Malpedia: A Collaborative Effort to Inventorize the Malware Landscape

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

It is published in the Journal on Cybercrime & Digital Investigations by CECyF, https://journal.cecyf.fr/ojs
It is shared under the CC BY license http://creativecommons.org/licenses/by/4.0/.

Abstract

For more than a decade now, a perpetual influx of new malware samples can be observed. To analyze this flood effectively, static analysis is still one of the most important methods. Thus, it would be highly desirable to have an open, freely accessible, curated, and cleanly labeled corpus of unpacked malware samples for research on static analysis methods. In this paper, we introduce *Malpedia*, a collaboration platform for curating a malware corpus. Additionally, we provide a baseline for a cleanly labeled malware corpus consisting of 607 families divided into 1792 samples. This corpus offers a plethora of possibilities for researchers, including using it as a testbed for evaluations on detection and analysis methods, quality assurance for classification, and contextualization of new malware.

To ensure the quality of our corpus, we adapted the requirements by Rossow et al. [1], derive specific requirements for the context of static malware analysis, and evaluate our corpus against them.

Based on our corpus, we show that looking beyond packers dramatically reduces the size needed for a corpus to be representative, as the number of distinct malware families and versions after unpacking is orders of magnitude smaller than the number of unique packed samples. Additionally, we perform a comprehensive study of the Windows malware in the corpus, scrutinizing its structural features. This analysis clearly illustrates that *Malpedia* offers a wealth of information, readily available for in-depth investigations.

Keywords: malware corpus, malware analysis

1 Introduction

It is a well-known fact that the number of registered unique malware samples observed since 2005/2006 has drastically increased, and currently sits at almost 700 million samples as e.g. tracked by AV Test [2]. This is a direct consequence of the common use of so called *packers*, auxiliary programs that contain routines such as compression or encryption algorithms used to alter the appearance of the respective payload code in order to protect it against detection or analysis. In fact, it can be assumed that the actual number of malware families or versions is significantly below these figures. Nevertheless, these observations can be taken as a symbol for the massive gain in importance of malware over the last decade, be it as a tool for digital crime or in the context of state-sponsored activities.

The overwhelming number of samples and its fast-paced growth has strong implications on malware research. They lead to a high demand for advanced methods to detect, analyze, and contextualize malware. While researching such methods, it is of uttermost importance to have representative data for evaluation available. Rossow et al. [1] have shown that in past academic studies there has been a lack of comprehensiveness and representativeness in the data used. This renders academic research less effective.

Ideally, researchers would have access to an independent, pooled resource that provides them with confidently labeled, unpacked reference samples for malware families and versions, alongside available meta information such as pointers to analysis reports or detection capabilities such as accurate YARA rules. Our

evaluation shows that there is in fact huge redundancy in packed samples versus unpacked reference samples for families and versions. Thus, such a corpus could likely be distilled down to a couple thousand instead of many hundred million files.

To address the above-mentioned situation, in this paper we introduce an independent platform called *Malpedia*. It allows malware researchers to contribute to a centrally curated, free corpus. Initially, it offers access to a corpus we created since 2015, spanning around 1800 cleanly labeled samples representing more than 600 families for several platforms, including Android, macOS/iOS, Linux/ELF and Microsoft Windows.

To show the usefulness of *Malpedia* and the corpus in its current stage, we conduct a comparative study on several characteristics across the families of Windows malware archived at the time of writing. This study leads to the following core results:

- Packers mostly serve as an initial barrier against detection.
- The number of unique unpacked samples is orders of magnitude smaller than the one of packed samples.
- Unpacked samples can be conveniently treated with methods of static analysis.
- The information extracted even with just cursory methods already gives an interesting insight in preferences and choices of malware authors.

We implement our comparative analysis in such a way, that it can be continually executed over the growing data set in the future and provide updated results via the web portal found at https://malpedia.caad.fkie.fraunhofer.de.

In summary, we make the following contributions with our paper:

- We define a set of requirements that malware corpora intended for static analysis should follow.
- We introduce a vetted collaboration platform with the mission to curate a reference malware corpus, providing unpacked and dumped samples to specifically address static analysis needs.
- We provide a malware corpus for free to the research community.
- We perform a comprehensive, quantitative static analysis of structural features across 446 families of Windows malware.
- We show that it is viable to assume that packers serve mostly as an initial barrier and beyond this, the number of distinguishable families and versions is magnitudes smaller than the number of unique samples encountered in the wild.

The remainder of this document is structured as follows. Section 2 specifies requirements which a malware corpus optimized for static analysis should follow. Section 3 presents the way in which we have implemented these requirements to build our corpus, *Malpedia*, and provides an outline of its current status. We use Section 4 to give insight into the design of the platform that we have created to share the corpus with the community and with which we want to curate it in the future. In Section 5, we use the current status of the data set to conduct a comparative study across various structural features, showcasing the usefulness of the *Malpedia* corpus for static analysis. Section 6 summarizes related work while Section 7 concludes the paper.

2 Goals and Requirements for a Malware Corpus focusing on Static Analysis

In this section, we first define general goals that a malware corpus focusing on static analysis should fulfil.

We then focus on requirements to ensure that a corpus is created towards these goals. For this reason, we recapitulate the guidelines to consider when planning and executing malware experiments, as defined by Rossow et al. [1] in 2012. After checking their applicability for this special case, we rephrase and extend these aspects and combine them into our own set of requirements.

2.1 Goals

The primary goal of any malware corpus should be to contain representative data. Ideally, it should cover longer periods of time within the malware landscape, in order to allow comprehensive measurements, reflecting changes over time. In that sense, it should also naturally be updated as necessary to keep up with the frequent developments as encountered when dealing with malware.

Furthermore, the corpus should provide the data in an easily accessible format. We believe that providing some verified unpacked format of the samples adds a huge benefit over just providing their original state (be it packed or not). As the corpus described here is intended to be specifically suited for static analysis, this allows to directly orientate analyses from a viewpoint behind the classical packer-barrier. Just as important, a malware corpus should provide rich meta information, at the very least accurate labels, in order to better judge and classify findings derived from it. Note that the packer-barrier has a strong presence in essentially every malware feed or data set currently available, and meta information is usually limited to detection labels as provided by anti-virus software.

Another goal should be that such a malware corpus is a resource equally useful for practitioners to ensure it is being reviewed for relevance from different perspectives. By providing vendor-neutrality in its coverage, it could also serve as a consensual ground-truth among malware researchers, both for naming in the concrete case and as a source for the verification of identification measures in general.

2.2 Adoption of Prudent Practices

Rossow et al. [1] defined their set of guidelines in a way that they would be applicable for a wide range of experiments involving malware. The guidelines address evaluations of techniques for detection, classification, and behavioral analysis, based on dynamic, static, or combined analysis methods.

This paper focuses specifically on the creation of a malware corpus optimized for static analysis. We therefore review the applicability of these guidelines, selecting the relevant subset that we consecutively include into our requirements for such a corpus. Rossow's guidelines are grouped into 4 categories: correctness of the data set, transparency, realism, and safety.

First, with regard to the correctness of data sets, it is highly relevant to balance over malware families and even platforms as malware has drastically diversified over the last years (cp. 2.3.2). Environment artifacts have only minor impact in the context of static analysis and they do not have to be addressed beyond documentation (cp. 2.3.5). Generally, goodware can be ignored since the goal is to compile a malware corpus. In the same way, there should be no side effects occurring related to blending with benign data. Since no detection or classification methods are proposed or benchmarked, splitting of training and evaluation data sets is not applicable, similar to monitoring of malware and the system privileges that would be used for this.

Second, with regard to transparency, we consider annotations with family names and further meta data as a core requirement for such a corpus tailored for static analysis (cp. 2.3.4). This requires that samples as clean as possible, i.e. they are unpacked and isolated from potential packer fragments in order to allow for their accurate identification (cp. 2.3.3). With regard to the sample selection policy, such a corpus should aim at both coverage in number of discernable families and versions to enable a comprehensive analysis of code evolution (cp. 2.3.1). The method and environment (including network connectivity) used to create the data for the corpus should be carefully documented (cp. 2.3.5). Not relevant is the review of true and false positives and negatives, since the requirements address a pure data collection project.

Third, with regard to realism, a corpus should aim at providing a collection of prevalent and timely malware families (cp. 2.3.1), including being kept up to date (cp. 2.3.6). This also ensures relevance for real-world applications in which the corpus may be used later on (cp. 2.3.1). The provision of appropriate malware stimuli in the sense of the corpus can be translated into ensuring that all modules, e.g. for different platforms or bitness, or providing additional functionality are extracted alongside in the unpacking process (cp. 2.3.3). Generalization of results from an OS version or providing Internet access is again not applicable as these requirements do not address an experiment in itself.

Fourth and finally, safety and with that the containment policy is mostly of importance for the further dissemination of the resulting corpus (cp. 2.3.6).

2.3 Definition of Requirements

Having reviewed the prudent practices for reasonably applicable components, we now formulate a set of requirements specifically tailored for a malware corpus focused on static analysis.

2.3.1 Representative Content

The most important requirement for a corpus is to be representative. This means that the selection of samples contained in the data set should be prevalent and suitable for the deduction of results that are of real-world relevance.

Another requirement in this context is to favor quality over quantity. In this regard, the intention is to strictly avoid redundancy in the samples contained, from the perspective of discernable malware families and versions beyond the packer-barrier. This is a viable requirement since the focus is on static analysis, and since unpacked versions of the samples are provided, de-duplication can be applied. Barabosch et al. [3, 4] showed that under given circumstances even small data sets of carefully selected samples can yield representative results.

This matches our own experiences, motivated by the following example: In cooperation with Shadowserver, we identified more than 80.000 samples of Citadel since 2013 and performed datamining on their botnet configurations. Over this time, we have observed more than 140 different unique identifiers correlating to builder kits, underlining its prominence among criminal actors at that time. However, all of the samples we observed can be represented by a mere 21 distinct versions, achieving a data reduction factor of 3,800x for this family with respect to this viewpoint. We observed similar factors for other families analyzed in a similar way, for example: TinyBanker (5,700x), Asprox (5,500x), VMzeus (471x), and KINS (105x).

2.3.2 Cross-Platform Orientation

Nowadays, the existence of malware for many different hardware platforms has been proven. With the trend of embedding technology into various objects of everyday live, it is more than likely that multi-platform orientation will become more and more important. Therefore, a malware corpus should not limit itself to a single platform. If desired, this reduction can be achieved artificially later on by selecting only parts of the corpus.

2.3.3 Unpacked Samples

To adhere to the goal of easily accessible contents in the context of static analysis, it is required to provide unpacked counterparts of the samples if needed and where applicable. This is essential to enable static analysis on the actual malware families' code in the first place. Additionally, these unpacked versions should be kept as free as possible from any packer fragments to not interfere with analysis.

However, we want to adjust this requirement of unpacking for two reasons to focus on memory dumps instead.

First, we think it is actually more beneficial to capture malware in the way closest to how it is typically naturally encountered. Clean unpacked samples are rarely found in the wild and often a result of either an actor's mistake or an analyst's efforts. This leads us to instead consider an in-memory view of an unpacked malware as a preferable option, which can be usually obtained by performing a memory dump.

Second, this in-memory view is actually more generic and can even serve as a form of normalization across families. Especially, as not every malware specimen can be transformed into an unpacked sample in the traditional sense of a runnable on-disk version of itself. Some specimen exist only in a shellcode representation that may be entirely dynamically loaded by other components.

We believe that dumps are a favorable approximation to unpacking for multiple reasons:

- Dumping is drastically easier to automate, as we do not aim for full reconstruction but rather something suitable for static analysis.
- Since the corpus aims for packed and unpacked counterparts, omitting intermediate packer/loader stages that could be addressed with granular unpacking is an acceptable compromise for higher automation success.
- Dumps may contain additional runtime-only information such as decrypted strings or API imports that ease analysis and provide additional insights. The imported libraries are also already mapped into memory.

It is important to note that basic information about the origin of the dump should be recorded as well, at least the base address of the memory segment it was taken from.

2.3.4 Accurate Labels and Meta Data

Another requirement is that samples should have annotations, at least an accurate label regarding their family membership. Where applicable, additional meta data should be recorded, such as version numbers within the family. In cases where the unpacking/dumping yields multiple results of interest, these have to be annotated accordingly. Typical cases where this can occur are families that consists of multiple modules or come with plugins, deployed from a single dropper file.

2.3.5 Documentation of Data Generation

It is important for a malware corpus to document how it was created, in order to enable accountability and reproducibility. In the first place, it should be tracked where the malware samples originate from. Furthermore, if the corpus contains derived data (such as dumps), it is also necessary to document the applied methods and their configurations (e.g. environment specification, parameters).

2.3.6 Curation and Dissemination

Should the corpus be updated over time, then it is important to ensure consistency in its structure and content. For this reason, the content should be curated in the same way it was originally created.

Additionally, the data contained in such a corpus is potentially harmful for many computer systems. It should be only made accessible to parties that are trusted and believed to be able to handle such a data set with the required carefulness.

3 The Malpedia Corpus

In this section, we introduce the *Malpedia* corpus, which has been developed in line with the requirements specified in Section 2. We first define our interpretation of the terms *Malware Family*, *Unpacked Sample*, and *Dumped Sample*. Next, we explain our approach for sample selection and our method to create clean memory dumps of malware. We continue by documenting our data storage format and give an overview over the current contents at the time of writing.

3.1 Terminology

Before explaining our methodology, we define some reoccurring terms as used in the context of *Malpedia*.

Malware family: We use the term malware *family* to group all malware samples that from a developer's point of view belong to the same project, i.e. code base. This also incorporates potentially existing additional components as used by the malware such as tailored loaders or plugins. We are aware that this definition is not fully sound and carries fuzziness with regard to the origin and similarity of code as well as its potential authorship. But from our impression, this ultimately allows us to reflect the current consensus among many practitioners. Furthermore, this definition gives us some freedom to make a distinction between the outcome of leaked source code (e.g. the offsprings of Zeus, Carberp, or Gozi) but also consolidate rewrites of the same project (e.g. GPCode).

Unpacked sample: An unpacked sample is a direct representative of the malware family itself, without presence of any third-party code not related to its

own code base that may have been applied post-compilation to conceal its identity. This does not address the removal of any staging or obfuscation as employed by the family itself. Ideally, an unpacked sample is also in a state where it can be natively started within its compatible operating systems, e.g. unmapped and with a correct entry point.

Dumped sample: We consider a runtime memory capture of the malware during execution as a dumped sample. In the majority of cases, this will originate from a memory-mapped process image with arbitrary initialized dynamic data, e.g. Windows API function imports or global variables.

3.2 Collection Approach

In *Malpedia* we want to prefer *quality over quantity*. This means that increasing the coverage of unpacked/dumped samples for a new family is more important than adding further samples for a known family. Instead of collecting as many samples as possible, we put emphasis on the importance of verification for all samples before adding them to the corpus to maintain a high degree of quality. By this, we hope to primarily increase the longitudinal coverage and achieve topicality in the data.

Furthermore, we aim to *prioritize prevalent malware families*, thus favor malware families that are very active and affect many users or high-value targets over others. In the default case, it should be sufficient to choose single representative samples per version for basic coverage. This also drastically reduces the amount of data required to meaningfully represent even lines of evolution for malware families.

Finally, we want to *provide means for quality assurance* by striving for complete YARA coverage. Precluding false positives and negatives across the whole data set on the one hand can serve as a proof of label accuracy while on the other hand creates a useful tool for malware identification.

The vast majority of samples included in *Malpedia* are also found in other repositories, such as VirusTotal.

Using this orientation, we hope to create a representative corpus (2.3.1) without platform limitations (2.3.2), and with accurate labels (2.3.4) for all samples.

3.3 Dump Creation and Family Identification

We decided to centralize and normalize the dump creation (even for future samples) by using virtualization. This allows us to always resort to the same VM images and state to ensure consistency across all dumps. So far, we only perform dumps for Microsoft Windows and limit ourselves to two versions: Windows XP SP3 and Windows 7 SP1 64bit, as they cover all our current needs. This preference for *older* OS versions is explained with the fact that we want to ensure maximum execution success of samples that may be impaired by modern security mechanisms embedded into recent versions of Windows. Additionally, all available Microsoft Visual Studio (MSVC) and .NET runtimes are installed. To further ensure maximum compatibility with packers, we have taken (not publicly documented) steps to harden the VM against detection.

Sticking to a limited set of VM snapshots yields stable and known environment parameters, such as user and computer names, Volume IDs, and Windows DLL versions. The latter enables the use of ApiScout, a technique we present in section 5.3.1, on all dumps contained in *Malpedia*. This method will be analogously expanded to also cover ELF and potentially macOS/OSX families, while Android and iOS malware will likely be provided as unpacked samples, if applicable and necessary.

In order to create dumps, we currently stick to the following procedure. We first attempt to simply start the sample of interest and wait for a period of time (by default 60 seconds, prolonged if necessary). After this time, we perform a full differentiation of allocated memory versus the clean state and dump all sections that have changed. We then use a set of heuristics to aid our following *manual* inspection in order to select the reference dumps for the sample. In cases where this method fails, we use manual in-depth static and dynamic analysis, to guide the unpacking process in order to yield an acceptable result.

The actual memory dumping is then performed using a kernel driver to avoid interference with hooks potentially set by the malware. Regardless of how the resulting memory dump was produced, we clean the data (i.e. remove packer fragments) if necessary, adequate, and possible.

Family identification is then performed using the following steps sorted by priority: applying existing YARA rules, verifying the classification that may have been available along the sample (e.g. analysis report or blog post), and using similarity analyses against the corpus of existing unpacked files and dumps.

We believe that these measures sufficiently fulfill the requirements of providing unpacked files (2.3.3), accurate labels (2.3.4), and documentation of data generation (2.3.5).

3.4 Storage and Organization

We use a hierarchical folder structure to project the data into families and versions. On top-level, we generally use a nomenclature of ⟨platform⟩.⟨name⟩, where ⟨platform⟩ may be `win`, `osx` or similar and ⟨name⟩ is typically one of the identifiers as given by third parties or `unidentified_⟨number⟩` where no such name can be identified. In cases of multi-platform malware, we resort to an identifier of the filetype or programming language such as `jar` or `js`.

Samples are stored by their SHA-256 hash, and associated unpacked or dumped files are stored as `sha256_unpacked` and `sha256_dump_⟨addr⟩` respec-

tively, where ⟨addr⟩ is the dump's originating base address extended to the addressing size, such as 0x00400000. In some cases, multiple dumps will be taken, to also account for additional modules deployed to memory or anti-analysis tricks. Depending on the concrete needs, subfolders may be used below the family identifier to indicate a version (e.g. by internal scheme, compilation timestamp if it proves to be reliable, or a first seen date) or component type (loader, payload, modules, ...). Furthermore, meta data and YARA rules are stored along the samples.

3.5 Data Set Status

All of the following evaluations refer to the *Malpedia* repository state as of October 31, 2017 (commit bf6532c). At this time, the corpus contains a total of 1792 samples inventorized into 607 families.

- 505 families for Windows
- 34 families for Android
- 29 families for macOS/OSX
- 24 families in ELF format
- 2 families for iOS
- 1 family for Symbian
- 12 families that are scripted or for other reasons potentially multi-platform

With regard to their state of unpacking and dumping

- 1149 (64.12%) samples are dumped (and partially also unpacked)
- 221 (12.33%) samples are just unpacked
- 422 (23.55%) samples are neither dumped or unpacked

So far, only Windows has been addressed with dumps, where 446 (88.32%) of the families are covered with at least one dump. Out of these, 98 families have documented use by one or more Advanced Persistent Threat (APT) actor groups. The total number of dumps is 1208 as for some samples multiple stages, e.g. loader and payload have been dumped.

With regard to YARA, 255 rules for 150 families exist, covering 907 (50.56%) samples. The long term goal is to achieve perfect coverage across the full corpus.

4 The Malpedia Platform

In this section, we present the platform that we have created to maintain the corpus in the future. Based on the feedback on a textual draft that was gathered from our peers in the CERT and research community, we explain its primary implementation aspects. This platform is our way of performing curation and controlling dissemination, as required (2.3.6).

It is important to note that *Malpedia* is operated and all data that is collected in it is made available under *Creative Common's CC BY-NC-SA license*, in order to express our vision of creating an independent, reusable resource.

But before going into details, let us first define the philosophy behind *Malpedia*.

Malpedia's Mission Statement: The primary goal of *Malpedia* is to provide a community-driven, independent resource for rapid identification and actionable context when investigating malware. Openness to curated contributions shall ensure topicality and an accountable level of quality in order to foster meaningful and reproducible research.

4.1 Implementation of Trust Mechanisms

The data of which *Malpedia* constitutes contains potentially sensitive and dangerous contents. It is therefore warranted to ensure a limitation of access to an audience aware of the risks and experienced in handling such data. We have decided to introduce a vetting process as access control measure and adopt the Traffic Light Protocol (TLP) [5].

We consider the majority of meta data such as names, aliases, referenced reports, and aggregated statistics as not critical and make them publicly available (TLP:WHITE), allowing them to be used as a reference.

Elements identifying concrete samples (hashes) and the actual malicious code itself should be withheld from public access (TLP:GREEN) in order to not tip off the attackers or harm bystanders. Means of detection such as YARA rules may be publicly sharable, depending on their source of origin but can also be further limited in distribution (up to TLP:AMBER).

These are best practices that have proven of value based on experiences gathered in trust groups such as closed-door mailing lists or conferences. For this reason, *Malpedia* will be operated in favor of established trust-mechanisms: The user base will be grown in an invite-only way where users will have to be vetted by a portion of the existing users in order to have their account activated.

4.2 Ensure High Standards for Contribution Quality

To maintain a high quality, contributions will only be accepted from registered users but not without further review. We use a double blind peer review model to validate the quality of submissions before integrating them into the corpus. We require at least two verdicts per submission. Registered users may volunteer as reviewers.

4.3 Automation Support

The tool landscape for malware research has significantly grown over the last years. To maximize the usefulness of the platform, the ways of interaction as provided through the website will also be made available via a REST API to allow easy integration for third parties. Additionally, direct access to the full corpus will be offered to registered users.

4.4 Baseline Data Set

Starting from scratch would likely be deterrent to the willingness of users to contribute. Therefore, we have bootstrapped *Malpedia* with reference data collected in numerous malware investigations conducted over the last five years. Additionally, we have systematically crawled publications (analysis reports, papers, blog posts) of major institutions, such as AV and threat intelligence companies and isolated samples as representatives for more than 600 families. An overview of the contents of this initial data set is given in Section 3 and a comparative analysis of the contained Windows malware is given in Section 5.

4.5 Contextual Enrichment: Meta Data

Many malware families are given multiple names, for example due to parallel discovery, company policy, or simply personal taste. On many occasions, this has caused unfavorable confusion within the malware research community.

With *Malpedia*, we want to provide a central resource for tracking as many of these respective aliases as we can identify and support them with concrete samples to build consensus on. Where applicable, we also track recorded links between malware families and threat actors, for which we also want to provide a bookkeeping of aliases. To avoid duplication of effort, we integrate and feed data back to the registers administered by the team of the Malware Intelligence Sharing Platform (MISP) [6].

Additionally, we think it is beneficial to also collect references to published research on malware families, including analysis reports, blog posts etc. to enrich the samples with contextual information. We hope that this will provide analyst's with information to bootstrap their own analyses on.

Finally, we have structured our suite of evaluation tools used in Section 5 in a way that it allows us to easily integrate it directly into the *Malpedia* web service. This allows us to continually provide updates on comparative assessments on basic structural properties of all the malware families' code bases as the corpus grows.

5 A Comparative Structural Analysis of Windows Malware

In this section, we focus on the subset of Windows malware to perform a comparison of selected structural properties. The families are listed in the Appendix, Table 6. We use our chosen normalized representation of dumps as motivated in Section 3.3. The analysis is split up into three parts.

The first part focuses on PE headers. Based on our experience, a majority of packers will simply unwrap their carried payload to memory in its original form, which allows us to inspect the actual PE headers of the original malware families. Apart from finding out in how many cases we have headers available, we can check and compare many header fields that may be relevant for analysts, such as the presence of header magics (such as MZ/PE), if the binary is for 32/64bit, if it is a DLL vs. an EXE, security properties, and so forth. Of special interest are also compilation timestamps, as these allow us to measure the age of our corpus and provide temporal context within the development of families. Features that characterize the workflow of malware authors are hints on languages and compiler versions used, including Rich Headers, which can give some insights on the system environment that the program was compiled in [7].

The second part examines properties of the malicious code itself. However, we limit ourselves to a very cursory analysis and will cover this topic in-depth in a dedicated follow-up publication. We use our own disassembler SMDA, which is optimized for function coverage in arbitrary code buffers (such as the dumps found in *Malpedia*) in order to derive key metrics of the code graphs. Apart from that, we have a closer look at the debugging information available and how this is used e.g. for naming of malware families.

The third part gives an overview of Windows API usage. We first classify three different import styles and then use ApiScout [8], a tool we developed to statically identify references to the Windows API. Over the data extracted, we perform a frequency analysis similar to Zwanger et al. [9] to measure the popularity of DLLs and API functions.

While we believe that we already have decent coverage, be aware that these are our initial results and that we will continually publish future statistics through our platform as the corpus grows.

5.1 PE Header Analysis

The first analysis part focuses on data contained in PE headers. As PE headers are the blueprints of Windows executables, they contain a lot of meta data that can be potentially useful e.g. for initial triage or as an outline when starting to analyze the malware in-depth. We start by assessing the general availability of PE headers in memory dumps because this determines the applicability of further methods. Because headers are luckily available for 94.62% of the families considered, we continue evaluating a range of header fields comparatively. This includes general characteristics, the compilation timestamp, linker information, and structural information such as the presence of data directories.

For our analysis, we avoid the usage of PE parser libraries because our input data are (potentially modified) memory dumps and not clean on-disk versions with intact header magics, which most parser libraries expect. Alternatively, we use a method we call `pe_check`. It is built around the idea of specifically avoiding the obvious header magics and instead locating a composition of mandatory header fields as orien-

```
Offset     Hexdump                                                                  Text

00000000: 4D5A9000 03000000 04000000 FFFF0000 B8000000 00000000 40000000 00000000  MZ..........ÿÿ..ž.......@.......
00000020: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 C0000000  ............................À...
00000040: 0E1FBA0E 00B409CD 21B8014C CD215468 69732070 726F6772 616D2063 616E6E6F  ..º..Ž.Í!ž.LÍ!This program canno
00000060: 74206265 2072756E 20696E20 444F5320 6D6F6465 2E0D0D0A 24000000 00000000  t be run in DOS mode....$.......
00000080: 498D63B3 0DEC0DE0 0DEC0DE0 0DEC0DE0 608D6190 68886481 2D9E6283 669F2CC1  I.c³.ì.à.ì.à.ì.à`.a.h.d.-.b.f.,Á
000000A0: 52696368 0DEC0DE0 00000000 00000000 00000000 00000000 00000000 00000000  Rich.ì.à........................
000000C0: 50450000 4C010500 7CF2275A 00000000 00000000 E0000221 0B010A00 00000100  PE..L...|ò'Z........à..!........
000000E0: 00000200 00000000 CAFE0100 00100000 00200200 00000010 00100000 00020000  ........Êþ...... ...............
00000100: 05000100 00000000 05000100 00000000 00501000 00040000 00000000 03004001  .................P............@.
00000120: 00001000 00100000 00001000 00100000 00000000 10000000 00BD0E00 4B000000  ..........................œ..K...
00000140: <data directories>
```

1. MZ Magic
2. PE Magic
3. DOS String
4. Rich Header
5. Machine
6. Num Sections
7. Timestamp
8. Characteristics
9. Linker Info
10. OS Required
11. SubSystem
12. DllCharacteristics
13. Data Directories

Figure 1: PE header fields considered in the analysis. Here: PE32 variant.

tation points. Based on these we then deduce the position of the file and optional header to subsequently directly access the fields of interest by their offset, as shown in Figure 1.

The results of this analysis are grouped by the individual tests and shown in Table 1. We list results both for individual samples and grouped by families. In cases where samples of a family give a conflicting result, we chose the majority value.

5.1.1 PE Header Availability

We start our evaluation with a group of tests centered around PE header availability. First, we perform `pe_check` and to our surprise, 422 of 446 families (94.62%) pass this test. Looking closer, we notice that we cover 3 more families with `pe_check` than following the usual routine of locating headers by the MZ magic. This would be e.g. the method the popular Python library `pefile.py` follows, whose result we list for comparison.

In total there are only 86 samples in 30 families that do not pass `pe_check`. Manual inspection of these cases result in the following observations: for 39 samples (18 families) we find headerless position independent shellcode, 17 samples (5 families) have a nulled header (determined by size), 16 samples (7 families) directly start with referenced data, 15 samples (5 families) start with an self-constructed Import Address Table (IAT) and 3 samples (2 families) perform an XOR operation over their header (which could technically be recovered). The disparity of 30 versus 24 families (as shown in Table 1) is caused by some families having fewer samples with modified header than without, triggering the majority decision.

For 403 families (90.36%) we also locate one of the following DOS Strings:

- `This program cannot be run in DOS mode`
- `This program must be run under Win32`
- `This program must be run under Win64`

On a side note, we have observed that the DOS string `"This program must be run under Win32/Win64"` seems specifically tied to Borland compilers.

Rich Headers, which can provide additional information in this context [7], are present for 272 (60.99%) families and are covered in Section 5.1.4.

5.1.2 General Characteristics

Next, we focus on a range of general characteristics.

First, we look at architecture required as determined by the PE header machine field. The vast majority of samples and families in *Malpedia* are tracked as 32bit, with occasional 64bit versions. Only GHOLE, a modified CoreImpact version used by threat actor RocketKitten, is currently tracked exclusively as 64bit. We think this number is mostly a result of our current dumping procedure and will shift over time, as we have unpacked 64bit variants or modules for 16 families.

Surprisingly, as much as 26.91% families' core components exist as DLLs, often being staged and loaded by additional code.

With regard to the execution mode as defined by subsystem, a majority of samples have been compiled to use GUI (85.92%) versus console (6.95%). This makes sense, as it has the advantage of being less noticeable by not running in a blocking fashion, spawning an additional command shell window.

A closer look at the minimum OS version required to execute the samples, 92.51% of these values are distributed between version 4.0 (Windows NT) and 5.1 (Windows XP), despite the fact that most of our samples have been observed in 2014 and later. 30 families (7.11%) require a version of Windows Vista or above, with 11 of them having been observed in APT context. Our explanation is that this is a result of precaution by malware authors to have their tools as compatible as possible with the typically unknown target environment. It should also be noted that due to us using only versions up to Windows 7 (which is version 6.1), we do not have dumps included that strictly require a OS version higher than that. On the other hand, we have not observed such malware yet, which would have been noticed through execution failure during dumping.

Another feature that we have analyzed are the security properties that have been activated, namely supporting SafeSEH, being compatible with No-Execute (NX) and environments supporting dynamic rebasing or Address Space Layout Randomization (ASLR). As shown in Table 1, SafeSEH is supported by 73.09% of the families. ASLR with 57.62% is a little more common than NX with 52.47% (family-level). Both features combined are supported by about half (49.10%) of the families.

Sections	Test	Samples True (%)	Samples False (%)	Samples n/a (%)	Families True (%)	Families False (%)	Families n/a (%)
5.1.1	pe_check	1122 (92.88)	86 (7.12)	0 (0.00)	422 (94.62)	24 (5.38)	0 (0.00)
	MZ Magic	1111 (91.97)	97 (8.03)	0 (0.00)	419 (93.95)	27 (6.05)	0 (0.00)
	PE Magic	1114 (92.22)	94 (7.78)	0 (0.00)	419 (93.95)	27 (6.05)	0 (0.00)
	DOS String	1003 (83.03)	205 (16.97)	0 (0.00)	403 (90.36)	43 (9.64)	0 (0.00)
	Rich Header	766 (63.41)	442 (36.59)	0 (0.00)	272 (60.99)	174 (39.01)	0 (0.00)
	pefile-parsable	1111 (91.97)	97 (8.03)	0 (0.00)	419 (93.95)	27 (6.05)	0 (0.00)
5.1.2, 5.1.3	32bit*	1117 (92.47)	5 (0.41)	86 (7.12)	421 (94.39)	1 (0.22)	24 (5.38)
	DLL	341 (28.23)	781 (64.65)	86 (7.12)	120 (26.91)	302 (67.71)	24 (5.38)
	SafeSEH	939 (77.73)	183 (15.15)	86 (7.12)	326 (73.09)	96 (21.52)	24 (5.38)
	NX	529 (43.79)	593 (49.09)	86 (7.12)	234 (52.47)	188 (42.15)	24 (5.38)
	ASLR	663 (54.88)	459 (38.00)	86 (7.12)	257 (57.62)	165 (37.00)	24 (5.38)
	NX+ASLR	475 (39.32)	647 (53.56)	86 (7.12)	219 (49.10)	203 (45.52)	24 (5.38)
	Valid Timestamp	1060 (87.75)	62 (5.13)	86 (7.12)	398 (89.24)	24 (5.38)	24 (5.38)
5.1.5	Export Table	264 (21.85)	858 (71.03)	86 (7.12)	98 (21.97)	324 (72.65)	24 (5.38)
	Import Table	1056 (87.42)	66 (5.46)	86 (7.12)	400 (89.69)	22 (4.93)	24 (5.38)
	Resource Table	610 (50.50)	512 (42.38)	86 (7.12)	293 (65.70)	129 (28.92)	24 (5.38)
	Exception Table	5 (0.41)	1117 (92.47)	86 (7.12)	3 (0.67)	419 (93.95)	24 (5.38)
	Certificate Table	18 (1.49)	1104 (91.39)	86 (7.12)	12 (2.69)	410 (91.93)	24 (5.38)
	Base Relocation Table	864 (71.52)	258 (21.36)	86 (7.12)	310 (69.51)	112 (25.11)	24 (5.38)
	Debug	192 (15.89)	930 (76.99)	86 (7.12)	99 (22.20)	323 (72.42)	24 (5.38)
	Architecture	1 (0.08)	1121 (92.80)	86 (7.12)	0 (0.00)	422 (94.62)	24 (5.38)
	Global Ptr	0 (0.00)	1122 (92.88)	86 (7.12)	0 (0.00)	422 (94.62)	24 (5.38)
	TLS Table	62 (5.13)	1060 (87.75)	86 (7.12)	37 (8.30)	385 (86.32)	24 (5.38)
	Load Config Table	317 (26.24)	805 (66.64)	86 (7.12)	141 (31.61)	281 (63.00)	24 (5.38)
	Bound Import	6 (0.50)	1116 (92.38)	86 (7.12)	5 (1.12)	417 (93.50)	24 (5.38)
	IAT	925 (76.57)	197 (16.31)	86 (7.12)	341 (76.46)	81 (18.16)	24 (5.38)
	Delay Import Descriptor	39 (3.23)	1083 (89.65)	86 (7.12)	19 (4.26)	403 (90.36)	24 (5.38)
	CLR Runtime Header	83 (6.87)	1039 (86.01)	86 (7.12)	64 (14.35)	358 (80.27)	24 (5.38)
	Reserved	0 (0.00)	1122 (92.88)	86 (7.12)	0 (0.00)	422 (94.62)	24 (5.38)

Table 1: Summary of PE header field analysis, on sample and family-level of aggregation ($*$: False indicates 64bit).

With respect to the number of sections found, 84.31% of the samples have either 3, 4, or 5 sections. These correspond strongly to the most common section names found: `.text` (89.22%), `.data` (80.66%), `.reloc` (78.43%), `.rdata` (70.68%), and `.rsrc` (55.35%).

5.1.3 Timestamp Information

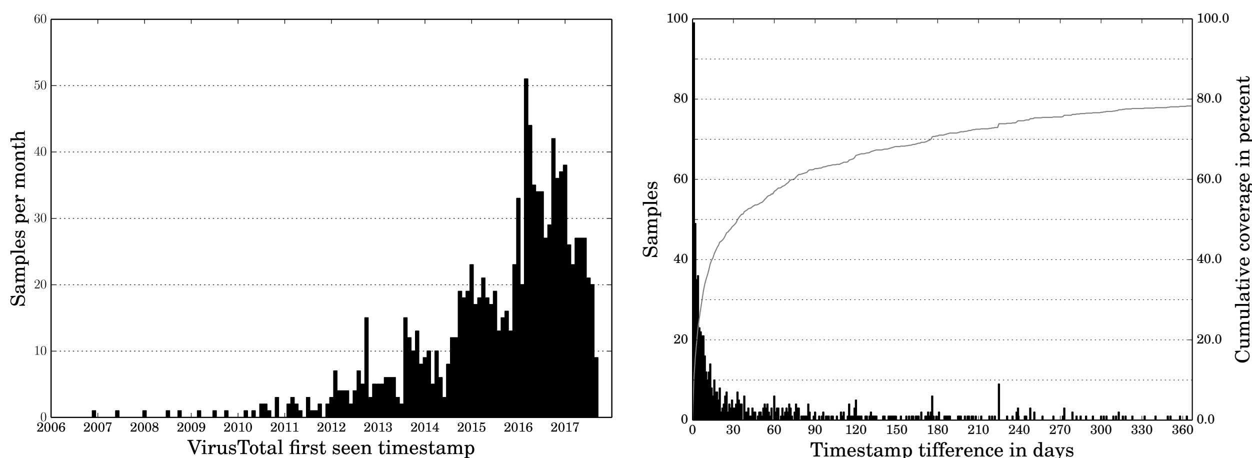
The next aspect we focus on is the temporal information available to us. We first measure the age of the current *Malpedia* corpus by looking up the date when samples have been first seen on VirusTotal (denoted as T_{VT}) and then compare this information with the compilation timestamp as found in the PE header (T_{PE}).

Out of the 1208 files used in this evaluation, 1173 (97.10%) have a T_{VT} value. As shown in Figure 2a, most of the samples have been seen first within the last 4 years, as the values between 2013 and 2017 make up 91.65%. While having less priority than keeping up with recent developments, we hope to close the gap of potentially relevant families of the past over time.

With this data at hand, it is now interesting to put the PE header's compilation timestamp into perspective. After filtering out 86 samples that failed `pe_check`, we can filter out another 62 samples that have a value of either null or the default Delphi timestamp (1992-06-19T22:22:17). From those, we remove another 30 samples where no T_{VT} is available. Next, we apply two plausibility checks. First, we check if the timestamp difference ($T_{VT} - T_{PE}$) is negative, which would mean the sample has been contradictingly observed on VT before it was compiled, which is the case for 38 samples. Second, using the range of 2006-12-11 (earliest T_{VT}, a sample of Gozi) and 2017-10-11 (timestamp of our corpus snapshot) we identify 43 samples (19 families) that fall outside of this range. Apart from all of them having a timestamp difference ($T_{VT} - T_{PE}$) of 7 years or more, we did contextual searches on the families to ensure that those timestamps are in fact the result of manipulations and that we can safely exclude them.

This leaves us with 949 timestamp pairs of T_{VT} and T_{PE}, belonging to 367 families (82.29%), that are potentially plausible. We have plotted their timestamp differences (limited to one year, covering 314 families) in Figure 2b. 10.43% samples have been seen on VT within one day, 32.24% within 7 days, 48.78% within the first month, and 78.30% within the first year after their compilation timestamp. Given that we did not aim at finding the first (packed) sample for an unpacked equivalent, we believe that this still shows that PE compilation timestamps are very often at least in a meaningful distance to their first seen date, relying on VirusTotal as a reference.

We have also looked closer at the remaining 206 cases in which the difference between VT First Seen and the PE timestamp is longer than one year.



(a) *Malpedia* corpus age, as indicated by VirusTotal first seen dates (T_{VT}) of 1173 (97.10%) samples.

(b) Difference between VirusTotal first seen date and compilation timestamp ($T_{VT} - T_{PE}$), 743 of 949 (78.30%) values shown (less or equal to one year).

Figure 2: Temporal information about the corpus.

We attribute 75 samples (47 families) more or less confidently to APT background. In these cases, files are sometimes "published" to VT alongside reports. Others are related to leaks, e.g. ShadowBrokers (DoublePulsar, Fanny, Oddjob, ...) and exhibit a drastically aged timestamp.

Another 59 samples (12 families) come from families where it is publicly known or otherwise plausible that they are based on builders/configurators (Zeus, Citadel, Ramnit, GPCode, RATs like PoisonIvy, ...). This allows for some of them to be used or appear years later, hence having comparatively old compilation timestamps.

For 26 samples (5 families) we can confirm forgery that is also found across other header fields and assume it is plausible that their PE compilation timestamps have been modified as well. These overlap with families that had samples sorted out earlier when reducing timestamp pairs to plausible ones (e.g. Locky, Necurs).

The remaining 46 samples (36 families) are not trivially explainable. Their timestamp offset may be the result of a variety of reasons, e.g. the system where they were compiled on having massive clock-drift, the samples having remained undiscovered/unsubmitted for a long time, or the timestamps being forged as well.

5.1.4 Compiler/Linker Information

A very interesting aspect in the context of the PE header is actually the Major/Minor Linker information field, as it may offer some insight into the toolkit preferences of malware authors. In order to determine these values confidently, we adapted the signature database of Detect-It-Easy (DIE) [10]. An overview of our results is shown in Figure 3.

After de-duplicating on family-level, we note 513 data points for 446 families. Out of 194 families where more than one sample is available, only 56 families (28.87%) have samples being compiled with more than one compiler/linker version. Malware authors seem to have a tendency to stick with their tool chains, as changes here only occurred in case of full rewrites of their project, for example with GPCode and Sakula RAT, which were both translated from C to Assembler. In all other cases, we only observed authors updating to a more modern version within the tool chain (2x within MinGW/gcc and 49x within MSVC).

No information about the tool chains used could be inferred in 48 (9.36%) cases, where no header was available, or the version field was not plausible (i.e. being nulled, obviously forged, or otherwise a result of header fragmentation).

We know of only a single family being written and compiled in Go: AthenaGo. Also, only for a single family GoAsm with Golink 0.40 was utilized (Sakula RAT). For 3 families, we inferred that they were directly built using Flat Assembler (FASM), skipping the linker step.

Pelles C' PoLink was used in 3 (0.58%) cases and Microsoft Assembler's (MASM) MIL was used in 12 cases (2.34%). Please note, that both PoLink and MIL may also have been used in conjunction with FASM, as FASM optionally allows using an external linker such as both PoLink and MIL.

MinGW/gcc account for 17 cases (3.31%), being split over 7 versions.

Borland compilers were used in 29 (5.65%) cases, with the majority of them being Delphi (27). Delphi Turbo Linker 2.25 and gcc 2.25 share the same version number but are easily distinguished through the respective characteristics found in the code.

The vast majority constitute families that were compiled using various versions of Microsoft Visual Studio. They are tied to 401 (78.17%) data points over 10 versions. Looking at the distribution, we can make two observations. First, a disproportional number of families are seemingly compiled with the outdated VC6, released in 1998. Possible explanations for

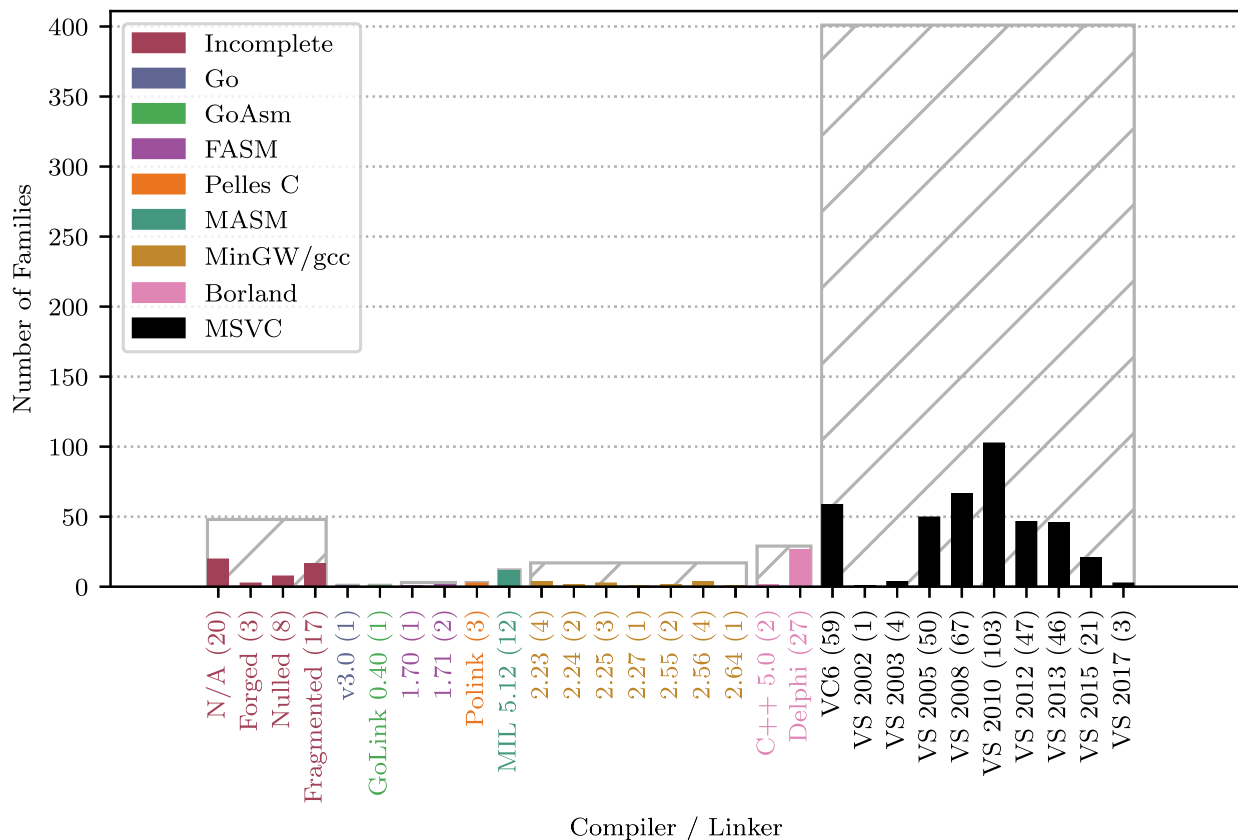


Figure 3: Distribution of compiler and linker versions, counted uniquely per family. Compiler groups (same color) are further divided into versions. The gray boxes enclosing groups indicate the sum of of values for that group. The vast majority of authors apparently use Visual Studio to build their code, with Visual Studio 2010 being the most common version.

this are that VC6 links statically against `msvcrt.dll` per default, avoiding dependencies. On top of this, it is linking against the last DLL version (which has the status of a system DLL) before Microsoft's policy change of requiring developers to ship the appropriate `msvcrt.dll` version along their code (which lead to a situation commonly referred to as "DLL Hell").

Second, all other MSVC versions are distributed around VS2010, slightly leaning towards older versions. One explanation for this may be that many programmers stick to an environment that they are familiar with, in the sense of "never touch a running system".

Next, we can incorporate the Rich Header [7] as an additional measure to estimate the plausibility of the information given in the linker fields. For the 766 samples in which a Rich Header is present (cp. Table 1), we find 735 (95.95%) of them having a linker field number corresponding to a MSVC compiler. Drilling down on these, at least 721 (98.10%) also have a version-matching PID entry in the Rich Header, showing that this information is reliable in almost all cases. We only find deviations for families, where we earlier noted that other header fields were likely forged (e.g. Necurs, Locky, and Lurk).

5.1.5 Data Directories

With regard to the presence of data directories, we notice that only Import Table (89.69%) and IAT (76.45%), Base Relocation Table (69.51%), and Resource Table (65.70%) appear in more than 50% of the families.

Out of the 192 samples having a Debug Directory, 163 (84.90%) contain references to PDB files, which are handled more in-depth in Section 5.2.2.

The presence of a Load Config Table indicates the presence of a directory of known, safe Structured Exception Handlers (SafeSEH, cp. Section 5.1.2), which is the case for almost a third of the families (31.61%). Note that way more families (326 or 73.09%) have the SafeSEH field activated but may not include such a directory for a number of reasons, such as not using Structured Exception Handlers, linking against modules not supporting SafeSEH, or being subsystem:console applications [11].

An Export Table is found for 264 samples of 98 (21.97%) of the families. An interesting field to look at in conjunction with Export Tables is the DLL field. Out of 341 samples being DLLs, only 239 (70.09%) export functions, meaning that the ones without either have a fully functional `DllMain` routine (which is officially discouraged by Microsoft) or get controlled through implicitly known function offsets. On the other hand, 25 out of 781 samples (3.20%) being executables have an

	Functions	BBlocks	Instructions	Function Calls
min	3.00	25.00	30.00	2.00
25%	181.25	1437.25	9362.75	446.54
50%	438.50	4261.83	23833.62	1377.08
75%	1107.00	9765.78	54632.33	3593.75
max	26360.00	337386.50	1848787.00	113008.50
mean	1300.92	10954.50	63390.37	4582.13

(a) Control Flow Graph Statistics.

	Functions	BBlocks	Instructions	Function Calls
Functions	1.000	0.884	0.910	0.946
BBlocks	0.884	1.000	0.992	0.941
Instructions	0.910	0.992	1.000	0.960
Function Calls	0.946	0.941	0.960	1.000

(b) Pearson Correlation Coefficient (PCC).

Table 2: Results of the cursory code evaluation across 382 families, averaged values per family (64 .NET families excluded).

Export Table, which is possible but unusual behaviour as well.

The 64 families (14.35%) having a CLR Runtime Header are all .NET based and identical with those excluded from the Code and Windows API usage analysis (cp. Sections 5.2 and 5.3).

TLS callback tables are available for 37 (8.30%) of the families and 19 (4.26%) use Delay-Load Imports. Only 12 (2.69%) of the families are signed, with 8 of them being connected to APT activity.

We have observed a Bound Import Table only for families written in MS VisualBasic (5 families, 1.12%) and Exception Tables (0.67% of families) are only found in families that also have 64bit binaries.

The Architecture and Reserved Directories are expectedly zero, in accord with the PE/COFF Specification [12], and a Global Ptr Table is also found in none of the families.

5.2 Code Analysis

For code analysis, we use our own recursive disassembler called SMDA, which is based on Capstone [13] and specifically optimized for the disassembly of arbitrary code buffers, i.e. it is suited for dealing with memory dumps as found in *Malpedia*. It implements a semantically-aware method for Function Entry Point (FEP) localization, oriented on the approach presented by Andriesse et al. [14]. Using a slightly more aggressive decision process for function identification, it achieves better coverage than comparable tools such as IDA Pro or radare2, at the cost of a slightly higher false positive rate.

Because of the complexity and wealth of information inferable from code, we limit ourselves to only a few aspects covered with cursory analysis in this paper. Instead, we will follow up with an in-depth evaluation of our method and an analysis of the malware code base in a consecutive publication.

5.2.1 Cursory Control Flow Graph Analysis

Using SMDA, we have disassembled all samples and extracted key indicators with regard to their Control Flow Graphs (CFG). The results are shown in Table 2a. Note that .NET families have been excluded and the values have been previously aggregated on family-level to only show representative values for x86/x64 machine code.

With regard to the number of functions, the smallest type of malware found are typically downloaders without any other functionality. In our corpus, examples for these minimalistic families are TinyLoader (3 functions), Dorshel (3 functions), Cabart (9 functions), StegoLoader (13 functions), Hamweq (14 functions), and Harnig (15 functions). Generally, the families below the first quartile (181 functions) are reigned by downloaders, modularized RAT servers, and purified ransomware. They also include some malware related to spam (e.g. Asprox, Matsnu, Pushdo).

The families between the first and third quartile (181 to 1107 functions) often include several of the well-known behavioral aspects tied to malware, such as information stealing, enabling financial theft, spam, DDoS, downloading and ransomware.

The families beyond the third quartile typically fall in either of two categories. First, some of them simply bring a vast range of functionality to the table. Second, the others are massively inflated through the extensive use of statically linked third-party libraries, including OpenSSL, Boost C++, or various Delphi modules. In fact, out of the 26 families identified as being written in Delphi, 21 fall above the third quartile.

Looking at the other values shown in Table 2a, it stands out that they very strongly correlate with each other across all quartiles, further underlined by the Pearson Correlation Coefficients shown in Table 2b. Overall, the ratio of basic blocks to functions is situated between 8 and 10, the ratio of instructions to functions is at around 49 and 54, and the ratio of function calls to functions between 2.4 and 3.2. Similarly, basic blocks consist on average of 5.5 to 6.2 instructions.

5.2.2 Program Database Information (PDB)

Microsoft [15] has defined a proprietary standard for creating meta information during compilation that can be used to enrich debugging sessions. A fragment occasionally found in malware are path specifications to the corresponding program database (PDB) files.

As already mentioned in Section 5.1.5, 163 samples belonging to 111 families (24.98%) in our data set contain references to PDB files. Looking closer at the paths of these PDB files, we can identify 32 somewhat expressive user names that are not as generic as "User", "Administrator", or the like. Furthermore, we find 49 references that can be interpreted as project names as chosen by the authors and 40 of them directly correspond to a name or alias that this family is

referred to. This shows that in case where information of the author's own naming is available, that this is often adopted as a reference by malware researchers.

5.3 Windows API Usage Analysis

In this section, we study how malware families make use of the Windows API. Generally, from a software analysis point of view, a program's interaction with an API can reveal a lot of insight into the behaviour of code. For this very reason, the inspection of API interactions is often an essential cornerstone when conducting detailed malware analysis, as it may be used as a pointer to the code regions responsible for e.g. persistence, networking, or other functional aspects of interest. We first lay out our methodology for the analysis and give a short introduction to ApiScout [8] including a showcase of its accuracy on a small selection of benign Microsoft binaries. Next, we apply ApiScout to the current data set and evaluate aspects such as the availability of Windows API import information in malware memory dumps as well as frequencies of occurrence for DLLs and APIs across malware families.

5.3.1 ApiScout: API Information Recovery from Memory Dumps

As explained in Section 3.3, we create memory dumps for all malware samples contained in *Malpedia* using a small set of reference VMs. Since we control the environment dumps are taken from, we can exploit the associated knowledge in our favour, e.g. by inventorizing all DLLs present on the system. Using this data, we can infer a complete view of the structure of the Windows API on a process-level perspective. It is also of importance that DLLs are usually loaded at the same base address across all processes [16] and this inventorization procedure results in a listing of all offsets of exports they provide.

In consequence, we can derive the actual addresses at which API functions will be available and through which they will be referenced in the loaded processes of programs, including malware. Dumping of malware has another benefit: It yields us a snapshot of the (unpacked) malware in memory during its execution. This means we may be able to observe API entry points dynamically loaded [17] and not just those referenced through the regular method of using the PE header's import tables. Using runtime VM snapshots also solves any issues potentially arising from dynamic rebasing through ASLR [16] since we only need to ensure that every DLL has been loaded once in order to have it assigned its randomized offset. We can then index these offsets along the base addresses during the inventorization process. The inventorization of our reference VMs results in 57,315 exports from 134 DLLs for Windows XP SP3 and 105,765 exports from 382 DLLs for Windows 7 SP1 64bit. Removing redundancies, we end up with 59,366 unique API functions in both systems combined. This lower number is explained by Windows 7 64bit containing variants of both 32bit and 64bit DLLs, needed for compatibility reasons.

Given a database of all exports of DLLs present in the system, we can now perform a lookup for arbitrary DWORDs/QWORDs and check if they potentially match an inventorized API address. This is the core idea of ApiScout and has been implemented in a library provided on GitHub [8]. For ApiScout, we avoid making structural assumptions about input buffers presented and simply scan every DWORD/QWORD linearly for potential API address identity. This way, we can handle shellcode and mapped PE files the same way.

To remove potential false positives (FPs), we include a parameter to optionally filter to import references that appear in groups (as is usually the case for regular structures, such as an IAT). If set, the filter will remove all import references that do not have a neighbour within a certain range. In the following, we use 32 bytes as filter width.

We performed a small test evaluation to measure the accuracy of ApiScout. We took 15 benign system binaries as found in Windows and dumped their memory during execution. Next, we parsed their import and delay import tables to be used as ground truth. In this scenario, ApiScout already achieved an F-Score of 0.991 and 0.995 with the neighbour filter activated. Manual inspection of the deviations reveals that ApiScout finds all entries in the respective import address tables (IATs) resulting in no recorded false negatives.

False Positives are found for only three dumps: `explorer.exe`, `mmc.exe`, and `cmd.exe`. To our surprise, all three of these programs make use of dynamic API loading during their runtime via `kernel32!GetProcAddress`. As this mechanism works around the Import Table, it explains why these imported API functions are not covered by the ground truth, which is based on import tables exclusively. This leads to a total of 5,367 correctly identified API functions and technically 51 FPs, which however are also entirely dynamic imports used by the respective programs and identified by ApiScout. This shows that ApiScout can be used to identify imports of Windows API functions as found in dumps of programs with high precision. However, note that ApiScout does not check if the offsets discovered are referenced by any code and therefore still should be taken as an approximation of the actual API interactions.

5.3.2 API Information Availability

When analyzing malicious software, the interaction of code with the Windows API often serves as an important cornerstone. Looking at how many malware families may have information on their Windows API usage available, we can quickly exclude 64 out of 446 families (14.35%) which have been previously identified (cp. Section 5.2) being written using the .NET framework, because in this case the import model can not be easily correlated with traditional Windows API usage.

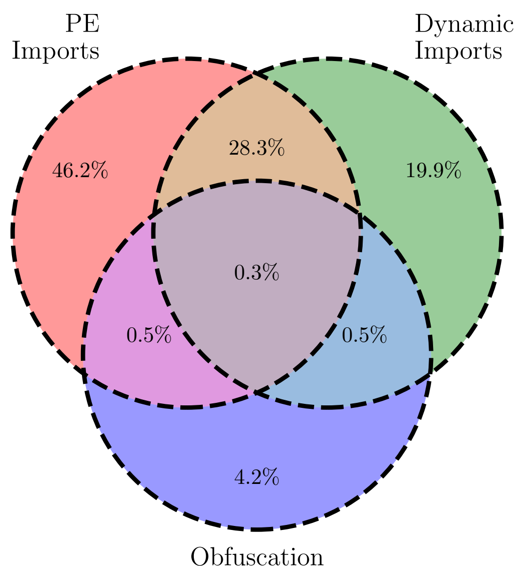


Figure 4: Distribution of Import Style for Windows APIs across 382 families (64 .NET families excluded).

However, we can investigate the remaining 382 families and check if they have API information available for analysis identifiable with ApiScout. Additionally, we can check for the regularity of references to the Windows API or if interactions happen in a concealed way, i.e. using obfuscation [18]. In order to distinguish, we have defined three classes of *Import Styles*:

- Static (i.e. regular) imports using the PE header's import table.
- Dynamic imports of exact WinAPI function addresses that are cached within the memory occupied by the malware and hence are still detectable by ApiScout.
- Custom import schemes that we label as "obfuscation" and explain in the following.

The results of this analysis are shown in Figure 4. Please note that we did not investigate every API obfuscation scheme in full detail and therefore did not extract the actual collection of APIs used by these families. It is also likely that we missed API obfuscation schemes due to not employing in-depth code analysis methods in this evaluation, meaning that the actual fraction may be higher than displayed.

First, we observe that almost half (46.2%) of the families collected use static imports exclusively. We believe that this import style similar to most regular programs works sufficiently well for many malware authors as it allows them to get along without adding potentially sophisticated methods that carry the risk of failure. It may also reduce the detection surface as their malware will not have a suspiciously low number of imports and will show fewer dynamic loading (e.g. LoadLibrary/GetProcAddress), which is usually recorded during the execution in dynamic analysis systems such as sandboxes.

Second, almost another half (49.0%) of families uses dynamically loaded imports, optionally combined with static imports or obfuscation schemes. In the cases where the dynamic loading is combined with another method, it is interesting to dissect the parts of the Windows API that malware authors deem worthy to treat in a specific way. Looking at the overlap between families using static and dynamic imports, we identify 108 families using both methods. For these, we have found between 1 and 263 dynamically imported API functions with a median of 21 and an average of 42.81 (or 24.26% of API functions these families import overall). Here, it seems that malware authors indeed try to hide suspicious activity, as about 40% of the exclusively dynamically loaded API functions correspond typically to behaviors such as process control, process injection, and network communication. As a side note, about 3.45% of these API functions have been redundantly imported by both the static and dynamic method.

Finally, we found at least 5.5% of the families using obfuscation schemes for which API function information is not recoverable with ApiScout. Exemplary, we have identified the following obfuscation methods being used. 8 families are resolving APIs every time they intend to use them (Shifu, StegoLoader, ...). Another 7 families store their imports in a separate, dedicated memory segment on the heap (Cryptowall, SolarBot, ...), while 2 families manage their imports on the stack (PoisonIvy and Dorshel). One family each build their own jump-table instead of using an API function offset table (Andromeda), add 5 bytes upon the API address likely to avoid hooks (Chthonic), build an on-demand offset table (Dridex), or store imports XORed with a static key (Qadars).

In conclusion, we note that for a majority of families information about interactions with the Windows API are easily recoverable and happen naturally through direct references to the function offsets within the respective DLLs.

5.3.3 DLL and API Usage Frequencies

Another interesting viewpoint on Windows API interaction are the frequencies with which different DLLs and APIs are used across all families.

Table 4 lists the general characteristics of Windows API usage. Zero API and DLL imports correlate with families using pure obfuscation schemes.

	API Functions	DLLs
min	0.00	0.00
25%	84.08	5.69
50%	122.25	8.00
75%	191.71	10.74
max	592.00	24.00
mean	150.26	8.32
Total Observed	3693.00	59.00

Table 4: Occurrence frequencies per family (64 .NET families excluded).

With 592, The most APIs are used by DarkComet [19], a fully fledged RAT written in Delphi. With 24, the most DLLs are used by ThumbThief [20], whose

	API	Occurrences	DLL	Occurrences
1	kernel32.dll!Sleep	330 (86.39%)	kernel32.dll	363 (95.03%)
2	kernel32.dll!CloseHandle	326 (85.34%)	ntdll.dll	352 (92.15%)
3	kernel32.dll!GetModuleHandle	323 (84.55%)	advapi32.dll	302 (79.06%)
4	kernel32.dll!CreateFile	314 (82.20%)	user32.dll	293 (76.70%)
5	kernel32.dll!WriteFile	312 (81.68%)	shell32.dll	220 (57.59%)
6	kernel32.dll!GetProcAddress	312 (81.68%)	ws2_32.dll	206 (53.93%)
7	kernel32.dll!GetModuleFileName	307 (80.37%)	wininet.dll	161 (42.15%)
8	kernel32.dll!LoadLibrary	303 (79.32%)	ole32.dll	151 (39.53%)
9	kernel32.dll!ExitProcess	293 (76.70%)	shlwapi.dll	140 (36.65%)
10	kernel32.dll!ReadFile	285 (74.61%)	oleaut32.dll	110 (28.80%)
11	kernel32.dll!GetCurrentProcess	280 (73.30%)	gdi32.dll	98 (25.65%)
12	kernel32.dll!GetTickCount	279 (73.04%)	msvcrt.dll	84 (21.99%)
13	ntdll.dll!RtlGetLastWin32Error	274 (71.73%)	crypt32.dll	68 (17.80%)
14	ntdll.dll!RtlAllocateHeap	261 (68.32%)	iphlpapi.dll	50 (13.09%)
15	kernel32.dll!WideCharToMultiByte	261 (68.32%)	psapi.dll	48 (12.57%)
16	kernel32.dll!CreateThread	257 (67.28%)	netapi32.dll	43 (11.26%)
17	kernel32.dll!MultiByteToWideChar	252 (65.97%)	urlmon.dll	40 (10.47%)
18	kernel32.dll!TerminateProcess	246 (64.40%)	version.dll	38 (9.95%)
19	kernel32.dll!GetCurrentProcessId	244 (63.87%)	mpr.dll	33 (8.64%)
20	ntdll.dll!RtlEnterCriticalSection	241 (63.09%)	winhttp.dll	28 (7.33%)

Table 3: Most common APIs and DLLs across all families (excluding .NET)

loader includes a variety of functionality for fingerprinting the system it is attacking. On average we observed 8 DLLs providing access to around 150 API functions per malware family. It is also notable that we found a total of 3,693 Windows API functions being used, out of 59,366 unique API functions tracked in the ApiScout databases (WinXP and Win7 combined).

The most common APIs and DLLs are listed in Table 3. Please note that we have grouped the respective ANSI and Unicode variants of API functions (such as LoadLibraryA and LoadLibraryW) for this table into single representatives, reducing the number of unique API functions from 3,693 to 3,316.

To our surprise, `kernel32.dll!Sleep` turned out to be the most common API function used across all families. Our interpretation for this is that malware, considered as a form of a somewhat autonomously acting program, needs to temporally organize its behaviour. In that sense, `Sleep` offers applicability in a multitude of cases, such as controlling communication frequencies (C&C), ensuring persistence (e.g. registry and file-system lookups), or delaying execution (e.g. as an anti-analysis method).

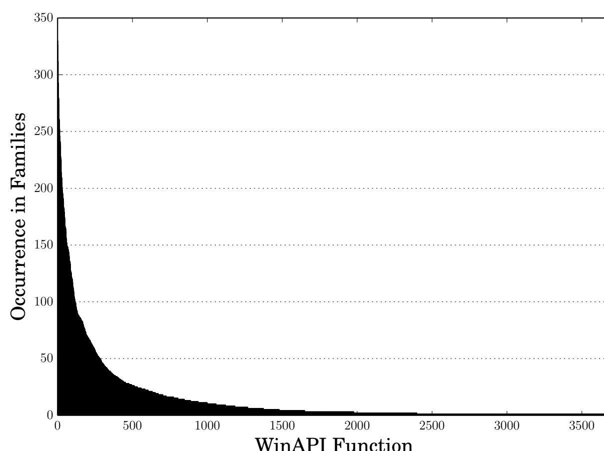


Figure 5: Occurrence frequency of WinAPI functions with regard to number of families they appear in.

Many other API functions are centered around the topic of execution control. This includes aspects such as dynamic imports (`GetModuleHandle`, `GetProcAddress`, `LoadLibrary`), handles (`CloseHandle`, `GetCurrentProcess`), error handling (`GetLastWin32Error`), or self-termination (`ExitProcess`, `TerminateProcess`). As expected, file-system interaction (`CreateFile`, `WriteFile`, `ReadFile`) is also very commonly found. It is notable that not a single network-related API function is within the top 20 list. We believe that this is explained by the freedom of implementation offered by the Windows API and also reflected by the most commonly used DLLs. Assuming a malware author wants his malware to communicate with a C&C server using the HTTP protocol. They now can choose between using `wininet.dll` for access to high-level functions, use the more service-oriented `winhttp.dll`, or opt for `ws2_32.dll` and re-implement simplified HTTP handling themselves.

This observation carries on into the fact that the concrete composition of API functions used by a malware family seems to be heavily characteristic for that family. Figure 5 shows for every of the 3,693 Windows API functions in how many malware families they appear.

Discounting families using obfuscation and .NET (360 remain), only `kernel32.dll!Sleep` and `kernel32.dll!CloseHandle` appear in more than 90% of families. Furthermore, only 44 API functions appear in more than 50% of the families. Taking the average number of API functions per family from Table 4, the API function residing in position 150 would be `kernel32.dll!WaitForMultipleObjects`, which is present in only 23.89% of the families. Looking the other way around, a massive 3,320 (89.90%) of all observed API calls appear in less than 10% of the families. We believe that this observation on the disparity of API compositions per family supports the effectiveness of the general idea behind approaches like ImpHash [21] and ImpFuzzy [22].

Because looking at individual API functions has limited expressiveness, we have decided to define 12 context groups with 93 sub-contexts that can be used to group API functions by their potential field of use. We have categorized 3817 of the 4239 (90.04%) API functions found in all samples, which cover 99.41% of individual API function appearances of our test data set. This functionality has also been integrated into ApiScout and will be updated as needed.

Our results for evaluating API function frequencies using context groups are listed in Table 5.

	Context	Occurrences
1	Execution Control	367 (96.07%)
2	Memory	361 (94.50%)
3	File System	353 (92.41%)
4	System	353 (92.41%)
5	String	352 (92.15%)
6	Network	312 (81.68%)
7	Time	304 (79.58%)
8	Registry	264 (69.11%)
9	GUI	248 (64.92%)
10	Other	210 (54.97%)
11	Device	193 (50.52%)
12	Crypto	175 (45.81%)

Table 5: Occurence frequencies of API context groups by family (excluding .NET).

As can be seen, only about four in five families exert network functionality. This is easily explained with families that require no immediate control to execute their intentional behaviour, or that are instrumented by other families. Examples for this are purely destructive wipers or information collectors that are dropped by other malware. Another example is ransomware in which the criminals require their victims to contact them actively via email instead of establishing a communication channel themselves (e.g. for sending back encryption keys). A bit surprisingly, almost equally as many families interact with API functions providing time information, where the concrete functions used to query are partitioned into `kernel32.dll!GetSystemTimeAsFileTime` (179), `kernel32.dll!GetLocalTime` (114), and `kernel32.dll!GetSystemTime` (86). The seemingly low number of families using Windows API functions related to Cryptography is explained with a likely high dark figure of authors using external code for popular algorithms such as CRC32, RC4, and AES instead of relying on the Windows API.

6 Related Work

There have been some efforts to collect and organize malware in the past. Nativ et al. [23] have been collecting and providing malware samples organized by families in their project "theZoo". Freyssinet [24] studied the malware ecosystem in detail, primarily focusing on botnets and collected meta data information on 412 malware families, organized and published in [25]. The Malware Wiki [26] is another extensive resource collecting meta data and high level descriptions for various malware families. MITRE organizes the Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK) [27] knowledge base, focusing primarily on APT activity and tying behaviours to actor groups and malware families. The collective of MalwareHunterTeam run the web service ID Ransomware [28], focusing on the identification of ransomware based on encrypted files and ransom notes. They track 497 distinct variants. *Malpedia* already has significant overlap with all of the above collections and we plan to cover as many as possible of the families contained in them with samples in the future. In a preservatory fashion, Hypponen provides a collection of 86 families of 1980s and 1990s malware in the Malware Museum [29].

Guidelines for malware naming schemes have been proposed by e.g. CARO [30] in 1991 and MITRE [31] in 2006. Even these early works already point out the tendency of introducing synonyms for malware family names instead of agreeing on unambiguous identifiers.

Sebastián et al. [32] experimented with the consistency of AV detection labels, noting significant noise that they addressed with their tool AVClass. They also emphasize that analysts have a need for accurate malware identification. Lever et al. [33] recently conducted a large scale analysis involving 26.8 million malware samples, primarily focusing on malware traffic. Using AVClass, they identified 3,834 clusters of families with more than 10 samples within their data set. This further supports our claim that the space of families and versions is way smaller than the number of packed samples. Ye et al. [34] recently provided a comprehensive overview of works that propose malware detection techniques using data mining. The collection of features extracted from the surveyed works is also highly compatible with *Malpedia*.

Belaoued et al. [35] extracted Windows API function frequencies from a selection of 50 malware samples for malware detection. Their API function frequency table overlaps in 50% with our results. Zwanger et al. [9] conducted an analysis of Windows API function call distribution from a kernel-mode perspective. They showed that malware and benign drivers expose discriminable characteristics in their API usage.

Rossow et al. [1] described best practices for designing malware experiments, surveying related work for their conformance with these requirements. Their work has also heavily inspired decisions taken in this work.

7 Conclusion

In this paper, we addressed the continuous lack of quality data suitable for static malware analysis. First, we defined requirements for a such a malware corpus tailored for static analysis. We next presented our efforts for a vetted curation and inventorization platform called *Malpedia*, including a baseline data set of more than 600 malware families.

To show the usefulness of the data set, we performed a comprehensive comparative analysis of structural features extracted from 446 families of cleanly labeled Windows malware, primarily focusing on PE header characteristics and Windows API usage.

Our key findings are the following. Packers mostly serve just as an initial barrier against detection and the majority of unpacked samples are quite well-formed and can be conveniently treated with methods of static analysis. The information extracted even with just cursory methods draws a consistent picture and gives an interesting insight in preferences and choices of malware authors. We firmly believe that the number of unpacked samples required to expressively represent the malware landscape interpreted as families and versions is many orders of magnitude smaller than the number of packed samples found in the wild. We think that the experiments conducted in this work demonstrate that *Malpedia* can serve as a solid foundation for various future research activities. By responsively publishing this data set through our platform for free, we hope it will contribute as a reference for identification and labeling in analysis processing chains or serve as a starting point for more in-depth studies requiring a significant number of malware families.

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 as Slavo Greminger for their valuable feedback.

Author details

Daniel Plohmann

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

Martin Clauß

Fraunhofer FKIE
Zanderstr. 5, 53177 Bonn
martin.clauss@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] 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), San Francisco, CA*, 2012.

[2] AV-Test GmbH, "Malware Statistics," October 2017. Tracking website by AV-Test: https://www.av-test.org/en/statistics/malware/.

[3] T. Barabosch, N. Bergmann, A. Dombeck, and E. Padilla, "Quincy: Detecting host-based code injection attacks in memory dumps," in *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Bonn, Germany*, 2017.

[4] T. Barabosch, S. Eschweiler, and E. Gerhards-Padilla, "Bee master: Detecting host-based code injection attacks," in *Proceedings of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), London, UK*, 2014.

[5] FIRST Traffic Light Protocol Special Interest Group, "TRAFFIC LIGHT PROTOCOL (TLP)." FIRST Standards Definitions and Usage Guidance: https://first.org/tlp/.

[6] C. Wagner, A. Dulaunoy, G. Wagener, and A. Iklody, "Misp: The design and implementation of a collaborative threat intelligence sharing platform," in *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security*, pp. 49–56, ACM, 2016.

[7] G. Webster, B. Kolosnjaji, C. von Pentz, J. Kirsch, Z. Hanif, A. Zarras, and C. Eckert, "Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage," in *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), Bonn, Germany*, 2017.

[8] D. Plohmann, "ApiScout: Painless Windows API information recovery," April 2017. Blog post for ByteAtlas: http://byte-atlas.blogspot.de/2017/04/apiscout.html.

[9] 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.

[10] Horsicq, "Detect-It-Easy," 2014. GitHub Repository: https://github.com/horsicq/Detect-It-Easy/.

[11] Microsoft, "/SAFESEH (Image has Safe Exception Handlers)," tech. rep., Microsoft, 2017. MSDN Article: https://msdn.microsoft.com/en-us/library/9a89h429(v=vs.110).aspx.

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

[13] N. A. Quynh, "Capstone disassembly engine." http://www.capstone-engine.org/.

[14] D. Andriesse, J. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P), Paris, France*, 2017.

[15] Microsoft, "Debug Interface Access SDK," 2015. MSDN Article: https://msdn.microsoft.com/en-us/library/x93ctkx8.aspx.

[16] M. Russinovich and D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition*. Microsoft Press, 5th ed., 2009.

[17] M. Galkovsky, "DLLs the Dynamic Way," November 1999. Article for MSDN: https://msdn.microsoft.com/en-us/library/ms810279.aspx.

[18] M. Suenaga, "A Museum of API Obfuscation on Win32," tech. rep., Symantec, 2009.

[19] B. Farinholt, M. Rezaeirad, P. Pearce, H. Dharmdasani, H. Yin, S. Le Blond, D. McCoy, and K. Levchenko, "To catch a ratter: Monitoring the behavior of amateur darkcomet rat operators in the wild," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P), San Jose, CA*, 2017.

[20] T. Gardoň, "New self-protecting USB trojan able to avoid detection," March 2016. Blog post for ESET: https://www.welivesecurity.com/2016/03/23/new-self-protecting-usb-trojan-able-to-avoid-detection/.

[21] FireEye, "Tracking Malware with Import Hashing," January 2014. Blog post for FireEye: https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html.

[22] S. Tomonaga, "Malware Clustering using impfuzzy and Network Analysis," March 2017. Blog post for JPCERT/CC: http://blog.jpcert.or.jp/2017/03/malware-clustering-using-impfuzzy-and-network-analysis---impfuzzy-for-neo4j-.html.

[23] Y. Nativ, L. Ludar, and S. Shalev, "theZoo," 2014. GitHub Repository: https://github.com/ytisf/theZoo.

[24] E. Freyssinet, *Lutte contre les botnets : analyse et stratégie*. PhD thesis, Université Pierre et Marie Curie - Paris VI, 2015.

[25] E. Freyssinet, "Botnets.fr," 2011. Wiki: https://www.botnets.fr/wiki/Main_Page.

[26] Various, "Malware Wiki," 2009. Wiki: http://malware.wikia.com/wiki/Main_Page.

[27] MITRE, "Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK)," 2015. Wiki: https://attack.mitre.org/wiki/Main_Page.

[28] MalwareHunterTeam, "ID Ransomware," April 2016. WebService: https://id-ransomware.malwarehunterteam.com/index.php.

[29] M. Hypponen, "Malware Museum," February 2016. Archive: https://archive.org/details/malwaremuseum.

[30] F. Skulason, A. Solomon, and V. Bontchev, "A new virus naming convention," 1991. Article by CARO: http://www.caro.org/articles/naming.html.

[31] CME Editorial Board, "The Common Malware Enumeration (CME)," November 2006. Article by CARO: https://cme.mitre.org/about/faqs.html.

[32] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling," in *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID), Evry, France*, 2016.

[33] C. Lever, P. Kotzias, D. Balzarotti, J. Caballero, and M. Antonakakis, "A lustrum of malware network communication: Evolution and insights," in *Proceedings of the 38th IEEE Symposium onSecurity and Privacy (S&P), San Jose, CA*, 2017.

[34] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Computing Surveys (CSUR)*, 2017.

[35] M. Belaoued and S. Mazouzi, "An MCA Based Method for API Association Extraction for PE Malware Categorization," *International Journal of Information and Electronics Engineering*, 2015.

Appendix

7ev3n	9002 RAT	AbbathBanker	Acronym	AdamLocker	Adylkuzz
AgentTesla	Alice ATM	Alina POS	AlmaLocker	Alphabet Ransomware	AlphaLocker
Alphanc	Alreay	AMTsol	Andromeda	Apocalypse Ransomware	Ardamax
Arefty	Arik Keylogger	Asprox	Athenago	ATMitch	AugustStealer
Aveo	Ayegent	AzorUlt	Babar	BadEncript	BadNews
Banatrix	Bart	Batel	Bedep	BetaBot	BlackEnergy
BlackRevolution	BlackShades	Bolek	Bravonc	Bredolab	BTCWare
Buhtrap	Bundestrojaner	Bunitu	Buzus	c0d0so0	Cabart
CadelSpy	Carbanak	Carberp	Cardinal RAT	Casper	CCleaner Backdoor
Cerber	ChChes	Chinad	Chir	Chthonic	Citadel
Client Maximus	CloudDuke	CMSbrute	CobaltStrike	Cobian RAT	Cockblocker
CodeKey	CoinMiner	ComodoSec	Conficker	Contopee	CoreBot
CoreShell	CradleCore	Crashoverride	CredRaptor	Crylocker	CrypMic
Crypt0l0cker	CryptoFortress	Crypto Ransomeware	Cryptolocker	CryptoLuck	CryptoMix
Cryptorium	CryptoShield	Cryptowall	CryptoWire	Cryptxxxx	CsExt
Cuegoe	Cutwail	CyberSplitter	CyberGate	CycBot	DarkComet
DarkPulsar	DarkShell	DarkTrack RAT	Daserf	DEloader	Deltas
DeputyDog	DeriaLock	Derusbi	Devils RAT	DiamondFox	Dimnie
DirCrypt	DMA Locker	Dorkbot	Dorshel	DoublePulsar	DownDelph
Downeks	DownRage	Dreambot	Dridex	Dropshot	DualToy
DuQu	Duuzer	Dyre	EDA2 Ransomware	EhDevel	Elise
Enfal	Erebus	Etumbot	EvilBunny	EvilGrab	EvilLoader
Xtreme RAT	FakeRean	FakeTC	Fanny	Fast POS	Feodo
FileIce Ransomware	FinFisher	Fireball	FireCrypt	FlokiBot	Flusihoc
Fobber	Formbook	Furtim	GameoverDGA	GameoverP2P	Geodo
Ghole	GhostAdmin	Ghost RAT	Glasses	Globe Ransomware	GlobeImposter
GodzillaLoader	GooPic	Gozi	GPCode	GrabBot	Graftor
Gratem	H1N1 Loader	Hamweq	Hancitor	HappyLocker	Harnig
Havex RAT	Hhawkeye Keylogger	Helminth	Heloag	Herbst	Herpes
Hesperbot	HiZor RAT	Hiddentear	HighTide	HiKit	HLUX
HtBot	httpbrowser	Hworm	IAP	Ice IX	Idkey
ImminentMonitor RAT	Infy	ISFB	ISMagent	ISMdoor	iSpy Keylogger
isr_stealer	isspace	jaff	jager_decryptor	jaku	jasus
Jigsaw	Jimmy	Joao	JqjSnicker	JripBot	KAgent
Karagany	KasperAgent	Kazuar	Kegotip	Kelihos	Keylogger (APT3)
KhRAT	KillDisk	KINS	KokoKrypt	Konni	KoobFace
Kovter	KrBanker	KrDownloader	Kronos	Kuaibu8	Lambert
LatentBot	Lazarus	Laziok	Limitail	Listrix	LockPOS
Locky	LockyDecryptor	LokiBot	LuminosityRAT	Lurk	Luzo
MadMax	Magala	Maktub	ManameCrypt	Manifestus Ransomware	MatrixBanker
MatrixRansom	Matsnu	Mewsei	Miancha	Micropsia	Mimikatz
Mirai	Miuref	MM Core	MobiRAT	Mocton	Moker
Mokes	MoleRAT Loader	Moonwind	Morphine	Moure	MultigrainPOS
Murofet	Mutabaha	Nabucur	Nagini	Naikon	Nanocore
Necurs	NetRepser Keylogger	NetSupportManager RAT	NetTraveler	Netwire	Neutrino
NeutrinoPOS	Newcore RAT	NexsterBot	NexusLogger	Nitol	njRAT
Nymaim	Oddjob	Odinaff	Opachki	OpGhoul	Orcus RAT
OvidiyStealer	PadCrypt	PandaBanker	Petrwrap	Petya	Pittytiger RAT
Ploutus ATM	PlugX	PoisonIvy	PolyglotRansom	Pony	PopcornTime
Poweliks Dropper	PowerDuke	PowerSniff	Prikormka	Pteranodon	Pushdo
Pykspa	Qadars	QakBot	QuantLoader	Quasar RAT	Radamant
Ramdo	Ramnit	Ranbyus	Ranscam	Ransoc	RapidStealer
RawPOS	Razy	RCS	RedAlert	RedLeaves	Remcos
Remexi	RemsecStrider	Retefe	Revenge RAT	Rincux	RockLoader
Rofin	Rokku	RokRAT	Rombertik	Romeos	Roseam
rtm	Rurktar	Sage	Sakula RAT	Sality	Samsam
Satana	SathurBot	Screenlocker	Sedreco	SedUploader	SendSafe
Serpico	ShadowPad	Shakti	ShapeShift	Shifu	ShimRAT
Shujin	Shylock	Sierras	Siggen6	Simda	Sinowal
Skyplex	Slave	SmokeLoader	Snifula	SNS Locker	Socks5Systemz
SolarBot	Spora	SpyBot	sslmm	Stabuniq	StegoLoader
Strongpity	SuppoBox	Swift	SyncCrypt	SynFlooder	SynthLoader
Sys10	SysGet	SysScan	Teerac	TeleBot	Tempedreve
Terminator RAT	TeslaCrypt	Thanatos	Threebyte	ThumbThief	Tidepool
Tinba	TinyLoader	TinyNuke	TinyTyphon	TinyZBot	Tofsee
TorrentLocker	TrickBot	Trochilus RAT	Troldesh	Trump Ransomware	Tsifiri
Turnedup	UACme	Uiwix	unknown_001	unknown_002	unknown_003
unknown_005	unknown_006	unknown_008	unknown_013	unknown_020	unknown_023
unknown_026	unknown_029	unknown_030	unknown_031	unknown_032	unknown_033
unknown_034	Unlock92	Upatre	Urausy	UrlZone	Vawtrak
VenusLocker	Virut	VMZeus	Vreikstadi	Wannacry	Waterspout
WinMM	WINSloader	WndTest	Woolger	XAgent	XbotPOS
XBTL	Xpan	XsPlus	Xswkit	XTunnel	Yahoyah
ZeroAccess	ZeroT	Zeus	Zeus Mailsniffer	ZeusSphinx	ZeusSSL
ZhmMmikatz	ZLoader				

Table 6: Windows malware families (446) covered in the evaluation presented in Section 5.