, we typically know the `<base_address>` it was taken from, which in turn allows us to calculate and check if `<offset> - <base_address>` falls within the size of the memory dump and points to one of the API reference candidates. If yes, we can increment our approximated reference counter for that API reference. For x64 machine code, the Opcode bytes for these instructions are `48FF15<offset>` and `48FF25<offset>` respectively, with `<offset>` being a relative offset. The calculation and check can be performed analogously.

While this approach seems naive and one would expect False Positives, we found that it is very accurate in practice. In order to measure ApiScout's accuracy, we used 15 common benign system binaries from the Windows XP SP3 system and produced memory dumps for their corresponding loaded program image. Using their Import and Delay Import Tables as ground truth, there are a total of 5,367 API references to be found in the Import Address Tables of these programs. On this data set, ApiScout achieves an F-Score of 0.991 and 0.995 (with neighbor filtering) respectively, correctly identifying all ground truth entries with zero False Negatives.

We investigated the results manually to analyze the False Positives. For the three programs `explorer.exe`, `mmc.exe`, and `cmd.exe`, 51 False Positives are found. As it turns out, all three programs use `kernel32.dll!GetProcAddress` to dynamically load additional references to API functions during runtime. This easily explains why ApiScout also correctly identified these references which were however not covered in the ground truth.

We used the same set of programs to also test the accuracy of the reference count approximation. For this reason, we used Hex Rays' IDA Pro 6.9 to identify all code references to the offsets of the IAT. IDA identifies a total of 30,928 references to the 5,418 API offsets, while ApiScout identifies 27,640 (89.37%). Through manual inspection we identified that the remainder were almost entirely code references through `call <register>` instructions, which have most likely been generated in order to preserve space when being used multiple times within one function. As explained earlier, it would be possible to derive their call targets using e.g. data flow analysis but this is out of scope for ApiScout as we are only interested in API reference recovery.

In summary, we believe that ApiScout produces very accurate results for the recovery of API reference offsets and acceptable estimates for API reference counts. Both of these information items serve as valuable insight on which further analysis can be built upon.

2.2 ApiVectors: Storage and Comparison of Extraction Results

After having recovered references to the Windows API, it makes sense to persist this information for use in further analysis. While it makes sense to store full ApiScout reports, related works (section 4) show that a compact representation enabling comparisons is favorable. However, these existing methods use hashing to create fingerprints, which incurs loss of source information. With our approach we want to fulfill the requirements that on the one hand the source information is preserved and on the other hand the representation is still compact, ideally consisting of printable symbols only.

In this section, we now introduce our proposal for a storage and comparison format for ApiScout results: ApiVectors. ApiVectors are effectively vectors with a space-optimizing compression that preserve information about the occurrence of references to selected Windows API functions for a given memory dump.

Please note that the concept outlined in the following could be generally applied to other system APIs as well, e.g. to Android's [9] API concept of packages, classes, and permissions or UNIX system calls.

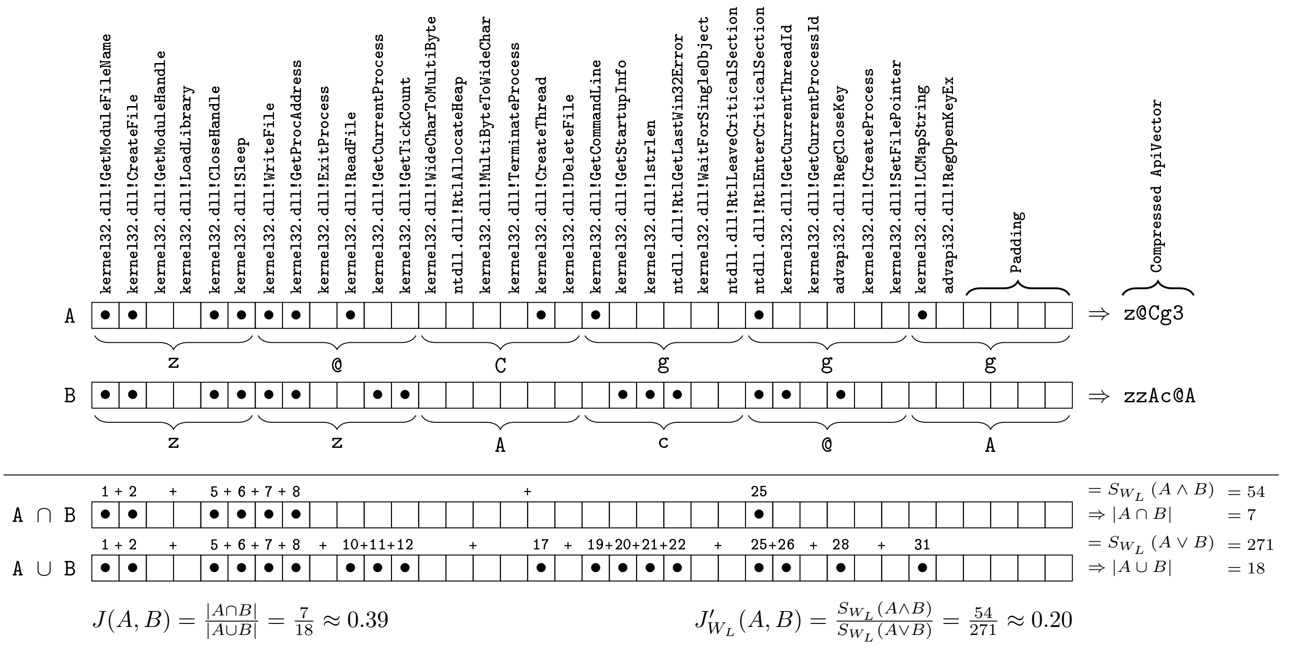

kernel32.dll!GetModuleFileName
kernel32.dll!CreateFile
kernel32.dll!GetModuleHandle
kernel32.dll!LoadLibrary
kernel32.dll!CloseHandle
kernel32.dll!Sleep
kernel32.dll!WriteFile
kernel32.dll!GetProcAddress
kernel32.dll!ExitProcess
kernel32.dll!ReadFile
kernel32.dll!GetCurrentProcess
kernel32.dll!GetTickCount
kernel32.dll!WideCharToMultiByte
ntdll.dll!RtlAllocateHeap
kernel32.dll!MultiByteToWideChar
kernel32.dll!TerminateProcess
kernel32.dll!CreateThread
kernel32.dll!DeleteFile
kernel32.dll!GetCommandLine
kernel32.dll!GetStartupInfo
kernel32.dll!lstrlen
ntdll.dll!RtlGetLastWin32Error
kernel32.dll!WaitForSingleObject
ntdll.dll!RtlLeaveCriticalSection
ntdll.dll!RtlEnterCriticalSection
kernel32.dll!GetCurrentThreadId
kernel32.dll!GetCurrentProcessId
advapi32.dll!RegCloseKey
kernel32.dll!CreateProcess
kernel32.dll!SetFilePointer
kernel32.dll!LCMapString
advapi32.dll!RegOpenKeyEx
Padding
Compressed ApiVector
A ⇒ z@Cg3
z @ C g g g
B ⇒ zzAc@A
z z A c @ A
A ∩ B
1 + 2 + 5 + 6 + 7 + 8 + 25
$= S_{W_L}(A \wedge B) = 54$
$\Rightarrow |A \cap B| = 7$
A ∪ B
1 + 2 + 5 + 6 + 7 + 8 + 10+11+12 + 17 + 19+20+21+22 + 25+26 + 28 + 31
$= S_{W_L}(A \vee B) = 271$
$\Rightarrow |A \cup B| = 18$
$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{7}{18} \approx 0.39$
$J'_{W_L}(A,B) = \frac{S_{W_L}(A \wedge B)}{S_{W_L}(A \vee B)} = \frac{54}{271} \approx 0.20$

Figure 1: A full example for the construction, compression, and similarity calculation of two ApiVectors A and B. The ApiVectorBase has length 32 and contains the most common Windows API functions as found in the Malpedia datatset. Compression is achieved by using Base64 with a custom alphabet (section Table 2) and applying run-length encoding for repetitive symbols. Similarity between the vectors is calculated using Jaccard similarity with optional weights for vector offsets.

Construction

To achieve the goal of compactness while preserving information, we need to systematize and abstract the information to be held in the vectors. Thus, in order to construct an ApiVector, we first apply normalization to all API references as found by ApiScout.

First, we drop the A/W suffix that is found for a range of Windows API functions, indicating if they process ANSI or Unicode strings [10]. We justify this with the fact that these API functions otherwise carry out the same functionality semantically (which is supported by having joint documentation pages in the MSDN).

Second, we unify the different versions of the Microsoft Virtual C Runtime (MSVCRT) into a single DLL name (mscvrt.dll, instead of msvcr80.dll, msvcr90.dll and so forth). This allows us to collect and compare this information even in case the compiler version used for a malware family is changed over time.

Third, we demangle API names [11] and drop all meta information except for the actual function name. This simply cleans up readability for the API names and also in some cases summarizes multiple polymorphic functions into one representation.

After this pre-processing, we perform abstraction introducing a mapping that converts the Windows API references into a vector. While not strictly required, we use powers of two as vector length (as it benefits our later presented visualization concept). The basis for our mapping and hence all ApiVectors is an ordered collection of DLL/API names that will serve as labels for the offsets in the vector, in the following we will refer to this as ApiVectorBase.

The vector entries can now be either bit-sized or byte-sized, depending on whether just the presence or also the reference count should be stored. We refer to them as ApiVector or FrequencyApiVector respectively. In case of the FrequencyApiVector, the count is deliberately capped at 255, allowing us to use a size of one byte per entry. This decision is justified with an observation from [3], where only 12 instances (out of 171,346 references to WinAPI functions in 1208 malware samples) had reference counts exceeding a value 255.

Two example ApiVectors with length 32 are shown in Figure 1. The figure also demonstrates compression and calculation of similarity as explained in the following two sections.

Compression

Plohmann et al. [3] further report that Windows API references in malware (and most likely in software generally) are not evenly distributed. Instead, they appear heavily skewed towards few API functions that are used with much higher frequency than others. Furthermore, assuming generally sparse vectors, we can exploit this knowledge and design our storage method in a way that reduces the required space while it still remains almost loss-less with regard to information in the best case.

In order to optimize storage, we propose the following compression method that can be used to convert ApiVectors into a compact format consisting only of printable characters (achieving the second goal). We first use an approach very similar to Base64 [12], but exchange the default alphabet with the custom alphabet shown in Table 2. To explain, we have replaced the numbers originally contained in Base64 by other printable symbols in order to have them available for run-length encoding, as clarified in the following.

000000	A	011010	a	110100	@	111010	*
000001	B	011011	b	110101	}	111011	/
000010	C	011100	c	110110	]	111100	?
...	...	...	...	110111	^	111101	,
011000	Y	110010	y	111000	+	111110	.
011001	Z	110011	z	111001	-	111111	_

Table 2: The custom Base64 alphabet used for compression of ApiVectors.

We first take the binary representation of an ApiVector and apply zero-padding to achieve a multiple of 6 in length. Then, we pack each subsequent 6 bits into a symbol from our alphabet, analogously to Base64. Next, we use the numbers that have been freed to perform run-length encoding. Repetitions of symbols for more than 2 times are replaced by the symbol suffixed with the number of repetitions, meaning a sequence of e.g. "AAAAAA" becomes "A6", while "AA" remains the same.

This yields our final compressed ApiVector. This representation both preserves information about which Windows API functions are referenced and is also compact and printable.

A full example for the compression of two vectors including application of the run-length encoding is shown in Figure 1. Obviously, the same method can be applied to FrequencyApiVectors. Additionally, using a vector length based on a power of two as recommended earlier allows to unambiguously derive the number of padding bits for every vector longer than 4.

Similarity and Matching

It is safe to assume that different versions of programs that originate from the same code base will likely contain similar usage patterns of the Windows API. Therefore, the next logical step is to look into a suitable method to compare ApiVectors of the same parameterization (i.e. ApiVectorBase etc.) and determine their similarity.

To compare binary vectors, numerous similarity and distance measures have been proposed [13]. For ApiVectors, we want to prioritize common appearance of API references in the vectors to be compared and are not interested in common absent APIs. This already rules out 32 of the 76 techniques presented in [13]. Because many of the remaining negative match exclusive metrics fall into the same hierachical cluster (indicating their correlation), we decide to use the Jaccard similarity [14] as one of the most commonly used and best understood representatives, i.e. for given ApiVectors A and B, the similarity is calculated as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Because A and B are already bit vectors, the calculation can be efficiently implemented through Boolean logic as:

$$J'(A,B) = \frac{S(A \wedge B)}{S(A \vee B)}$$

in which S counts the number of bits set in a given bit vector. This is similar to the method used in [15].

We now additionally introduce the possibility to apply weights to the individual vector offsets. Such weights make sense, because the occurrence frequency of Windows API functions may vary greatly. Thus, the weights can be used e.g. to diminish the influence of very common or boost the influence of rare API functions. With regard to our computation, we extend S in a way that before addition of the vector fields, the Hadamard product (i.e. element-wise multiplication) with a second weight vector W is created and denote this new function as S_W, and the Jaccard similarity using this function as $J'_W(A,B)$. A complete example for such a calculation is shown in Figure 1. It features an ApiVectorBase of length 32 and the weight vector W_L, which uses the position in the vector as weight.

For FrequencyApiVectors, we can no longer use the Jaccard similarity because we neither have sets of API references nor a bit vector but instead individual occurrence counts of API references. Still, we want to preserve the characteristics of the Jaccard similarity. Therefore, we adopt the continuous form of the Tanimoto similarity [16]:

$$T(A,B) = \frac{A \cdot B}{|A|^2 + |B|^2 - A \cdot B}$$

To highlight the similarity to the Jaccard similarity, note that in the case of A and B being bit vectors as in our previous case, $T(A,B)$ will calculate an identical result to $J'(A,B)$. We also introduce the application of weight vectors for $T(A,B)$, yielding $T_W(A,B)$ for weight vector W. However in this case, we have to apply the weight vector directly to A and B before the actual calculation of $T(A,B)$ and again use element-wise multiplication between W and A, B respectively.

3 Evaluation

In this section, we will evaluate the concepts of ApiScout and ApiVectors from different viewpoints. First, we give a short outline of the data set used in all of the following evaluations, which is the Malpedia data set [17]. Next, we present the results of applying ApiScout to this data set, which provides us with some general statistics on the Windows API usage characteristics of malware. After this, we will have a closer look at the possible parameterization of ApiVectors. In this context, we examine to what degree Windows API usage information can be used as a similarity measurement tool for malware classification. Finally, we show that our method of ApiVectors greatly outperform the two current state-of-the-art approaches `ImpHash` and `ImpFuzzy`.

3.1 Dataset

For the following evaluations, we use the Malpedia malware corpus [17]. Malpedia is a manually curated collection of cleanly labelled and unpacked malware. The organization of the corpus has been designed following best practices [18] with numerous contributions through a community of professional malware analysts, researchers, and incident responders.

As the corpus is inventorized in form of a git repository, we can reference the data used in this evaluation as being tied to commit hash `d863e1d`, dated 2018-05-17. In this revision, the corpus contains memory dump data for 673 distinct peer-reviewed Windows malware families, represented by 1,854 samples. These memory dumps originate from two fixed system snapshots (Windows XP and Windows 7) for which the maintainers provided us the respective Windows API databases as required by our approach (section Section 2.1). In all of the following, ApiScout will be configured to use a neighborhood window of 32 bytes for filtering. Additionally, we remove database entries for DLLs with a base address of 0x400000, 0x10000000, and 0x180000000 as we noticed that they are slightly prone to False Positives because their base addresses overlap with the offsets of a significant number of memory dumps in the Malpedia data set. In the lookup database, this affects 20 DLLs (1.3%) with 4,746 API functions (3.7%) for Windows XP and 405 DLLs (10.6%) with 14,912 API functions (5.9%) for Windows 7. Please note that we carefully reviewed all occurrences of hits for these DLLs (958 out of 249,054 hits total, i.e. 0.4%) to ensure that no actual references to the Windows API were removed or highly relevant DLLs were affected.

3.2 Malware Usage Characteristics of the Windows API

The first part of the evaluation we will apply ApiScout to all memory dumps in the data set and examine the results. We will also put this data in relation to the earlier findings reported in [3]. The data set contains also malware that is written in scripting languages and other frameworks that make only indirect use of the Windows API. Because of this, we first have to discard malware families written using the .NET framework because it proxies references to the Windows API in a way that can not be compared to the remainder of families that exist as natively compiled code and directly interact with the Windows API. This affects 84 out of the 673 (12.48%) families with 130 samples, leaving us with 589 families that can still be analyzed using ApiScout.

According to the taxonomy presented in [3], three different Windows API usage styles can be identified that have been defined as follows:

1. Static (i.e. regular) imports using the PE header's import table.
2. Dynamic imports of WinAPI function addresses that are cached within the memory occupied by the malware.
3. Custom import schemes without caching of references that are considered as further obfuscation.

Although the data set has almost doubled in its number of covered malware families since its original publication, the distribution of usage styles has merely changed. Our observation is that the fraction of malware families using static imports exclusively has slightly increased and is still the biggest part with now 51.2%. Families exclusively using dynamic imports make up 19.0% and a combination of static and dynamic imports are used by 25.7%. Consequently, the fraction of families using obfuscation has decreased to 2.9% and 3.9% in combination with one or both of the other methods. In other words, this means that ApiScout can successfully extract Windows API usage information from 96.1% of the given Windows malware families in this data set.

	WinAPI functions	DLLs
Minimum	0.00	0.00
25%	80.00	5.00
50%	115.00	7.50
75%	172.00	10.00
Maximum	706.00	24.00
Average	140.88	7.95
Total Observed	4664.00	62.00

Table 3: Distribution of DLL and API usage as observed through ApiScout extraction. Aggregated over 589 malware families.

For the 589 families, we observe a total of 4,664 unique API functions from 62 DLLs being used. Note that application of the normalization procedure described in Section 2.2 would reduce this number to 4,008 API functions. The characteristics of the distribution over families are shown in Table 3. We can observe that on average, about 140 API functions from 8 DLLs are referenced. Compared to the whole spectrum of the Windows API or even the range of DLLs found being mapped in a running system as described in Section 2.1, this appears to be a rather small number.

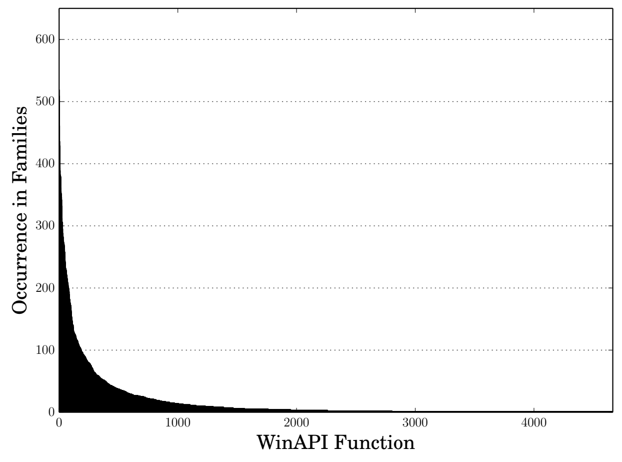


Figure 2: Occurrence frequency of WinAPI functions with regard to number of families they appear in.

Looking at the frequency of occurrence of the individual API functions aggregated over families (Figure 2), only 30 API functions appear in more than 50% of the families while 4332 API functions (92.88%) appear in 10% of the families or less. This lets us conclude that on the one hand, there is a strong set of very common Windows API functions that seems to be essential to (malicious) programs, while on the other hand, there is a significant range of API functions required or available to implement a variety of different capabilities. We think that especially the long tail in this distribution should benefit the formulation of classification methods based on Windows API usage.

3.3 Parameterization of ApiVectors

In this section, we explore how parameterization affects ApiVectors and FrequencyApiVectors, primarily the size and composition of their ApiVectorBase. The aggregated ApiScout extraction results for all samples inventorized in Malpedia as explained in Section 3.2 allow us the following conclusions:

First, the set of Windows API functions even for this wide range of different (unpacked) malware families in this data set with 4664 elements is still manageable. This means we can use a value in this range as upper bound for our vector length. We decide to use 4096 as the gain of using 8192 does not warrant the cost of introducing extensive padding.

Second, the distribution is highly irregular and skewed towards the most common functions and we additionally observe an average number of just 140.88 referenced Windows API functions per sample, which means we can expect mostly sparse vectors (a fact that we already exploited for our compression method of ApiVectors, see Section 2.2). Additionally, this means the most frequently observed Windows API functions are likely relevant to the innerworkings of malware while the less frequent naturally carry more information (according to Shannon entropy [19]). We therefore first study ApiVectorBases composed of the first n most common Windows APIs as extracted by ApiScout.

Analysis of Coverage

First, we will focus on the Windows API function coverage. For obvious reasons, it is in our interest to maximize the coverage to have our vector represent the API functions used in a sample as complete as possible. On the other hand, we have to balance against the expected size in bytes as we want to minimize the meta data overhead.

In Table 4, the coverage of used Windows API functions is shown for a range of different vector sizes. As one can see, the vector of size 4096 expectedly covers nearly all imports of all samples in the Malpedia dataset. Reducing the vector size to 2048 only decrease the mean coverage by about 2%. On the other hand, the smaller vectors of 64 to 256 cover only 33.3% to 67.5% of APIs on average. For a size of 512, the respective ApiVector already covers 82.08% of the identified used Windows API functions, and a vector size of 1024 with a coverage of 92.5% seems the best compromise between size and coverage.

Size	10%	20%	30%	40%	Median	Mean
64	15.2%	20.0%	23.8%	27.2%	31.2%	33.3%
128	25.5%	33.3%	37.9%	43.6%	48.7%	50.7%
256	44.4%	51.3%	57.3%	63.0%	68.9%	67.5%
512	63.6%	72.8%	77.3%	81.8%	85.6%	82.1%
1024	81.5%	89.2%	92.6%	94.4%	96.0%	92.5%
2048	94.1%	97.4%	98.4%	99.1%	99.6%	97.4%
4096	98.9%	100.0%	100.0%	100.0%	100.0%	99.4%
C-1024	69.2%	78.4%	83.2%	86.6%	90.3%	86.7%

Table 4: Percentage of covered WinAPI function for the least covered 10-40% of samples. Results organized by vector size.

	64	128	256	512	1024	2048	4096	C-1024
GUI	0	0	15	69	195	551	1202	27
System	5	16	32	57	108	204	547	150
Execution	25	43	74	132	209	316	464	229
String	7	10	17	32	101	223	315	52
Network	1	14	30	60	100	175	295	192
File	13	22	34	67	104	168	241	114
Device	0	0	1	7	28	79	142	66
Other	0	0	2	8	52	72	117	24
Crypto	0	1	6	20	34	57	115	48
Memory	9	15	28	37	59	86	102	68
Registry	4	5	10	15	19	33	57	32
Time	0	2	7	8	14	25	35	22
Unknown	0	0	0	0	1	59	464	0

Table 5: Breakdown of semantic categories at different vector lengths. Distribution in C-1024 for comparison.

It is important to note that using ApiVectorBases that are composed entirely by frequency of occurrence does not address the semantic context in which these Windows API functions are typically used. Because we strive for the concept of ApiVectors to be useful to the practical work of malware analysts, we decided to review the list of observed Windows API functions and classify them by their typical use cases. We decided to use 12 semantic context categories into which we tried sorting the 4096 most common Windows API functions. We had to leave 464 functions unassigned (mostly undocumented, most likely internally used functions), for which we could not reliably determine their corresponding category. Table 5 gives an overview of the distribution of categories in the different vector lengths, sorted by the number of occurrences in the longest vector.

As can be seen, GUI-related functions take up more than a fourth of the inspected Windows API functions. The reason for this is primarily that interaction with graphical elements is extensively organized per functional element such as Menus, Forms, and Buttons, wheras access to this interface is mostly provided through components and libraries such as GDI [20] and MFC [21]. String manipulation also ranks rather high with 315 identified functions, where many of them are redundant in functionality. The Other category mostly contains API functions related to aspects like mathematical functions, compression, or data structures and Variant datatypes [22].

Interestingly, key functional aspects like memory management and access to the Windows Registry are enabled through few, but very important Windows API functions, allowing malware authors few implementation choices, while other aspects such as network access can be realized down from socket-based operations (e.g. using `ws2_32.dll`) up to more abstracted methods provoding convenience in their use (e.g. with `wininet.dll`).

As stated earlier, we want to balance the information carried in a vector against its size. We therefore used our domain knowledge in reverse engineering and malware analysis to create a custom vector of length 1024 (referred to as `C-1024` throughout this paper). For `C-1024`, we manually balanced the ApiVectorBase by reducing the number of GUI, String, and Other functionality in order to include more Windows API functions connected to more relevant and meaningful categories when investigating malware. This shift in focus allows us to especially capture several indications of interesting behavior that would otherwise be left out. In the following, we will evaluate `C-1024` along the other vector sizes.

Analysis of Storage Utilization

Next, we compare the required storage of the compressed ApiVectors (section Section 2.2) for all samples of the Malpedia dataset. Table 6 gives an overview of the distribution of values for different parameterizations. As expected, the required storage increases notably with growing vector sizes and using ApiFrequencyVectors increases the value drastically. For (binary) ApiVectors, the maximum size of the vectors grows roughly proportionally with the vector size, however the average size increases only marginally, which is again related to the value sparsity in longer vectors.

We have also listed the values for two other established methods of capturing Windows API import characteristics: `ImpHash` and `ImpFuzzy`. `ImpHash` produces MD5 hashes for the import table, while `ImpFuzzy` internally uses SSDeep [23] to capture and compare similarity of import tables. Please note that for both of these methods the original information cannot be extracted, while ApiVectors preserve (most of) this information.

Method	Size	Min	25%	Mean	75%	Max
Binary	64	4	11.0	10.4	12.0	12
	128	4	20.0	19.2	23.0	23
	256	4	32.0	34.1	43.0	44
	512	4	45.0	57.5	79.0	87
	1024	5	52.0	83.8	119.0	172
	2048	5	56.0	103.1	147.0	304
	4096	5	57.0	111.2	156.0	538
	C-1024	5	51.0	74.3	106.5	167
Frequency	64	4	41.0	59.5	82.0	87
	128	5	64.0	102.6	147.0	172
	256	5	93.0	158.9	228.0	329
	512	5	120.5	219.5	309.0	594
	1024	6	138.0	270.8	374.0	1035
	2048	6	147.0	302.1	425.5	1304
	4096	6	148.0	313.1	435.0	1890
	C-1024	6	127.0	240.1	342.0	780
`ImpHash`		32	32	32	32	32
`ImpFuzzy`		3	14.0	54.4	82.0	100

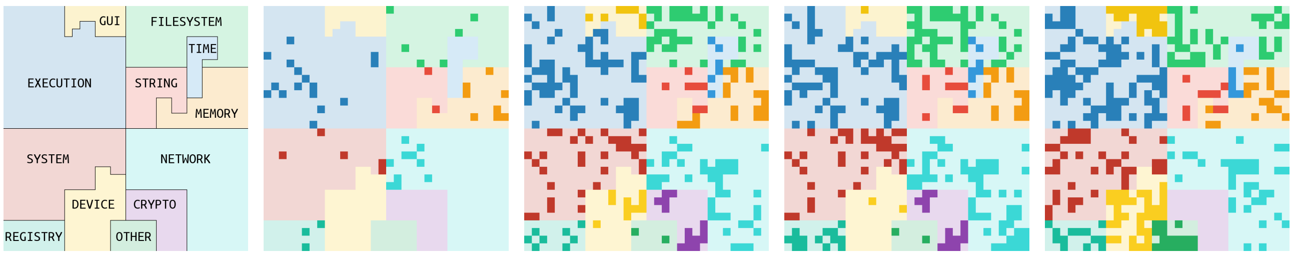
Table 6: Space consumption in bytes for all considered vector lengths in both binary and frequency mode. Sizes for `ImpHash` and `ImpFuzzy` for comparison.

ApiQR: Visualization of ApiVectors

One major side benefit of using vector lengths of a power of two is that every vector that is a square number (such as in the case of `C-1024`) can be neatly fitted in a Hilbert curve [24]. We used this characteristic to create a concise visual summary of an arbitrary ApiVector which we call ApiQR.

For this visualization, we sort the Windows API functions in our `C-1024` ApiVector by their semantic categories instead of occurrence frequency. As a consequence, and due to the nature of a Hilbert curve, functions of the same semantic category are placed spatially close to each other in ApiQR diagrams. We have introduced colors to better set off the different categories. Each cell in the Hilbert curve represents one API function and is colored according to its respective vector entry. In case an entry is present, it is colored solidly, while absent entries are colored with a similar, but faded color. The result is a representation where the presence of each single Windows API function from the reference vector can be directly inspected.

This visualization of ApiVectors allows a rapid first assessment of a malware's potential functional capabilities. Figure 3 shows the ApiQRs of four different malware samples that originate from four different malware families. In the left-most diagram it can be observed in Figure 3b that DELoader has only a very sparse vector which reflects in its ApiQR diagram. We manually analyzed the sample and it consists of only 81 functions in total. Its functionality is limited



(a) Semantic Categories (b) DELoader (c) Zeus (d) Citadel (e) DarkComet

Figure 3: ApiQR visualizations for binary files of four different malware families.

to downloading a payload via HTTP, which is then decrypted using RC4 and integrity-verified using CRC32 (statically linked, which is why it does not show in the Crypto section of the ApiQR) and then finally injecting it into a target process (`explorer.exe`).

Regarding the two middle diagrams, Zeus and Citadel (Figures 3c and 3d), whose source-code is derived from the leak of Zeus, are feature-rich banking trojans and information stealers that both have several hundred functions. First, the similarity in the two ApiQR diagrams is very striking. It can be seen that Citadel is making use of more API functions than Zeus, which is explained by some additions that have been made, functionality-wise.

DarkComet (Figure 3e) on the other hand is a fully-fledged Remote Administration Tool (RAT), which contains a lot of intrusive functionality including capabilities to log or produce keystrokes as well as screen grab the victims Desktop.

An interactive version of ApiQR diagrams has been integrated into Malpedia [17]. We believe that these examples for the application of ApiVectors and ApiQRs qualify the approach as a valuable practical tool for malware analysis.

3.4 Malware Identification

After having had a closer look at the effects of the size of ApiVectors on coverage and storage, we will now evaluate how well they perform when used for malware identification.

As shown in Table 3 and Figure 2, the average number of distinct APIs used per malware family has been measured at just 140, while the occurrence frequency of individual APIs across families is low for the vast majority them. In combination, this means we can likely expect sparse vectors that consist of relatively diverse sets of APIs. Furthermore, we assume that it is likely that API usage patters are even more stable than code since they are the semantic skeleton that describes the overall capabilities of a given malware family. In this light, the approach seems very promising for malware identification because different versions of the same malware family usually already share a great part of their code-base. It also holds true for potential future versions of a given malware.

Now, to compare ApiVectors with each other, we use the distance functions $J'(A,B)$ (derived from the Jaccard distance) and $T(A,B)$ (continuous form of Tanimoto distance) as defined in Section 2.2. To begin with, we will examine the influence of vector size and optional weights on the matching performance in order to find an optimal parameter configuration. Following this, we will compare the performance of our approach with `ImpHash` and `ImpFuzzy`.

Finding an Optimal Parameter Configuration

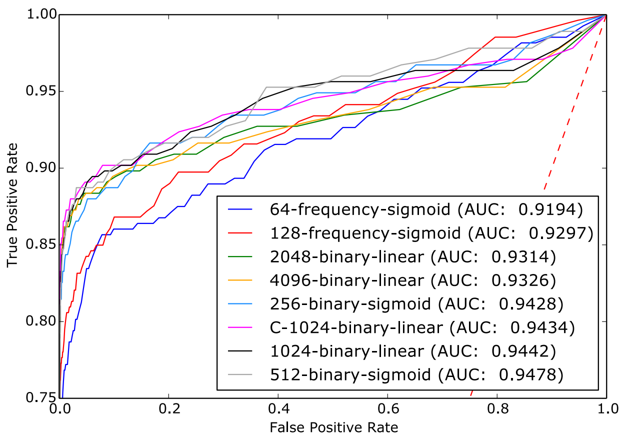
Similar to the analysis presented in Section 3.3, we now examine the influence of two parameters on the matching performance of ApiVectors. For vector sizes, we will again use the range from 64 to 4096 and our self-defined `C-1024`. Additionally, we now have to account for the strongly imbalanced distribution of occurrence frequencies observed in Figure 2. From an information theory point of view [19] the less common API entries carry more information and it could be worthwhile amplifying their signal by introducing weights.

To analyze the influence and usefulness of weights, we investigate three configurations:

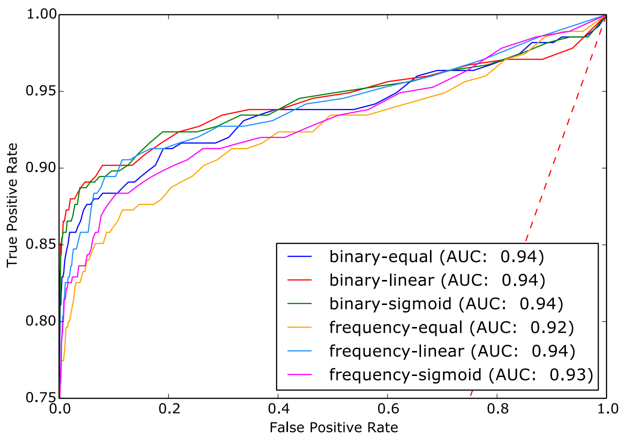
- Equal weights: Every vector offset has equal weight (set to 1).
- Linear weights: Every vector offset is weighted with a linearly increasing value, giving higher importance to API functions with lower occurrence frequency.
- Non-linear weights: We use the following sigmoid function to achieve non-linearly increasing weights: $W_S(i,n) = 50(1+\tanh\left(3\frac{2i-n}{2n}\right))$, with i being the vector offset and n being the vector size. Note that we model the sigmoid using a rescaled $tanh$ function, which in our case returns values in the range of 5 to 95.

While equal weights can be seen as a baseline, linear and non-linear weights are used to measure the impact of shifting weights towards lesser frequent API functions in the vector.

To measure the matching performance, we again resort to using the Malpedia dataset [3]. Because our distance function returns a value between 0.0 and 1.0, we can measure our similarity results against a threshold and evaluate using the Receiver Operating Characteristic (ROC) curve [25], measuring the True Positive Rate (TPR) against the False Positive Rate (FPR). We compare every (non .NET-based) sample (1,724)



(a) Best parameterization per vector length



(b) All parameter results for `C-1024`

Figure 4: Comparison of ROC curves for selected parameter combinations and vector lengths (y-axis rescaled).

against every other. As True Positives we count cases in which a sample of the same family yielded a similarity score above the threshold. As False Positives, we count every case where a sample of another family yielded a similarity score above the threshold. It has to be noted that only for 275 families, more than one sample was available in the dataset and thus only for these matches can be achieved. These families together contain 1505 samples (82.92% of the whole dataset).

The comparative results of the evaluation for different parameters are shown in Figure 4a, with only the best combination of binary/frequency vector and weight selected. We can quickly see that short vectors with 64 and 128 achieve the weakest performance at low FPRs. Interestingly, these are also the only frequency-based vectors among the best-per-class results, meaning that this detailed view adds value in this situation. This is likely due to the fact, that, here, frequency adds information value because only the high-occurrence APIs are considered in the vector. Frequency-based vectors also seem to perform better at very high FPRs, which however are undesirable anyway. For longer vectors, frequency information seems to introduce additional noise that lessens performance at low FPRs. In fact, the vector length gives no significant performance improvement beyond a vector length of 512. Generally, binary vectors of lengths 512 to 1024 with linear or sigmoid weights perform best. In the range of FPRs of less than 10%, the crafted vector `C-1024` performs best, closely followed by the other variants from that range.

In-Depth Analysis of the Results for C-1024

With regard to the influence of weight functions, the data shown in Figure 4b for `C-1024` is also representative for all vectors 512 and longer, which all exhibit very similar diagrams. We observe that adding weight to lesser frequent API functions (i.e. higher offsets in vector) seems indeed to make sense as it improves performance throughout the board (both for linear and sigmoid weights). Frequency again seems to introduce noise for thresholds leading to low FPR scenarios ($< 5\%$), regardless of weights. We conclude that the binary vector representation seems favorable over the frequency-based representation.

We now continue with a closer look at cut-point values as well as reasons for False Positives and Negatives. For binary vector `C-1024` with linear weights, a threshold of

- 0.18 leads to a TPR/FPR of 90.18% and 9.45%,
- 0.22 leads to a TPR/FPR of 89.10% and 4.74% (closest distance to the (0,1) point),
- 0.32 leads to a TPR/FPR of 86.55% and 0.99%,
- 0.55 leads to a TPR/FPR of 80.72% and 0.09%.

We now analyze the FPs and FNs in detail, using a threshold of 0.55 and thus providing a False Positive Rate below 0.1%. In this case, 409 FPs are found. A total of 225 (55.01%) FPs affecting 19 families are caused by confirmed code-sharing, i.e. relationships between distinct families in the Malpedia dataset, mostly caused by earlier code leaks. Among these are a cluster of 10 zeus-based families (Zeus, IceIX, Citadel, KINS, VMZeus, GameoverP2P, ...), and other families like ISFB/Dreambot, Pony/EvilPony, HLUX/Kelihos, and AlinaBot and its offspring.

Another 148 (36.19%) FPs affecting 36 families are the result of sharing larger parts of statically linked code from standard libraries. The groups can be identified: 15 families are written in Delphi and 3 written in Go, both languages known for massively linking library code during compilation. The other 18 families are written in C/C++, but they also share significant parts of WinAPI functions: a strong set of about 50 functions tied to statically linked functions from the MSVCRT. Many of these WinAPI functions are also among these with the highest occurrence frequency, as analyzed in Section 3.2.

Also, 8 FPs in 5 families were made with ApiVectors with less than 20 entries (as few as 3), which obviously results in inaccurate results that should be avoided.

The remaining 29 (7.09%) FPs are the most interesting. A total of 20 FPs with matching scores from 0.55 to 0.88 are between malware families that have been attributed by various analysts to the same 11 APTs, with APT1, APT28 and Lazarus being most prominent. Additionally, all of those tools are also considered private, i.e. their source is not available to other malware authors for reuse. Binary diffing showed that these tools do not necessarily have significant code overlap but seem to have similarity in the choice of WinAPI functions for different tasks, such as network connectivity or information gathering. We believe that in these cases, the authors "handwriting" may actually show to some degree through their code. This means that the technique could actually serve in the context of attribution, e.g. as one feature in a larger feature vector. The other 9 cases are also exclusively between families that have to be considered semantically similar: 2 RAT and 3 ransomware families. This shows that the presented technique to some degree also captures similarity in capability of programs.

With regard to False Negatives, a total of 271 are found for threshold 0.55. They can be categorized into three groups that we now analyze in more detail.

129 (47.60%) samples in 83 families do not exceed the threshold although there are theoretically matching samples available. We investigated this more closely and noted that in most cases the timestamps between available samples would span longer periods of time (up to several years) in which the evolution of the codebase changed so much that the WinAPI usage drastically changed in the process. For example in the malware family Qadars, with a certain version an API obfuscation scheme was introduced, while families UrlZone and Geodo were refactored into multiple modules over time, leading to a significant change in their WinAPI usage.

120 (44.28%) samples in 54 families are the result of how the data set is labeled. Because Malpedia does group associated modules like droppers, loader, or plugins along the families, these get matched against the main payload code. Because these modules often have a completely different code base and usage of WinAPIs, they technically cannot be compared and should be ignored in this evaluation.

20 (7.40%) samples in 12 families produce inaccurate matches because they have few (20 or less) imports, similar to the case with FPs.

If we completely disregard the threshold values and instead look at the Precision at Position 1 metric, our method still correctly classifies 86% of the samples, solely based on WinAPI import information.

To summarize, the analyzed ApiVector `C-1024` appears to be a decent configuration to be used for malware classification. Using a threshold between 0.32 and 0.55 leads to a FPR between 1% and 0.1%, which should allow for robust identification results. It appears worthwhile to further investigate the exact set of WinAPI entries that are affected by statically linked code and adjust their weight accordingly. Also, limiting application of the method to vectors of a given minimum size seems reasonable.

3.5 Comparison with ImpHash and ImpFuzzy

After evaluating the matching performance of ApiVectors in isolation and determining an optimal parameter configuration, we evaluate the methodology against two popular related approaches: `ImpHash` and `ImpFuzzy`. Both of these methods work by extracting a string representation of the import table as defined by the PE/COFF [26] standard and then deriving a checksum which is used for matching.

The `ImpHash` of a file is calculated by generating the MD5 checksum of the concatenated entries of the extracted import table. Since MD5 is a cryptographic hash, this approach only allows to find files that have exactly the same constellation of import table, even down to sequence of the imports. This further means that when comparing `ImpHash` values, there is no way of determining a fine-grained similarity except for the binary decision of equality. It has to be assumed that this can impact the matching performance when used for malware identification because changes to the code or even just recompilation of a program may potentially lead to a different import table or order of its entries. Both cases will prevent the sample from being matched to another sample of the same family.

The improvement of `ImpFuzzy` over `ImpHash` (as the name suggests) is the introduction of the fuzzy hashing algorithm SSDeep over MD5, which allows the computation of a similarity score. Now, even if two import tables, and thus their SSDeep checksum, are not completely identical, they may still achieve a value exceeding a given target threshold. Because the order of imports plays a role in checksum calculation, we have extended our experiments to alternative versions of `ImpHash` and `ImpFuzzy` with (alphabetically) sorted import tables to investigate if this has an influence on the results. Therefore, the following evaluation shows results for both the original methods as well as our modification with sorted import tables.

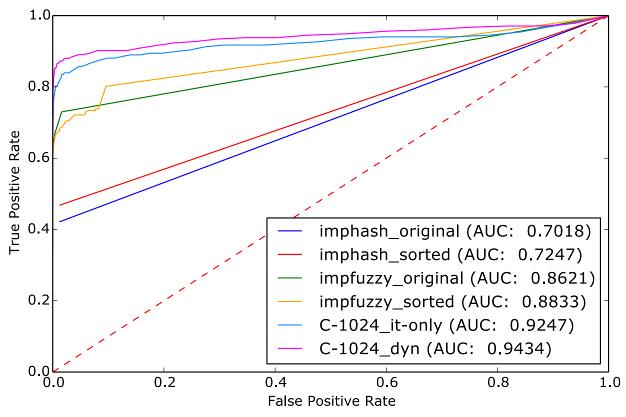


Figure 5: Classification Performance of ApiVector `C-1024` versus `ImpHash` and `ImpFuzzy` and their variants with sorted import tables.

In the following, we will discuss the results of `ImpHash`, `ImpFuzzy`, and ApiVector `C-1024`, which achieved the best overall performance in our previous analysis. Because ApiVectors normally work on both import table based and cached dynamic WinAPI references, we additionally analyze the performance of our approach with data limited to import tables to achieve a better comparison against the other approaches. Figure 5 shows the obtained classification performances for all methods.

As can be seen, `ImpHash` expectedly shows the lowest performance because of its strict matching method. Interestingly, the performance of this approach improves when being used with sorted import tables instead of the natural order encountered in the binaries. It is also interesting that `ImpFuzzy` is already a great improvement over `ImpHash`, and sorting imports in this case does more harm than good, at least at desirably low FPRs. We have again investigated the reasons for this. Apparently, `ImpFuzzy` is affected by the same effects as ApiVectors, i.e. exceptionally small import tables as well as such programs making extensive use of static linking, which shrouds the detectable characteristics of WinAPI usage to a certain degree.

Ultimately, ApiVectors perform best and give a substantial improvement of matching performance based on WinAPI information. It is also remarkable, that incorporating the information of cached dynamic imports on top of import table information has significant influence on the matching performance. This shows that the WinAPI functions referenced in this way carry highly relevant information, potentially also for the interpretative work of analysts.

Recalling the different representations for all three methods, it is obvious that the size of an `ImpHash` is always exactly 32 bytes (due to MD5), while the size of `ImpFuzzy` and ApiVectors may vary based on the number of entries. As shown in Table 6 the average length of `ImpFuzzy` is 54.6 Bytes, which makes it already considerably bigger than `ImpHash`. However, this increase of size does allow proper matching beyond an equality test with binary outcome. Nevertheless, both `ImpHash` and `ImpFuzzy` do not preserve any information about the imports used to construct their hashes and therefore do not allow reconstruction of this data.

As pointed out in Section 2.2, for ApiVectors we only discard likely dispensable information during the construction of ApiVectors and the vector configuration `C-1024` has an average API coverage of 86.7% (section Table 4). Apart from that, the information about which references to Windows API functions have been discovered by ApiScout (i.e. both static references from the import table and cached dynamic imports) are preserved in the compressed vector and can be fully reconstructed. The mean size for `C-1024` ApiVectors is 74.8 bytes. While ApiVectors are just 20 Bytes bigger than `ImpFuzzy` on average, they maintain the majority of information in a recoverable way. Additionally, the classification performance was evaluated to be much better than those of `ImpFuzzy` and `ImpHash`, which in our opinions easily justifies the bigger size and shows that ApiVectors can serve as a great feature in malware classification.

4 Related Work

In this section, we summarize related works on WinAPI import recovery, WinAPI import-based similarity analysis (both static and dynamic), and visualization of Windows API data.

Sharif et al. [5] have presented a similar method for resolving Windows API references based in Virtual Addresses that they limited to call targets derived from disassembly. They used it in the context of a reconstruction step in their unpacking framework Eureka. The *de facto* standard for import recovery among malware analysts is ImpRec [27] and its open-source pendant Scylla [4]. Both tools primarily aim at dynamically locating the IAT during the unpacking process, enumerating the respectively loaded DLLs export tables, and rebuilding an import table based on this information. Kotov et al. [28] recently also proposed a new method to analyze calls to the Windows API based on the composition of stack arguments.

`ImpHash` [7] was introduced by Mandiant as a method for hunting similar samples and is the refinement of a technique mentioned in an earlier FireEye report [29]. The basic idea is to concatenate the entries of a given PE files import table and calculate the MD5 hash over the resulting string. This technique has since also been adapted to Mach-O binaries under the name SymHash [30]. `ImpFuzzy` [8] has been proposed as an improvement for `ImpHash`. The basic methodology is identical (concatenation of import table entries) but instead of MD5, the fuzzy hash SSDeep [23] is used. Jang et al. [15] presented BitShred, an approach to encode malware samples as high dimensional feature bit vectors. Windows API calls are among the dynamic analysis based features used and for calculating similarities, they also used the Jaccard index.

Rieck et al. [1] dynamically extracted Windows API call sequences using a sandbox and then apply machine learning to train a behavior classifier. Similarly, Fujino et al. [31] extracted API calls through dynamic analysis using a sandbox in order to derive similarity of executions. They first mapped the individual API calls to a normalized representation and then, using text-mining techniques, they extracted a feature vector used for clustering similar samples. Fredrikson et al. [32] presented an approach to automatically derive WinAPI-based specifications for discriminative, potentially malicious behavior from execution traces. Alazab et al. [33] presented a detection method based on structural and behavioral features of malware, including Windows API usage. Additionally, Alazab et al. [34] introduced an approach to map Windows API functions to potential malicious behavior. Zwanger et al. [35] mapped Windows Kernel API functions to semantic classes and presented a static analysis approach to detect rootkits based on the occurrence distribution of these classes.

Trinius et al. [36] used treemaps to visually present patterns of occurrence frequency for Windows API calls observed in sandbox runs. Gove et al. [37] created SEEM, a system to explore and compare features of multiple malware samples. Among other things, they mapped Windows API functions to capabilities and use bit vectors for visual comparison.

5 Conclusion

In this paper we made four contributions. Firstly, we created ApiScout, a method to recover API usage information from memory dumps. It is capable of handling dynamic imports and does not need a live process environment. This leads to ApiScout redefining the possibilities in API usage information recovery by surpassing the capabilities of former state of the art tools ImpREC and Scylla.

Secondly, we defined ApiVectors, a way to efficiently store and compare Windows API information. It enables analysts to do a cursory assessment of potential behavior of a binary as well as similarity analysis of API usage across binaries.

Thirdly, we researched 589 distinct malware families from the Malpedia corpus regarding their API usage. We showed that all families combined use not more than around 4500 API functions. Additionally, our extensive evaluation of the similarity assessment and matching performance of ApiVectors provides valuable insights into the possibilities and limitations of (purely) WinAPI-based static malware classification. For example, we showed that for the majority of malware families the set of used WinAPI functions is characteristic, except for only a few cases in which binaries with large parts of statically linked libraries (using programming languages such as Delphi or Go) will negatively impact the matching performance because they introduce a large common set of WinAPI functions they collectively depend on. With regard to the results, ApiVectors identify about 86% of samples correctly, which is a significant improvement over the currently used approaches `ImpHash` and `ImpFuzzy`.

Last but not least, we publish both, ApiScout and ApiVectors, as a production-ready Python library together with a reference set of ApiVectors for more than 700 malware families. We will also make our evaluation framework available to enable reproduction of our experiments and results.

System API usage information has always been a core asset for in-depth analysis of malware. In this field, ApiVectors enables the assignment and visualisation of semantic classes to APIs in binaries. This can easily be included in static analysis tools like IDA Pro and should lead to easier orientation in a binary for an analyst. Furthermore, we hope to have shown that it is also a very valuable feature in malware classification that is worth more research. It should even be a good asset in additional contexts like authorship analysis and attribution. The threat actors APT1, APT28, and Lazarus Group could be a good study object for such research since their extensive sets of exclusive tools are well documented.

Acknowledgment: The authors would like to express eternal gratitude to the Shadowserver Foundation for continuously supporting malware research. We would also like to thank the anonymous reviewers of Botconf as well for their valuable feedback.

Author details

Daniel Plohmann

Fraunhofer FKIE
Zanderstr. 5, 53177 Bonn
daniel.plohmann@fkie.fraunhofer.de

Steffen Enders

TU Dortmund
Otto-Hahn-Str. 14, 44227 Dortmund
steffen.enders@tu-dortmund.de

Elmar Padilla

Fraunhofer FKIE
Zanderstr. 5, 53177 Bonn
elmar.padilla@fkie.fraunhofer.de

References

[1] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and Classification of Malware Behavior," in *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'08)*, July 2008.

[2] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (IEEESP)*, May 2015.

[3] D. Plohmann, M. Clauss, S. Enders, and E. Padilla, "Malpedia: A Collaborative Effort to Inventorize the Malware Landscape," *The Journal on Cybercrime and Digital Investigations*, vol. 3, January 2018.

[4] NtQuery, "Scylla Imports Reconstruction," 2011. GitHub Repository: https://github.com/NtQuery/Scylla.

[5] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A Framework for Enabling Static

Malware Analysis," in *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS'08)*, October 2008.

[6] I. Guilfanov, "IDA Pro," 2018.

[7] Mandiant, "Tracking malware with import hashing," January 2014. Blog post: https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html.

[8] S. Tomonaga, "Classifying malware using import api and fuzzy hashing – impfuzzy," May 2016. Blog post for JPCERT/CC: https://blog.jpcert.or.jp/2016/05/classifying-mal-a988.html.

[9] J. Bader, "Android Package Index," 2018. Overview of system API: http://www.johannesbader.ch/tag/dga/.

[10] Microsoft, "Conventions for Function Prototypes," tech. rep., Microsoft, 2018. MSDN Article: https://msdn.microsoft.com/de-de/library/windows/desktop/dd317766(v=vs.85).aspx.

[11] A. Fog, "Calling conventions," tech. rep., Technical University of Denmark, April 2018.

[12] S. Josefsson, "RFC 4648: The Base16, Base32, and Base64 Data Encodings." http://tools.ietf.org/html/rfc4648/, Oct. 2006.

[13] S.-s. Choi, S.-h. Cha, and C. Tappert, "A survey of binary similarity and distance measures," *Journal of Systemics, Cybernetics and Informatics*, 2010.

[14] P. Jaccard, "The distribution of the flora in the alpine zone.1," *New Phytologist*, vol. 11, February 1912.

[15] J. Jang, D. Brumley, and S. Venkataraman, "BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, (New York, NY, USA), ACM, 2011.

[16] T. T. Tanimoto, *An elementary mathematical theory of classification and prediction by T.T. Tanimoto*. International Business Machines Corporation New York, 1958.

[17] Fraunhofer FKIE, "Malpedia," December 2017. Website for the corpus: https://malpedia.caad.fkie.fraunhofer.de.

[18] C. Rossow, C. J. Dietrich, C. Kreibich, C. Grier, V. Paxson, N. Pohlmann, H. Bos, and M. van Steen, " Prudent Practices for Designing Malware Experiments: Status Quo and Outlook ," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, May 2012.

[19] A. Lesne, "Shannon entropy: a rigorous notion at the crossroads between probability, information theory, dynamical systems and statistical physics," *Mathematical Structures in Computer Science*, vol. 24, June 2014.

[20] Microsoft, "Windows GDI," tech. rep., Microsoft, May 2018. MSDN Article: https://docs.microsoft.com/en-us/windows/desktop/gdi/windows-gdi.

[21] Microsoft, "MFC Desktop Applications," tech. rep., Microsoft, 2018. MSDN Article: https://msdn.microsoft.com/en-us/library/d06h2x6e.aspx.

[22] Microsoft, "VARIANT structure," tech. rep., Microsoft, 2018. MSDN Article: https://msdn.microsoft.com/en-gb/library/windows/desktop/ms221627(v=vs.85).aspx.

[23] J. Kornblum, "Identifying Almost Identical Files using Context Triggered Piecewise Hashing," *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, vol. 3, August 2006.

[24] D. Hilbert, "Über die stetige abbildung einer linie auf ein flächenstück," in *Dritter Band: Analysis · Grundlagen der Mathematik · Physik Verschiedenes: Nebst Einer Lebensgeschichte*, Springer Berlin Heidelberg, 1935.

[25] P. Collinson, "Of bombers, radiologists, and cardiologists: time to ROC," *Heart*, vol. 83, September 1998.

[26] Microsoft, "PE Format (Windows)," tech. rep., Microsoft, 2017. MSDN Article: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx.

[27] mackT, "Import REConstructor (ImpREC)," 2001. Tool entry in the Woodmann RCE library: http://www.woodmann.com/collaborative/tools/index.php/ImpREC.

[28] V. Kotov and M. Wojnowicz, "Towards Generic Deobfuscation of Windows API Calls," in *Proceedings of the Workshop on Binary Analysis Research (BAR)*, February 2018.

[29] FireEye, "SUPPLY CHAIN ANALYSIS: From Quartermaster to Sunshop," tech. rep., FireEye, November 2013.

[30] A. Shelmire, "SymHash: An ImpHash for Mach-O," October 2016. Blog post for Anomali: https://www.anomali.com/blog/symhash.

[31] A. Fujino, J. Murakami, and T. Mori, "Discovering similar malware samples using api call topics," in *Proceedings of the 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, January 2015.

[32] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, “Synthesizing near-optimal malware specifications from suspicious behaviors,” in *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, May 2010.

[33] M. Alazab, R. Layton, S. Venkataraman, and P. Watters, “Malware Detection Based on Structural and Behavioural Features of API Calls,” in *Proceedings of the 1st International Cyber Resilience Conference*, August 2010.

[34] M. Alazab, S. Venkataraman, and P. Watters, “Towards Understanding Malware Behaviour by the Extraction of API Calls,” in *Proceedings of the 2nd Cybercrime and Trustworthy Computing Workshop (CTC’10)*, July 2010.

[35] V. Zwanger and F. C. Freiling, “Kernel mode api spectroscopy for incident response and digital forensics,” in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW), Rome, Italy*, 2013.

[36] P. Trinius, T. Holz, J. Goebel, and F. C. Freiling, “Visual Analysis of Malware Behavior using Treemaps and Thread Graphs,” in *Proceedings of the 6th International Workshop on Visualization for Cyber Security*, Oct 2009.

[37] R. Gove, J. Saxe, S. Gold, A. Long, and G. Bergamo, “SEEM: A Scalable Visualization for Comparing Multiple Large Sets of Attributes for Malware Analysis,” in *Proceedings of the 11th Workshop on Visualization for Cyber Security (VizSec’14)*, November 2014.