Classification, Characterization, and Contextualization of Windows Malware using Static Behavior and Similarity Analysis

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Daniel Johannes Plohmann

aus

Bonn-Bad Godesberg

Bonn, 2022

Dissertation

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn.

Erstgutachter:	Prof. Dr. Peter Martini Rheinische Friedrich-Wilhelms-Universität Bonn
Zweitgutachter:	Prof. Dr. Christian Rossow CISPA Helmholtz-Zentrum für Informationssicherheit, Saarbrücken
Tag der Promotion:	28.06.2022
Erscheinungsjahr:	2022

It is the pervading law of all things organic and inorganic,
Of all things physical and metaphysical,
Of all things human and all things super-human,
Of all true manifestations of the head,
Of the heart, of the soul,
That the life is recognizable in its expression,
That form ever follows function. This is the law.

Louis H. Sullivan, The Tall Office Building Artistically Considered, 1896.

Contents

1. Introduction

Over the last 15 years the characteristics of malicious software (short: malware) have changed dramatically. In the early days, most variants were created with the intention to demonstrate skill and gather attention within a secluded scene of programmers. Source code was published and traded on Virus eXchanges (VX) and the ambition was rarely to cause serious damage but to explore and push boundaries.

A major shift in behavior can be observed around the years 2006 and 2007 when a rapidly rising number of malware specimen surfaced that concentrated on criminal activities. They were clearly geared towards amassing financial gain by systematizing fraud, enabled by organizing compromised machines in the form of so-called botnets. Some botnets were dedicated to drastically scale up spam volume [1], which in turn was partially used to deliver malware [2], including malware families renowned for online banking fraud such as `win.zeus` and `win.gozi` [3]. With growing resources and experience on the attacker side, malware has been constantly refined ever since.

In recent years, a further progression towards extortion attacks using file-encrypting malware (ransomware) can be observed. An outstanding case occurred `win.wannacry` in May 2017 [4], which was able to affect more than 200,000 computer systems in just 8 hours due to its self-spreading mechanism. The attack also had notable impact on critical infrastructure including transportation, telecommunication, and health-care providers around the world. As one example, it widely disrupted operationality of the UK National Health Service (NHS) and caused financial damage estimated at £ 92 million [5].

Similarly, malicious software plays a key role in state-driven digital espionage and sabotage [6]. Documented cases of targeted attacks and high-profile network intrusions go back to at least the late 1990s [7]. Backed with immense resources of nation states, attack campaigns have reached a significant amount of intricacy and sophistication, as for example the case of `win.stuxnet` demonstrated, which targeted industrial control systems and destroyed an estimated 20% of the Iranian nuclear centrifuges [8].

Along with the increase in scale and complexity of malware families and their associated attacks, a corresponding need for better understanding of the threats has grown. In particular, a strong demand for detailed analysis of and information about malware arose, which is a prerequisite to perform accurate threat assessment and to derive strategies for protective countermeasures and mitigations.

As a result, malware analysis has remarkably matured as a research field and profession. Nevertheless, substantial challenges remain that hinder even better and more efficient analysis. Two major challenges that we identify and tackle in this dissertation are availability of *ground truth* and *situational awareness*.

Without question, the availability and quality of ground truth has huge impact on experiments and thereby on the evaluation of methods and tools. As in many other fields, openly available and representative data sets for malware research are hard to come by. This also applies for Windows malware and especially for data suited for static analysis, as protection mechanisms such as polymorphic packing are estimatedly found in around 95% of malware samples encountered in the wild [9, 10], making their payloads inaccessible to direct application of such techniques. For a long time, this has particularly affected the academic research community, where several studies had to make use of unsatisfactory data sets, for example containing only few, potentially long outdated malware families or otherwise large corpora of unlabeled samples.

Situational awareness is crucial for the decision-making that steers efficient and effective malware analysis. Among the frequently recurring tasks in this regard, classification of malware is very important, as it enables to distinguish known from unknown malware. If an analyst is already familiar with a malware family, they may decide to immediately focus on potential changes in the code compared to prior versions. Otherwise, having pinpointed the family additionally allows an analyst to contextualize their investigation early on in case previously published results of other researchers exists. In general and specifically for unknown malware, a core objective is usually a characterization of the threat. This includes the assessment of potential behavior and capabilities, which typically require interactions with the operating system interfaces. Another common facet of analysis is the identification of components used for the creation of malware, such as third-party libraries or code fragments that already surfaced in other malware families.

1.1. Research Questions

In order to approach the challenges outlined before, we define the following research questions.

Research Question 1 (RQ_1): How should a malware corpus be composed in order to enable academic researchers to conduct representative malware research while simultanously serving as a relevant resource for practical malware analysts?

Undoubtedly, ground truth of high quality is a necessary foundation in order to be able to do meaningful research and to achieve expressive results. Apart from exploring and formulating general requirements for ground truth, a goal of this research question is the harmonization of potential nuances in aspects found in academic and practical malware research. As a part of the answer we compose a data set satisfying the identified requirements, with a focus on enabling static analysis of Windows malware and providing memory dumps as a format for an unpacked representation. Given the availability of comprehensive ground truth, we are now enabled to pursue various follow-up questions.

Research Question 2 (RQ_2): To which degree is the integrity of original payload meta data and file structure maintained, and based on this data, what can be inferred about tool chains and methodologies as used by the malware authors?

An interesting question that has not been addressed extensively in the literature so far is concerned with the consistency and authenticity of payload (meta) data encountered

during malware research. Malware authors and users have a valid interest in hiding their tracks and aggravating analysis, which may lead to tampering with binaries prior to deployment. The degree of reliability in turn has implications for the applicability of both analysis methods and interpretability of information extracted from malware, for instance when attempting to date the origin of certain malware versions based on the compilation timestamp included in a binary.

Research Question 3 (RQ_3): Using static analysis, how can Windows API usage information be robustly extracted from memory dumps?

A program's interactions with the Windows API provides extensive insights into its potential behavior and capabilities, which emphasizes the relevance of robust methods of identifying these interactions. Despite this fact, only few methods have been published and best to our knowledge they have not been formally evaluated, which we also address with this question.

Research Question 4 (RQ_4): How frequently do malware authors apply obfuscation schemes to their WinAPI usage?

In a spirit similar to RQ_2, we investigate the popularity of anti-analysis methods used to conceal Windows API usage. For this, we define a three-tiered taxonomy of obfuscation degree, divided into no obfuscation, dynamic imports, and heavy obfuscation. As a result, we notice that no obfuscation and dynamic imports during the runtime are common while only a minority of malware families uses heavy obfuscation. This allows us to study WinAPI interactions in malware more in-depth with the next question.

Research Question 5 (RQ_5): How characteristic are Windows API usage profiles for malware families and can they be used in the context of malware identification?

Given the large variety of malware families and behaviors exposed by them, it has been shown before that for example WinAPI call traces recorded in dynamic analysis environments can be successfully used to classify malware [11]. In the given research question, we investigate whether this is also possible for statically extracted WinAPI usage profiles, and measure how unique the usage of individual WinAPI functions across hundreds of malware families actually is.

Research Question 6 (RQ_6): How can code and Control Flow Graph information for Intel x86/x64 code be robustly recovered from memory dumps, without making further assumptions about the structural properties of the given file?

Accurate disassembly is a prerequisite that many other advanced binary code analysis techniques build upon. During past malware investigations, we noticed a decline in disassembly quality offered by popular tools when they are confronted with memory dumps as input format opposite to proper unmapped executable files with header meta data. Adressing this question, we quantify the impact and recombine disassembly strategies found in the literature to propose a method that achieves high disassembly quality without suffering similar effects when processing memory dumps.

Research Question 7 (RQ_7): How frequent is third-party library usage as shared code in Windows malware?

As labels for functions are typically not initially available during investigations, analysts are prone to choosing and studying functions of interest that later turn out to be statically-linked library code. To better understand how common library code is encountered in malware, we use a fuzzy code similarity method to estimate the frequency with which it is encountered. Being able to reliably identifying library code may also save valuable analysis capacity.

Research Question 8 (RQ_8): Apart from libraries, what is the actual overlap of intrinsic code in Windows malware families?

Related to RQ_7, insights into code overlap between malware families that is not caused by library code are of high interest. If no extensive code sharing can be assessed, this would imply that source code is mostly kept private, which in turn would suggest that actual overlap will more likely indicate common authorship or other kinds of relationship.

1.2. Contributions

In this section, we summarize our primary contributions. A part of the results has been already published in the following peer-reviewed papers:

- Malpedia: A Collaborative Effort to Inventorize the Malware Landscape [12]
- ApiScout: Robust Windows API Usage Recovery for Malware Characterization and Similarity Analysis [13]

Chapters 4 and 5 thematically incorporate these publications, but discuss the respective topics in much extended depth. Additionally, we use a consistent data set snapshot throughout this dissertation, which is by itself also significantly extended and updated compared to the previous publications.

In accordance with the research questions, our contributions can be grouped into three major topics:

1. Malware ground truth
    - We specify a set of requirements for malware ground truth that is well-suited to enable representative research in both academic context and practical malware analysis.
    - Following these guidelines, we create Malpedia as a reference data set tailored towards static analysis with a focus on Windows malware. The reference snapshot used in this thesis contains 1,136 manually verified malware families with 3,469 representative samples, with a majority of them unpacked and thus ready for in-depth analysis.
    - We conduct an extensive comparative structural analysis across 839 Windows malware families. Showing that most extractable meta data seems plausible, we use it to derive insights into malware author preferences and methodologies.

2. Malware behavior and interaction with the Windows API (WinAPI)
    - We propose ApiScout as an approach to extract WinAPI usage information from memory dumps and demonstrate its reliability.
    - Applying ApiScout to Malpedia, we show that dynamic WinAPI imports are found in 49.5% of malware families studied while hard obfuscation is rarely encountered (3.9%). We furthermore provide a detailed usage occurrence analysis of individual WinAPI functions for which we also provide a semantic categorization scheme.
    - We present ApiVectors as a concept to capture WinAPI usage profiles of malware families. We expose that these profiles are characteristic per family and can be successfully used for malware classification.
3. Malware code and similarity analysis
    - We introduce SMDA, a combination of disassembly best-practices able to robustly retrieve control-flow graph information from memory dumps without relying on structural file meta data.
    - We propose MCRIT, a system using a MinHash-based fuzzy code similarity method, and use it to estimate that on average at least 15-20% of the code found in malware is potentially introduced through the inclusion of third-party libraries.
    - Using MCRIT to exclude suspected library code from malware during similarity analysis, we demonstrate that similarity scores above 10% across families are rare apart from were plausible reasons for relationship exist, such as public source code leaks or documented links on other levels.

1.3. Thesis Outline

The remainder of this thesis is structured as follows.

Chapter 2 provides background information on fundamental notions used in this thesis. This includes a general overview of relevant concepts used in binary code analysis, and an introduction to malware and the analysis of malware.

Chapter 3 features an overview of related work. It surveys the state of available ground truth for malware research and outlines previous works on Windows API usage analysis. Furthermore, seminal publications on code analysis with a focus on disassembly and similarity analysis are listed and works on malware analysis workflows and methodology are summarized.

Chapter 4 starts out with the definition of a set of requirements for the composition of high quality malware ground truth, especially suited for static analysis. This is followed with the detailed description of Malpedia, our reference implementation of such a corpus. The corpus is then used to conduct a comparative structural analysis of Windows malware, comparing file integrity and meta data from unpacked malware samples of 839 Windows malware families.

Chapter 5 examines how malware interacts with the Windows API. ApiScout is introduced as a method to reliably extract API usage information from memory dumps and then used to analyze the samples in Malpedia. Based on a usage frequency analysis, ApiVectors are defined as a method to store and compare usage profiles, which is then used for malware classification.

Chapter 6 deals with disassembly and code similarity analysis. First, SMDA is explained as a method for effective recovery of code and control flow information from memory dumps. Next, we introduce MCRIT, a Minhash-based system for efficient fuzzy one-to-many code comparision. MCRIT is then used to investigate third-party library usage and code sharing in the malware families covered by Malpedia.

Chapter 7 provides a summary of the thesis. Apart from discussing the primary contributions and conclusions, we give an overview of practical impact that our results have achieved and which avenues for future work we envision.

2. Background

In this chapter we provide background information that we consider essential for understanding the matters discussed in the core chapters of this thesis. As this thesis is primarily concerned with malicious software for the Windows operating system and its analysis, we focus on these topics.

We start with an explanation of binary code analysis which we follow up with an introduction to its underlying basic terms and concepts, including the procedure of compilation, typical program structure, programmatic API usage, and disassembly as a key technique to enable in-depth analysis. Next, we characterize the phenomenom of malicious software and survey the foundations of methodology used to analyze it.

2.1. Binary Code Analysis

A fundamental research direction of computer science is program analysis [14], with its primary use cases being code optimization and verification. A wide array of approved techniques that has evolved over years in this discipline is directly applied on a program's source code [15]. However, source code is not always available for analysis, and furthermore, a misalignment between source code representation and its intended effects on the one hand and what is actually executed by a processor on the other hand has been documented [16].

In such situations, the analysis has to be performed directly on compiled binary code instead. It has to be noted that the unavailability of source code makes analysis significantly more challenging. This is primarily due to the reason that the translation from source code to machine code typically involves the removal of e.g. structural and type information, which are not needed for execution and thus are only implicitly encoded into the program during the compilation procedure. Binary Code Analysis (BCA) is considered a core technique for reverse engineering, which is concerned with the recovery of human understandable program semantics from its given binary representation. In this context, primary purposes of analysis are enabling compatibility with legacy software for which the source code is lost, auditing the security of proprietary software, as well as the analysis of malicious software, as described in detail in Section 2.3.

Generally, BCA encompasses both techniques of static and dynamic analysis, the former being performed without the execution of the code under analysis and the latter on its execution. Both methodologies have their respective tradeoffs.

Advantages of dynamic analysis are directly connected to its ability to follow a concrete execution path, meaning that all data manipulated at an arbitrary point during execution is immediately inspectable and there is no disambiguity about the flow of execution.

However, its potential limitations also directly arise from these properties. On the one hand, being tied to a concrete execution path may significantly limit visibility in terms of code coverage with regard to the whole program. On the other hand, concrete execution implies actual runtime with all potential dependencies within the operating system, which may introduce significant analysis overhead.

Static analysis has its advantages exactly in these weak spots, striving for completeness in analysis and code coverage while avoiding runtime overhead pitfalls. However, it has to invest more effort with inference and approximation where the concreteness of dynamic analysis provides immediate accuracy. For this thesis, we direct our focus almost exclusively on applications of static analysis as the promise of completeness is beneficial to our objectives.

We now continue with the introduction of several basic concepts relevant in the context of BCA.

2.1.1. Compilation

Computer programs typically consist of a series of structured statements that advise the execution of certain actions that manipulate the program flow or state. They are usually written in a programming language and the collection of statements is referred to as the program's source code, which may be organized in several source code files.

In order to run computer programs, they first need to be prepared and transformed for execution. Two popular concepts for this are interpretation and compilation. Interpretation on the one hand is a concept in which an intepreter, which is typically a so-called runtime system, directly reads the source code and executes its statements as they are encountered while following along the program flow. Compilation on the other hand is the term that describes the transformation of a program written in a source code language into a machine code suited for execution by a certain target processor architecture and operating system. [17]

The overall procedure of compilation from source code to an executable program is described by Aho et al. [17] as a structured sequence of four phases that consecutively process each other's output: preprocessor, compiler, assembler, linker.

Preprocessor: In the first phase, the preprocessor searches for all source code files which are part of the program. In case the programming language supports shorthand symbols (macros), they are also expanded.

Compiler: The compiler itself works in multiple phases which can be summarized as analysis and synthesis, which are also called front-end and back-end. Analysis splits the source code into fragments, which are then captured using a grammatical structure that allows to organize the program in an intermediate representation (IR). Part of this phase is verification, which can be used to alert on potential syntactic or semantic errors. Synthesis is the construction of a concrete program for the target architecture from the IR. It is a sequence of instructions in assembly language. Optimization of the code may occur during both analysis and synthesis.

Assembler: The assembler produces relocatable and executable binary code from the textual assembly representation that was generated by the compiler in a straightforward manner.

Linker: Larger programs may constitute of several components that are compiled individually into object and library files. The linker has the task to combine these relocatable parts into one coherent program through static or dynamic linking. In static linking, a single output file is produced through direct inclusion of all dependencies. When using dynamic linking, the resulting program is prepared in a way that it can specify its dependencies during load time and then work with memory references to the external code.

Popular frameworks for compilation are the GNU Compiler Collection (GCC) [18] and the LLVM compiler infrastucture [19] as well as Microsoft Visual Studio as an IDE for its assorted products Visual C++, Visual C#, Visual Basic, etc. [20].

The result of the full compilation procedure is usually not just plain code but self-contained executable program files, which are the subject of the next Section.

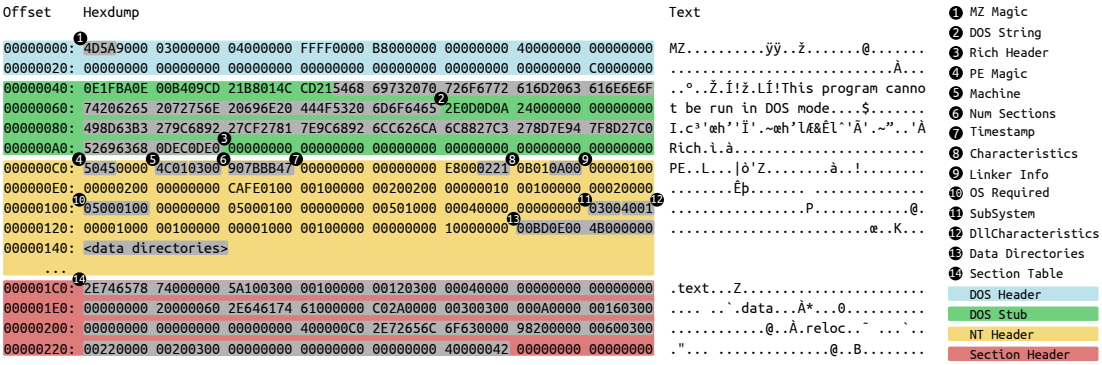


Figure 2.1.: Structural Overview of a Windows PE header [12]. The highlighted fields are discussed later on in Section 4.4.

2.1.2. Structure of Windows Executable Programs

Modern operating systems need additional information on how to run an executable binary program. This is typically achieved by embedding meta data along the code, usually in the form of program headers and structuring it into sections, which in combination serve as a sort of blueprint for the creation of a suitable executable environment for the program. A popular example and root for many formats is the Common Object File Format (COFF) format, which was originally used by both Unix-like and Windows operating systems. Linux nowadays has moved to the Executable and Linkable Format (ELF), while Windows still uses the Portable Executable (PE) file format, which is based on COFF.

Following the scope of this thesis on Windows malware, we will now focus on the PE format [21]. Figure 2.1 shows an example of a typical header of a PE file.

It starts off with the DOS Header, which in itself starts with the two bytes `MZ`, an abbreviation of the name of Mark Zbikowski, who designed the MS-DOS format [22]. Another relevant field is situated at offset `0x3C`, which indicates the start offset of the PE signature.

The DOS Stub follows the DOS Header. It is in itself a minimal 16bit program that signals incompatibility with 16bit systems by simply emitting a string (typically "This program cannot be run in DOS mode") and exiting. Depending on how the program was created, the DOS Stub may be potentially followed by the Rich Header, a data structure inserted only by the Microsoft Linker [23]. The Rich Header is not officially documented by Microsoft but its structure has been extensively analyzed by researchers [23, 24, 25, 26]. The data structure contains information about the program's source code, in particular the number of input files processed by components of the compiler tool chain.

The next part is the NT Header, which is divided into a COFF File Header and Optional Header. As examples for relevant fields, the COFF File Header specifies the required CPU architecture and number of sections. It also contains a field with a compilation timestamp. The Optional Header carries detailed information about how to set up the program for execution in memory, including the address of the program entry point. Another part of the Optional Header are a sequence of 16 data directory entries, which each contain a pointer and size for these respective additional data structures. Important examples for data directories are the Import Table and Import Address Table, the Resource Table, and the Base Relocation Table.

Finally, usually one or more Section Headers follow, that each describe the name, location, and size of a section in the PE file and how it is supposed to be mapped into virtual memory. It should be noted that the PE format generally allows great leniency and it is for example possible to produce a PE file without any sections [27].

Because program interaction with the Windows Application Programming Interface (Windows API, short WinAPI) is extensively discussed in this thesis, we now provide a short overview of the related concepts. The central idea of the Windows API is to enable and manage interference of third-party programs with the operating system. It provides a set of well-defined interfaces and data structures to ensure that all interactions happen in an orderly fashion. The Windows API is functionally organized in several components that for example handle access to file systems, execution contexts such as services, processes, and threads, the Windows registry, peripheral devices and the graphical user interface, or networking [28].

From a technical point of view, a program is usually dynamically-linked (cf. Section 2.1.1) against all required endpoints of the WinAPI and the respective information is encoded in the program's Import Table. This means that this information is generally obtainable through means of static program analysis. It is however also possible to obtain references to WinAPI functions during runtime, e.g. by making use of the WinAPI functions `kernel32.dll!LoadLibraryA` and `kernel32.dll!GetProcAddress` to command the loading of a system library into memory and then resolving the address of a desired WinAPI endpoint for use.

2.1.3. Disassembly

In order to conduct Binary Code Analysis, fundamental preprocessing in the form of disassembly has to be conducted. The goal of disassembly is to produce a program representation in assembly language that is equivalent in syntax to the source program [29]. Thus, disassembly aims at the inversion of the last phases of the compilation procedure (cf. Section 2.1.1).

Two basic strategies exist for the disassembly procedure: Linear sweep and recursive traversal. Linear sweep processes the program entirely sequentially, which makes it very fast but also susceptible to situations where the instruction stream is interrupted, e.g. when code and data are intertwined. Nonetheless, it has been shown that this is a very effective method for instruction recovery, especially for Linux binaries [30, 31].

Recursive traversal on the other hand interprets the disassembled code during the processing and considers code branches to identify promising locations for continuation of the analysis. Downsides of this approach are that not all code may be discovered in case direct code references are missed.

When using disassembly, a number of conceptual primitives are typically discussed:

Instruction: An instruction is a single machine-level command that advises the CPU to perform an action. Its binary representation can be converted to a semantically equivalent but human understandable form in assembly language. This so-called mnemonic representation is a core element of disassembly and consists of an opcode (e.g. `mov`, `add`) and potentially zero to four operands [32].

Basic Block: A sequence of instructions that are executed consecutively without interruption, i.e. have only one entry point and exit are called a Basic Block (BB). BBs are primarily used as a concept to structure control flow.

Function: The term function is conceptually similar to functions in the source code. In disassembly, they can be understood as the set of Basic Blocks that constitute the implementation of said function. Typically, a function has only a single function entry point (FEP) and multi-entry functions or shared basic blocks across functions are very rare [30].

Control Flow Graph: The term Control Flow Graph (CFG) normally refers to a single function and is formally a directed graph $G = (V, E)$ in which the vertices V are the set of Basic Blocks and the edges $E \subseteq V \times V$ are defined between these Basic Blocks. It is also referred to as an intraprocedural Control Flow Graph. The interprocedural Control Flow Graph (ICFG) on the other hand are all CFGs of functions and additionally any references between functions that are for example created through calls and jumps.

2.1.4. Code Similarity

Code Similarity Analysis is concerned with the comparison of two or more chunks of code in order to map out their potential similarities and differences. It is among the most interesting use cases of binary code analysis and its primary real world applications

include vulnerability and patch analysis, but also the detection, analysis, clustering, and attribution of malicious software. As a result, it has received significant attention in the academic community over the last years. [33]

Key challenges opposite to the comparison of source code lie particularly in the abstractions (e.g. removal of symbolic information) and transformations applied during compilation. A given compiler toolchain and its parameterization like optimization levels may introduce variance that can make it hard to even match code originating from identical source code. For this reason, a wide range of approaches has emerged and methods applied may vary greatly depending on the concrete instance of problem considered. As an example, a stronger emphasis may be set on measuring syntactic equality, which may imply higher likelihood of same origin, versus classifying semantic equality, when interested in identification of similar logic and algorithms.

2.2. Malware

Malicious Software, or short malware, is the umbrella term for any kind of software that exposes unwanted or harmful behavior on a target computer system. It unifies the increasingly recessing terms that were used in the past to label pieces of malware into groups by their functionality, e.g. viruses being self-replicating by infecting other executables, worms using network-based active spreading mechanisms, or trojan horses concealing their malicious capabilities while posing as legitimate software [34].

Originally, malware was written primarily for personal enjoyment and the gain of reputation within a small community by showing technical skill in its creation. This changed drastically with the wide-spread adoption of the Internet. Allowing interaction with infected systems at ease, malware started to be used to directly access and manipulate many systems simultanously, leading to so-called botnets (short for “robot networks”, inspired by the Czech term “robota”). [34]

With society’s growing dependence on computers for communication and electronic commerce, financial exploitation of compromised systems became lucrative and a driving factor for the creation of malware. Around 2006, this paradigm shift expressed itself in rising economization of the malware scene and a strong increase of professionalism. This can be observed through multiple factors.

First, in order to circumvent detection capabilities of Antivirus (AV) software, polymorphic packers, i.e. programs that protect malware as an inner payload through various encoding and encryption mechanisms were widely adopted. This increased the absolute numbers of individually malicious files (also called samples) detected by year from 7.3 million in 2008 to 38.3 million in 2014 and now 89.2 million in 2020 as reported by AV-Test [35]. However, since malware is still authored manually, the number of different families and variants will remain magnitudes smaller than the number of cryptographically hash-unique samples observed.

Second, the skill required to create and manage botnets was reduced with the introduction of “botnet construction kits”, which allowed less technically verse actors to operate them and also cause notable damage. This further evolved into the adoption of

the operational "as-a-Service" model, lately most notorious in the form of Ransomware as a Service (RaaS) [36]. An author of ransomware may sell or license their malware to one or more affiliate threat actors, who then carry out the actual operational attacks and compensate the author in the form of a fee or share in profits. Finally, economic pressure motivates malware writers to produce the least detected, most persistent, and functionally capable product, which leads to increasingly mature and complex malware. The same applies to aforementioned polymorphic packers, which are competetively developed and marketed.

Throughout this thesis, we will use the following definitions when referring to these respective terms:

Malware Sample: This term simply refers to a single file containing malicious software.

Malware Family: A malware family constitutes of the entirety of malware samples that originate from the same source code and belong to the same project from a developer's point of view. Incorporating this developer's perspective enables us to differ between cases where multiple malware projects are derived from a common code base, e.g. when it was published or leaked (e.g. `win.zeus`, `win.gozi`, `elf.mirai`) and also when a project was rewritten (e.g. `win.gpcode`). Apart from the malware core, this may include further components that serve as functional extensions specific to the family, such as dedicated loaders or plugins that enable various capabilities. While not being a perfectly sharp definition, this serves as an established compromise and consensus among many practitioners. [12]

Malware Version: A version (or variant) within a family resembles a development snapshot of the malware that was captured in the wild. The distinctiveness of versions may largely vary, ranging from minimal incremental differences to major changes to the source code. In some cases, malware families use their own internal versioning scheme, in other cases compilation timestamps may provide a guideline as they frequently appear to be genuine (cf. Section 4.4.2).

(Un)packed Malware: The term unpacked malware implies that a malware sample is an immediate representation of the malware family itself. There is no presence of any third-party code that is not related to its own code base that may have been applied after compilation in order to conceal its identity and potentially evade detection. The code in the unpacked malware may still be divided into multiple components or stages or even be obfuscated if it is a feature encoded in the family's source code itself. Opposite to this, a packed sample is (unpacked) malware that has been further processed, which often involves compression and encryption as well as addition of a code stub that reverts the packed code to its original form and prepares it for execution. [12]

2.3. Analysis of Malware

The economic impact caused by a successful malware attack can be devastating, e.g. it is estimated that a company on average suffers financial damage between multiple hundred thousand to millions of Dollars [37], which implies that it is highly relevant to defend

and fight against malware. In order to have a prolonged effect, countermeasures have to go beyond mere protection of computer systems, e.g. by detecting potential malicious intent of files before their execution.

A foundation for performing active mitigation is accurate information about malware, obtained through detailed analysis. Core objectives of malware analysis are usually to figure out its means of persistence, communication, and functional capabilities [38]. Identifying persistence mechanisms allows to identify and remediate compromised machines. Being able to decode malicious Command and Control (C&C) traffic enables tracking of actors as well as mapping out the network infrastructures they use for their purposes. Analysis results on capabilities found in a given malware sample can serve as a basis for impact estimation when assessing potential damages in case of an incident. Extraction and description of characteristics of certain malware families and attack methodologies may in some cases generate leads for the attribution of attackers.

Effective analysis of malware heavily leverages concepts of reverse engineering. In general, reverse engineering is the process of obtaining knowledge or design information up to a degree that is sufficient to recreate the original product which has been subject to analysis [39]. Reverse engineering when applied to the analysis of computer programs typically requires Binary Code Analysis (cf. Section 2.1) as especially with malware, source code is almost never available. Luckily, it is rarely necessary to fully examine a malware sample as full recreation is barely the default goal. Only selected code aspects may be of such high interest that the effort of in-depth study or re-implementation is justified, e.g. decryption of network traffic or extraction of configuration files for automation purposes. However, malware analysis frequently faces techniques used to additionally protect the software, further aggravating analysis. As a notable example, when considering full automation of unpacking malware, this task in itself is exposed to the fundamental limitation of undecidability imposed by the halting problem [40].

As outlined in Section 2.1, Binary Code Analysis can be seen as divided into the complementory approaches of dynamic and static analysis. We will now introduce the key techniques within these in the context of malware analysis. While performing practical analysis, an analyst will usually switch between both methodologies to benefit from their individual advantages.

2.3.1. Dynamic Analysis

A primary objective of dynamic malware analysis is to closely monitor execution in order to make observations and conclusions about runtime behavior. This is usually already very informative, as programs have to extensively interact with an operating system's interface to achieve meaningful impact (cf. Section 2.1.2).

Because harmful behavior is expected from malware, the execution environment is typically set up carefully to contain potential damage. This involves aspects such as not having valuable data unintentionally available within the system and limiting network access, first and foremost to avoid accidental participation in attacks (e.g. distributed denial of service) as a side effect of the analysis. A helpful tool frequently used in order to achieve containment and consistency is virtualization [41]. The abstraction into a

virtual guest system provides a natural barrier from its host. Furthermore, being able to snapshot a full system state as supported by most virtualization solutions allows to quickly and reliably revert to known clean states. A downside of virtualization is that some packers or malware families will probe the system trying to detect it, in which case they may alter or stop their execution.

With regard to analysis methodology, blackbox analysis describes the approach of solely monitoring the effects of execution without analysis of the code causing them [42]. While appearing basic in its application, it is a powerful way that often already yields actionable Indicators of Compromise (IOCs). This includes details such as under which file paths a malware may try to achieve persistence or to which Command and Control servers it may try to communicate, allowing to scan for or block them.

A natural development was the automation of blackboxing, which lead to the concept of sandboxing [43]. A sandbox is a tailored system that provides orchestration of the common sequence of analysis steps and scope. It will first ensure execution is started from a clean state. Based on its configuration, it will record any changes to the file system and configuration, log network conversations, and typically also track suspicious interactions with the system API that are known to be used to enable malicious capabilities. After an analysis runtime of usually a few minutes has concluded, output is generated. This is typically a report optimized for human interpretation or for further machine-based processing. Sandboxing in professional environments is often horizontally scaled to many thousands of malware executions daily and coupled with anomaly analysis to identify new variants.

It has to be noted that the previously mentioned drawback of dynamic analysis usually being tied to limited, concrete execution paths (cf. Section 2.1) also affects blackboxing and sandboxing in the way that path coverage is potentially constrained and may not cover regions in the malware where critical capabilities are located.

Debugging is another highly relevant use case of dynamic malware analysis. Originally, debugging describes the procedure of runtime error analysis in the context of software development. As an analysis tool, debuggers are capable of providing full information about CPU register and memory contents and enable following program execution instruction-wise or between so-called breakpoints that can be set on memory addresses. This allows for very fine-grained analysis that is often applied to obtain a better understanding of narrow and select code regions such as particularly complex functions. It is even possible to debug code locations when they have not been reached through a natural execution flow.

While this dissertation focuses on static analysis, one technique used originates from digital forensics: the capture of volatile memory on a running system [44]. Also referred to as memory dumping, it can be used to obtain a variation of the unpacked state of a malware sample without consideration of how this state was reached, i.e. the packer's unpacking procedure. This is in almost all cases equally suitable for in-depth analysis as an unmapped representation and provides even better potential as a normalized representation as demonstrated in Section 4.3.1.

2.3.2. Static Analysis

Static analysis in the context of malware analysis can be divided into (initial) cursory and detailed analysis [38].

Cursory static analysis is typically performed initially when a new investigation into a malware sample is started and intended to enrich the analysis context and help to create an overview of its file characteristics, e.g. whether it is packed or not. One primary objective here is often to perform classification and identify the malware family for a given malware sample as this allows to potentially accelerate analysis by referring to previously obtained information and results. Common techniques for this are the creation and matching of exact or approximate fingerprints in the form of cryptographic or fuzzy file hashes, an examination of meta data including file header fields (cf. Section 2.1.2), file entropy analysis, collection and assessment of plaintext strings, or examination of Windows API imports. Most of these analysis steps can be widely automated, while interpretation of results may be performed manually, comparable to sandboxing.

Detailed analysis implies mainly the application of Binary Code Analysis techniques (cf. Section 2.1). This heavily involves an in-depth dissection of code using disassemblers such as IDA Pro [45] or Ghidra [46]. As the full disassembly of a program will frequently constitute of several hundred or thousands of functions, a key challenge is to maintain orientation and situational awareness in the binary and being able to direct focus to select areas of interest [47]. From a reverse engineering workflow perspective, it is advised to work towards defined analysis goals or hypotheses that capture narrow scope around specific functionality. A useful paradigm in this context is to perform top-down or bottom-up analysis that follows local data- or control flows and is anchored around cornerstones. These can be for example referenced strings or WinAPI functions that have tell-tale characteristics [38].

If applicable, decompilation can be used to recover (pseudo-)code that approximates the source code, which is significantly more accessible to humans than disassembled instructions and can also accelerate producing an understanding of the code's semantics [48].

2.4. Summary

In this chapter, we provided an overview of foundational topics relevant for this thesis. We first conceptually introduced Binary Code Analysis which is applied in all three main chapters. This included a procedural summary of compilation and an explanation of how executable Windows PE files are structured, because we conduct a comparative study across 839 malware families in Chapter 4 and investigate how malware interacts with the Windows API in Chapter 5. We then explained disassembly and code similarity, which are the main topic of Chapter 6 where we introduce approaches for both. Furthermore, we gave a general introduction to malware itself and its analysis, which is the central theme of this dissertation. Here, we focused on how Binary Code Analysis is specifically applied to malware in the form of dynamic and static analysis.

3. Related Work

In this chapter, we give an overview how the research presented in this dissertation is related to other work in the field. We cover the following areas: First, we discuss ground truth data sets created and used for malware research. Second, we present research analyzing how the Windows API is used in malware and how it can be used to infer potential capabilities of malware. Third, a survey of works on methods for disassembling of code is given. Fourth, we give an overview of research into measuring code similarity. Finally, we list research into malware analysis workflows and methodology.

Where appropriate, we also point out non-academic research as for multiple of the topics listed, outstanding work has been published outside of traditional academic formats.

3.1. Ground Truth for Malware Research

Ground truth is a term used to describe the "gold standard for assessing detection/classification algorithms" [49] and as such plays a fundamental role in experimental research. Naturally, reliable ground truth is highly relevant in malware research as well to produce representative results. In this section, we give a survey of how data sets have been created and used in malware research, covering both malware samples and meta data.

3.1.1. Collections of Malware Samples

Before discussing works that have been dedicated to malware data sets or used them in an outstanding way, we highlight publications that are concerned with how ground truth and experiments should be constructed and that looked at the impact and implications of violations of such good practices.

Most notably, Rossow et al. [50] published their set of prudent practices, a detailed collection of guidelines of considerations for sound experimentation with malware. They reviewed earlier works from 2006-2011 and analyzed their compliance with said guidelines or otherwise shortcomings. They conducted additional experiments to support their theses. One significant insight of this was that randomly sampled data sets usually tend to be highly imbalanced, e.g. 80% of malware samples in one data set belonging to about 10% of the families. Allix et al. [51] investigated the relevance of malware timelines for (Android) data set construction. They showed that this can have significant impact on machine learning-based detection schemes. Roy et al. [52] raised concerns about the quality and in particular age of input data used as data sets in (Android) malware classification studies. Pendlebury et al. [53] published TESSERACT, an open source evaluation framework which accounts for effects of temporal and spatial distorsion in data

sets. They reviewed three previously published Android malware classifiers and showed that they were affected by biases from the data sets they had been created and trained on. Miller et al. [54] conducted a study in which they evaluated the performance of a large-scale malware detection system across VirusTotal submissions for 2.5 years. While by itself achieving a 72% detection rate, they showed that the classifier performance could be improved to 89% with even 42% detections on otherwise undetected samples when given a budget for 80 human expert reviews on a daily basis. Additionally, they discussed the impacts that temporal inconsistency in labeling of training data may have a negative impact of as much as 20% on classifier performance. In more general terms, van der Kouwe et al. [55] systematically analyzed the benchmarking practices across 50 defense papers. They postulated a list of 22 "benchmarking crimes" and showed that they frequently occur even in publications at highly regarded venues. Abt and Baier [56] did a survey of 4 years of papers in the network security fields. They noted that 70% used manually compiled data sets, and only 10% of authors shared these upon request.

Upchurch and Zhou [57] proposed Variant, a malware data set created to serve as a gold standard for similarity testing of code. The data set consists of 85 samples grouped into 8 families, and samples have been unpacked to allow them to be directly compared. Nappa et al. [58] monitored 502 drive-by attack servers with exploit kits for one year. They collected the malware distributed by these servers, which resulted in the Malicia data set, containing 11,363 samples associated with 55 malware families. Rieck et al. [11] published their malware data set Malheur alongside their machine learning framework for behavorial malware classification. Malheur contains 24 malware families identified by 6 Antivirus labels each, it is limited to 300 samples per family, and families under 20 samples have been discarded during creation. Lin et al. [59] conducted a comparison of malware samples obtained through passive collection via honeypots with active collection from P2P filesharing. They noticed that the collection method influences the type of malware collected, i.e the samples from honeypots contained more bots, while P2P filesharing appeared to be used to primarily deliver trojans. Ceschin et al. [60] created a malware data set consisting of 50,000 samples used for targeting victims in Brazil. They argued that frequent adjustments to detection models are necessary to keep up with the evolution of threats. Grill et al. [61] investigated bootkits in particular, creating a collection of 2,424 samples covering 8 years. Ronen et al. [62] produced the data set used for the Microsoft Malware Classification Challenge (BIG 2015). It contains more than 20,000 samples for 9 selected families, including ample processed data to tailor it for easy use in machine learning experiments. Catak and Yazi [63] created and published a malware data set of 7,107 samples including their respective Windows API traces as generated using Cuckoo Sandbox. Barabosch [64] used samples from 102 malware families to study host-based code injection attacks. Anderson and Roth [65] published the EMBER2017 and EMBER2018 feature data sets, derived from 1.1 and 1 million PE files scanned in or before 2017/2018, annotated with the classes benign and malicious. The data set is intended for the creation and benchmarking of ML-based malware detectors working on PE file meta data. Harang and Rudd [66] provide the SOREL-20M data set, which in nature is similar to EMBER but more extensive. It consists of almost 20 million files with pre-extracted features and other meta-data as well as high-quality detection labels verified across multiple sources.

With respect to Android malware research, we note a number of works that have been dedicated to the assembly and publication of research data sets. In 2012, Zhou and Jiang [67] were the first to publish a collection of Android malware samples in their Android Malware Genome Project. This data sets consisted of 1,260 malware samples that were mostly repackaged applications with added malicious functionality. According to their estimation, they covered the majority of existing Android malware families (48) at the time. Arp et al. [68] released the Drebin data set. It consists of 5,560 applications and includes 179 malware families that were collected over a period of 2 years. Wei et al. [69] compiled the Android Malware Dataset (AMD). It consists of "24,553 samples, categorized in 135 varieties among 71 malware families ranging from 2010 to 2016". Allix et al. [70] built AndroZoo, a massive collection of currently more than 14 million Android Packages (APKs). The data set includes both goodware and malware, in case of the latter with their respective Antivirus detection labels. It was constructed by crawling app stores including GooglePlay and AppChina but also includes samples from Torrents and other downloadable locations.

Other Android data sets were created with a more specific focus. Maiorca et al. [71] specifically looked into obfuscation in the context of Android. They published the PRA-Guard data set, which consists of 10,479 samples which are obfuscated with 7 different techniques. Kiss et al. [72] created the Kharon data set, targeting in-depth research methods. It consists of 7 + 12 fully annonated Android malware samples. The Canadian Institute for Cybersecruity published several data sets for Android Malware in a specific context. Among them for example the work by Lashkari et al. [73], who contributed the Android Adware and General Malware (AAGM) Dataset: it consists of 250 adware samples (5 families), 150 malware samples (5 families) and 1,500 benign files. Alswaina and Elleithy [74] provided a survey on the state of Android malware detection, identification, and also the data sets used over time.

A number of works investigated how malware families developed over time, following their lineage. Calleja et al. [75] obtained the source code for 151 malware samples, spanning 30 years of malware development. They noticed an exponential increase (about one order of magnitude per decade) in complexity when measured in source code files, lines of code and functions per sample. Goldberg et al. [76] studied the derivation of phylogenic trees for malware and used directed acyclic graphs (DAG) to describe them. Their results were building on earlier work by Sorkin [77], who noted about 6,000 unique computer virus files at that time. Dumitras and Neamtiu [78] reasoned about the factors affecting lineage and provenance research. They identified several threats to the validity of this line of work, and made a case for validating results on benign open source software. Lindorfer et al. [79] tracked the development of 11 malware families over time. They analyzed the incremental change in the code over time, noticing that families primarily grew in size and functionality, which was particularly observed for Zeus. Jang et al. [80] studied and compared straight line and directed acyclic graph lineage. They applied their proposed method on a ground truth collection of 84 malware samples with known lineage from the Cyber Genome Project. They observed that PE header timestamps were a reliable data point to support lineage. Haq et al. [81] studied malware lineage. Using a data set of 7,793 samples from 10 malware families, they showed that by mapping samples to versions, a data reduction of 26 times can be achieved.

The Malpedia data set follows best practices and guidelines formulated in prior work by Rossow et al. [50]. It is unique in that it unifies meta data information and references for malware families, all of which are covered with representative samples. For most (Windows) malware samples, an additional unpacked state is provided in the form of memory dumps, which allows immediate processing using static analysis.

3.1.2. Analysis of Antivirus Detection Labels

Several works examined if Antivirus detections can be used as a reliable source for malware family classification labels.

Bailey et al. [82] were among the first to investigate if AV detection labels can serve as a reference for family classification by contrasting label clustering results with clustering based on behavorial information gained through dynamic analysis. Maggi et al. [83] studied the inconsistencies in naming across four AV products on 98,000 malware samples. An important observation by them was that closed subsets of names across AV products may not exist consistently, which implies that these labels may not suffice to properly validate clustering techniques. Mohaisen and Alrawi [84] evaluated the AV detections for 12,000 manually verified samples of 11 families. Among other results, they found that AV detections as a label source may inflate the number of families significantly, as the number of labels assigned across all engines was a median of 69 and average of 139.

Perdisci and ManChon [85] introduced a system for Validity Analysis of Malware-clustering Outputs (VAMO). They studied the effects of inconsistent labels in detail and showed that plurality vote outperforms majority vote for finding consensus across AV engines. Kantchelian et al. [86] reviewed ground truth labeling approaches used in more than 30 prior works and demonstrated that using confidence weights for different AV engines was beneficial. Gregio et al. [87] noted that traditional names such as Virus or Worm have become widely obsolete due to malware exhibiting manifold capability and instead proposed a behavior taxonomy. They also examined how generic Antivirus detection labels correspond to their chosen behavior classes. Sebastián et al. [88] proposed AVclass, a system for extrapolating family labels for the collection of AV detections of a sample, as aggregated on VirusTotal. Using predefined lists for generic tokens and family aliases, the AV detections are processed and reduced to a single consensus using plurality vote. The system achieved a F1 score of .939 on 8.9M samples combined from multiple data sets. Sebastián and Caballero [89] also proposed the improvement AVclass2, which no longer needs predefined lists but instead automatically derives class, family, behavior, and file properties from the aggregated AV detections, if possible, and builds a taxonomy based on the collective tags. Hurier et al. [90] intensively studied the potential disagreements in AV engine decisions for Android malware. This poses a challenge to building ground truth and data sets as they show that both choosing thresholds for detections or assigning different weights to vendors may introduce biases. Based on their findings, Hurier et al. [91] presented Euphony as a follow-up, a method to extract family names from AV labels and demonstrated its application on Android malware. The extraction method works without upfront definition of generic terms necessary in order to clean detections labels. The method builds weighted graph structures

encoding the names from multiple AVs which are reduced to trees and then clusters to derive one name per sample through majority voting. Ugarte-Pedrero et al. [92] gave detailed insight into how a day's worth of incoming malware files is processed by a security company. They explain how extensive filtering and grouping by behavior are used to minimize the number of samples to be analyzed manually and discuss limitations of state-of-the-art tools and implications for such analysis and decision pipelines.

Our approach for family tagging in Malpedia does not rely solely on Antivirus detection labels but instead uses the classification references provided by subject matter experts in analysis reports as one primary source of information. As a result, family names correspond closer to the identifiers as given and used by human analysts. Nevertheless, the data set is also augmented with Antivirus labels where suitable and possible.

3.1.3. Collections of Meta Data on Malware

Apart from data sets consisting of malware samples, a number of works have focused on collecting meta data on malware and threat actors that can be used for contextualization.

The Malware Wiki [93] is a public knowledge base in the form of a wiki that documents information about malware families. Freyssinet [94] studied botnets holistically, focusing on malware components, operations, and actors. The result was an extensive collection of meta data stored in a wiki accessible at `botnets.fr` and the formulation of a strategy to organize the fight against botnets. Bandla and Castro [95] maintain APTnotes, a GitHub Repository organizing public information such as blogs and reports on attacks with APT background. Roth et al. [96] collect publicly available information on APT groups and their used tools, including their respective aliases as given by security vendors. The MITRE ATT&CK framework [97] is an extensive knowledge base collecting information of adversary tactics and techniques. It also features a section documenting attacking tools and the threat actors they are associated with. The Malware Intelligence Sharing Platform (MISP) [98] was originally created as an effort to simplify the storage and exchange of Indicators of Compromise and is a widely adopted open source framework for exchanging intelligence at large. It contains so-called galaxy clusters, which are taxonomies that can be used for annotation of content. Among them are extensive and highly popular collections for malware and threat actors. The Council on Foreign Relations provides a Cyber Operations Tracker [99] that collects "publicly documented state-sponsored cyber activity since 2005". It documents source and target of attacks along with the type of incident, e.g. espionage or sabotage. ThaiCERT [100] maintains a collection called Threat Group Cards, that was originally published as a report but is now also available as a website. Their portal integrates and harmonizes information from several other sources, including MISP, MITRE ATT&CK, and Malpedia. Laurenza and Lazzeretti [101] created dAPTaset, a collection of meta data around APT activity. It imports raw data from sources like MITRE ATT&CK and MISP, parses this data for Indicators of Compromise and enriches them using further sources, among them Malpedia. Gray et al. [102] collected 896 reports on APT activity, which were then parsed in order to extract 15,660 hashes of malware associated with 164 threat actor groups. As a result of their efforts, the meta data set is made available to researchers upon request.

Our data set Malpedia is unique in that it not only collects meta data but also provides the actual underlying binary files as ground truth, in an unpacked representation where feasible. Malpedia has already been used as a data source by several other projects as indicated above.

3.2. Windows API Usage Analysis of Malware

One of the anchors of in-depth malware analysis is the examination of how the malware interacts with the system. Especially on Windows, the Windows API is an inescapable gateway to lower-level access to e.g. the file system and networking. Thus, it is also central to many proposed detection, classification, and analysis methods.

We will first provide a summary of other publications that have addressed the recovery of information about how programs import and use the Windows API. After that, we will give an overview of approaches to use this information for detection and classification of malware. In this context, we also highlight works that used this information to analyze potential malicious capabilities found in a program.

3.2.1. WinAPI Usage Recovery and Deobfuscation

Suenaga [103] provided an extensive overview of different obfuscation techniques for Windows API calls. Apart from runtime API address resolution, advanced concepts involving control flow redirection are discussed. O'Meara [104] investigated and documented a case where API name hashing was used in the malware family `win.heriplor`, associated with threat actor Energetic Bear. They showed how the obfuscation method can be used to systematically identify more samples of the family by searching through a big malware repository.

Sharif et al. [105] presented a method for dynamically resolving WinAPI usage in the context of their analysis framework *Eureka*. They described how a lookup table for WinAPI functions and their corresponding memory addresses can be generated using dynamic analysis by evaluating the loaded modules in a process memory space and then resolving use of the WinAPI by checking addresses against this table. Raber and Krumheuer [106] presented an approach called *QuietRIATT* for reconstructing an Import Address Table. Their method uses a modified version of Microsoft Detours [107] to record calls into the WinAPI, which can be processed using IDA Pro [45] and ImpREC. Xi et al. [108] proposed a framework called ADSD (API Deobfuscation based on Static and Dynamic techniques). They perform slicing to locate calls to `kernel32.dll!GetProcAddress` and then use emulation to resolve the respective WinAPI reference. Choi [109] presented a method for dynamic API deobfuscation using memory access analysis. By tracking read and write operations, both direct and indirect calls using obfuscation can be analyzed in order to resolve their respectively referenced WinAPI functions. Korczynski [110] presented RePEconstruct, a method and tool to automatically unpack malware using the dynamic binary translation framework DynamoRIO. It can also rebuild Import Tables. Kawakoya et al. [111] proposed

a method for reconstructing Import Address Tables in cases where position obfuscation [112] is used to masquerade the position of DLLs in the memory layout. Kotov and Wojnowicz [113] presented a method to recognize the usage of Windows API based on their passed arguments in order to generically overcome usage obfuscation. For argument recovery, they used symbolic execution and a vector representation to represent them for matching via Hidden Markov Models. A limitation of their approach is that they only considered functions with 3 or more arguments and targeted 25 selected WinAPI functions only.

With regard to practical methods based on static analysis for WinAPI usage recovery from memory dumps, the current state of the art are Scylla [114], which is the successor of ImpRec, and ImpScan [115], a plugin for the Volatility memory forensic framework. Our approach ApiScout is a generalized method of the approach proposed by Sharif et al. [105]. It is more robust than Scylla and ImpScan as it does not assume a single coherent Import Address Table. Instead, it scans for groups of two or more individual WinAPI references and properly validates recovered entries, resulting in almost no false positives and false negatives.

3.2.2. Malware Detection and Classification by Analysis of WinAPI Usage

Many approaches that involve WinAPI usage information in the context of detection and classification of malware have been proposed. Most of these rely either on static or dynamic analysis, which is why we use this criterion to organize the following summary.

Starting with dynamic analysis, Christodorescu et al. [116] introduced one of the first formal concepts for malware detection based on semantic abstraction and interpretation of behavior. Their framework is based on templates that consist of instruction sequences in which variables and symbolic constants are translated into an intermediate representation. This makes their matching of templates robust against compiler artifacts and obfuscations like instruction reordering, register renaming, or garbage insertion. Christodorescu et al. [117] followed up their work with the presentation of a prototype. Their examples and explanations demonstrated a strong reliance on WinAPI usage as a solid semantic anchor for behavior specifications. They then performed mining on a number of malware samples and extracted traits and behaviors that can be used both for describing and matching malicious behavior. Preda et al. [118] additionally presented a proof for the stability of patterns extracted via the approach of Christodorescu. Chen et al. [119] further improved upon the work by Christodorescu et al. [116, 117] by adding summarization of graph patterns for attack behavior which optimized it to work better on polymorphic families. Frederikson et al. [120] continued this line of work by finding optimizations for discriminitive behavior specifications. Again relying mostly on WinAPI usage sequences, specifications are abstracted and synthesized to better generalize their detection potential.

Hu et al. [121] proposed a tracing system called Argus to monitor WinAPI usage by programs. Based on a 35-dimensional vector with each entry representing presence of one potentially malicious behavior, they performed detection of malware. Liu et al. [122] defined 35 behaviors based on WinAPI sequences that can indicate malicious programs,

including creation of files or modification of programs to be started on OS initialization. Bayer et al. [123] provided an overview of malicious behaviors observed across 901,294 submissions to their analysis system Anubis. These behaviors were derived from WinAPI call sequences recorded and interpreted by the system.

Rieck et al. [124] used machine learning to train models for the detection of malicious behavioral patterns based on sandbox traces. They presented a method for feature extraction and embedding in order to translate characteristic WinAPI interactions into a vector space, upon which a SVM can be applied. They used a corpus of 10,000 malware samples from 14 families and showed that the method provides high reliability. Trinius et al. [125] defined a new representation to express observable behavior they call *Malware Instruction Set* (MIST). MIST instructions translate and abstract WinAPI calls and their processed arguments into a more space efficient binary format. Because of this, storage overhead of sandbox traces is significantly reduced while also allowing better embedding into vector spaces, e.g. for clustering malware by similar observed behavior. The approach covers 130 WinAPI functions, categorized into 20 groups. Rieck et al. [11] did a second study of machine learning applied to malware classification, using MIST to embed observed behavior into a high-dimensional vector space. Using an incremental approach to combine clustering and classification, they showed a significant improvement in accuracy over the results presented in their previous work, while simultanously reducing the memory required and increasing the processing throughput. Kolbitsch et al. [126] used behavior graphs consisting of WinAPI interactions to model specific program activity. Initially, these graphs are extracted from runtime intruction traces recorded during the execution of malware in the analysis system Anubis. After creating a reference set of behavior graphs, they showed that these graphs can be effectively used for endpoint protection when comparing them against execution behavior of malware. Apel et al. [127] compared the performance of four different distance metrics when used for clustering dynamic execution behavior traces, consisting of WinAPI function calls. Cheng et al. [128] applied information retrieval methods to the classification of malware. They first recorded sequences of WinAPI calls via Cuckoo sandbox [129] as behavioral features. These sequences were then processed with an irrelevance reduction method to isolate significant behaviors which were then used for similarity analysis.

Ki et al. [130] used Detours [107] to enable WinAPI call sequence analysis in order to detect malware. By tracing 23,080 samples, they found 2,727 WinAPI functions and categorized them into 26 classes represented by letters A to Z. After normalizing WinAPI call sequences into sequences of letters, they then used methods known from DNA sequence alignment (such as multiple sequence alignment and longest common subsequence) to isolate characteristic sequences for malware. These were then used to classify unknown samples. Gupta et al. [131] used a similar approach, hooking 534 WinAPI functions and mapping them to 26 semantic classes identified by letters A to Z. The translated WinAPI call traces were then compared to each other using ssdeep [132].

Anderson et al. [133] used dynamic and static analysis in combination to improve the classification of malware. One of their 6 data sources were dynamic system call traces, in which they recorded 2,460 distinct WinAPI functions used across 1,556 traces. They then grouped these WinAPI functions into 94 semantic categories, primarily to reduce

dimensionality for their clustering. Shijo and Salim [134] also leveraged a combination fo static and dynamic analysis for the detection of malware. They derived WinAPI call n-grams from dynamic traces and combined this data with information on statically extracted strings.

With regard to approaches relying exclusively on static analysis, Schutz et al. [135] were among the first to exploit information about a program's interaction with the Windows API to detect potentially malicious behavior. They extracted information about DLLs and APIs used and represented this information as three different kinds of boolean (DLL and API presence) and integer vectors (APIs used per DLL). These vectors were used to derive rules for detection. Lu et al. [136] studied the applicability of different machine learning algorithms to the problem of malware detection. They used a mix of content- and behavior-based features, with content-based features incorporating the presence of calls to WinAPI functions. For this, they statically extracted all entries from the Import Tables across 1,200 malicious and benign programs and found 2,682 different WinAPIs being referenced. Based on this collection, they constructed a binary vector to encode the presence of all API functions. Sami et al. [137] statically extracted WinAPI import information from 65,000 binaries. They observed 44,605 distinct WinAPI functions being used and used frequency analysis to derive a subset of these functions to be used in a binary vector representation over which malware detection was enabled. Sathyanarayan et al. [138] defined a set of critical WinAPI functions that are often observed to enable malicious functionalities. Using IDA Pro, they extracted call frequencies for these critical WinAPI functions and used a vector representation as a profile for a given program. They then used a Chi-square test to decide whether or not a vector belongs to a class of malicious programs. Baranov et al. [139] used symbolic execution to analyze System Call Dependency Graphs (SCDG), i.e. potential sequences of WinAPI interactions, and classify malware.

Alazab et al. [140] proposed a method to derive information about potential malicious behavior by statically analyzing a program's Control Flow Graph and WinAPI interactions. They defined 6 behaviors encapsulating 76 WinAPI functions and evaluated their presence across 386 malware samples. Alazab et al. [141] followed up with another related approach that statically analyzed the frequency with which certain WinAPI functions were called.

Shafiq et al. [142] showed that structural information extracted from PE files can be used to detect malware. One of their features was the presence of references to the Windows API through usage of the network enabling DLL files `wsock32.dll` and `wininet.dll`. Beaucamps et al. [143] presented a method to apply model checking for malware detection. They introduced a generic framework that abstracts behavior from its concrete implementation using string rewriting. In examples they outlined how sequences of WinAPI interactions can be described with labels, which then in turn can be combined to increase the expressiveness. This was then applied to the analysis of dynamic program traces produced by using Intel Pin [144]. In a follow-up work, Beaucamps et al. [145] expanded their framework to also cover parameters and thus involving data flow analysis as well as allowing interleaved abstraction patterns. In the course, they also explain how this extension allows application for static analysis.

Zwanger and Freiling [146] explored the spectrum of APIs used by binary program code running in kernel mode. By defining 19 semantic groups for these APIs and measuring their usage frequency in histograms, they were able to discriminate programs of different characteristics such as drivers for hardware, the filesystem, and network versus rootkits.

Caillat et al. [147] proposed *Prison*, a method to monitor inter-process communication, which also allows to detect and mitigate malicious behavior. It is implemented via hooking system services in the Windows kernel by patching the System Service Dispatch Table. The monitoring and interception is tailored to relevant WinAPI functions tied to suspicious behavior and their respective entry points to the kernel. Kirat and Vigna [148] introduced *MalGene*, a system to automatically extract malware analysis evasion signatures. The system is based on sequence alignments over WinAPI calls, for which similarity is measured using methods known from bioinformatics. Mohaisen and Alrawi [149] presented *AMAL*, a system for "high-fidelity, behavior-based automated malware analysis and classification". The approach is separated in two stages. First, malware samples are sandboxed and artifacts related to file system, registry, network, and volatile memory are extracted. Some of these artifacts are interactions with the Windows API. Second, after mapping these artifacts into corresponding feature vectors, these vectors are clustered and then have labeled samples injected to map clusters to families.

Tamada et al. [150] presented a method to encode watermarks into software based on the sequence and frequency of Windows API calls. Choi et al. [151] proposed a similar method depending on the Windows API call sets that capture API usage over all functions found in a program. Guan et al. [152] statically extracted sequences of system functions and evaluated the applicability of frameworks used for phylogeny and protein alignments to measure similarity across programs.

Other works have examined if and how a program's interaction with the Windows API can be interpreted to provide human analysts with an overview of expected functional capability and jumpstart their in-depth analysis.

Comparetti et al. [153] raised awareness on the fact that sandboxing will often only capture certain behaviors but will not provide information about so-called dormant functionality. They proposed Reanimator, a system that uses program slicing to locate presence of code that matches previously identified genotype features for certain behaviors that they call malspecs. Guevara [154] proposed a method for semantic exploration of binaries. The method relies on a-priori defined sequences of WinAPI functions associated with certain malicious behaviors that are matched against the interprocedural Control Flow Graph of a program to locate and highlight these behaviors. Oosthoek and Doerr [155] applied MITRE ATT&CK behavior recognition on the Malpedia data set by using an industry state-of-the-art malware analysis framework, Joe Sandbox [156]. Ballenthin et al. [157] published capa, a framework to encode and match rules for known (malicious) program behavior. These rules can be defined to e.g. require the presence of certain instructions, constants, but also WinAPI functions to indicate behaviors. Capa uses our disassembler SMDA (presented in detail in Section 6.2) as its primary disassembler backend for Python3. Alrawi et al. [158] presented FORECAST, a system using

symbolic execution for detection of malicious capabilities in memory images of malware which also estimates their execution likelihood.

Two representations to assess and compare WinAPI usage commonly used in practice are *ImpHash* [159] and *ImpFuzzy* [160]. Instead of a vector representation, both use a concatenation of the WinAPI DLL and API names which is then transformed using cryptographic (ImpHash) or fuzzy (ImpFuzzy) hashing.

Our approach ApiVectors picks up the idea of a vector representation to encode the usage of WinAPI functions, as previously proposed e.g. by Schutz et al. [135] and Lu et al. [136]. However, we do not only consider entries from the Import Table like previous works but also incorporate those entries usually protected by Windows API usage obfuscation, i.e. dynamically created WinAPI references during runtime, which can be recovered by ApiScout. Performing our analysis across a data set as diverse as Malpedia allows us to derive a very representative vector from the 4,994 distinct WinAPI functions we found being used. Because we provide a semantic categorization for all of the observed WinAPI functions (which is almost twice as comprehensive as previous works by Ki et al. [130] and Anderson et al. [133]), we are also able to ensure that the selection in the vector consists of WinAPIs that have high practical relevance to analysts.

Instead of using the vector representation for machine learning-based classification, we propose the use of a similarity measure based on a weighted Jaccard index. The weighting allows to emphasize less commonly used WinAPI functions, which are thus potentially more characteristic for a malware family. This is one of the reasons why ApiVectors achieves better classification results than the comparable approaches ImpHash and ImpFuzzy, which instead only use equal weights and project the WinAPIs using non-reversible hash functions prior to comparison. Maintaining the ApiVector representation for comparison also provides inspectability of the procedure, increasing the interpretability of results.

3.3. Code Analysis

As malware is typically delivered as a compiled program, in-depth analysis is usually required to be carried out on binary code level. An initial step in this context enabling all further analysis is the identification and interpretation of binary data into instructions and the structure they are arranged in, a procedure referred to as disassembling. As the produced disassembly serves as a base on which other analysis techniques are conducted, it should be as complete and accurate as possible.

In the following, we will first give an overview of related work on disassembly. After this, we will address the field of code similarity analysis, which is highly relevant in the context of malware analysis as it is a primary tool to accelerate analysis and provide context.

3.3.1. Disassembly

Linear sweep disassembly is the straightforward approach to disassembly and is based on the assumption that instructions are located in a sequence within a continuous code section. Application of this technique turns an individual instruction decoder immediately into a disassembler and is implemented e.g. by OBJDUMP [161]. While being a very fast method, it is susceptible to data being mixed into the instruction stream, like jump tables or strings.

Among the first to define a recursive traversal algorithm for disassembly were Sites et al. [162]. They used it to extract code and be able to translate it for other instruction sets, benefiting from performance advances on other architectures. Cifuentes and Van Emmerik [163] used a form of recursive traversal in their system UQBT, applying it in the context of binary rewriting. They [164] also presented a method for the recovery of n-conditional branching when implemented as a jump table. It is based on program slicing and aims at locating and evaluating the offset table containing all target addresses. Their method recovers the sought information for more than 89% of the cases. Another early work describing recursive traversal was by Theiling [165], who proposed using a bottom-up instead of top-down approach for control flow graph extraction. Their disassembly algorithm allows to better account for uncertainty and ambiguity in a number of special cases, including switch tables or functions without a dedicated return instruction. Schwarz et al. [166] suggested to combine the advantages of linear sweep and recursive traversal into a hybrid disassembler algorithm, that uses linear sweep for instruction recovery while using recursive traversal for validation of function boundaries. Linn and Debray [167] revisited recursive traversal and proposed obfuscation techniques to thwart static analysis. De Sutter et al. [168] defined techniques for the reconstruction of indirect control flow transfers when recovering and restructuring CFGs from binary programs. They achieved a target recovery success rate of above 90%, consequently allowing to reduce the code size missed by a higher degree than before.

Meng and Barton [169] studied the frequency with which complex code constructs, such as jump tables, tailcalls, or overlapping instructions occur in compiled code. Andriesse et al. [30] performed a comparison of nine state-of-the-art disassemblers on an extensive data set of both Linux and Windows binaries compiled with varying compilers and settings. They also addressed the prevalence of complex constructs and how they affect disassemblers. Additionally, 30 publications were reviewed and a mismatch between the expectations and presented results in the literature was noticed, especially with regards to accuracy and reliability of instruction recovery (underestimated) and function start recovery (overestimated). Andriesse et al. [31] followed up their investigation with the presentation of Nucleus, an algorithm for compiler-agnostic function detection. The approach uses a bottom up methodology, first identifying sequences of instructions as basic blocks and then inferring function structure from control flow connections between these components. Di Federico et al. [170] proposed Rev.NG, a framework for disassembly based on reaching definitions analysis applied to code lifted into an intermediate representation. This allowed them to apply the same set of techniques for multiple architectures. Pang et al. [171] did a systematic study and overview of techniques used

in (open-source) disassemblers for different aspects such as linear sweep versus recursive traversal, resolution of cross-references, indirect jumps, jump tables, etc.

A line of works has studied the applicability of machine learning in the context of disassembly, for both instruction and function border detection. Rosenblum et al. [172] used machine learning for the identification of function entry points (FEP). They defined a model for Conditional Random Fields (CRF) using idioms of up to four instructions as well as call and overlap information. Their approach achieved a notable improvement over FEP detection in Dyninst [173] and IDA Pro. Wartell et al. [174] addressed the challenge of differentiating data and code. For this, they interpreted x86 binaries as a sequences of bytes and used machine learning to train a language model that allowed them to classify these as code or data. Bao et al. [175] defined a method called ByteWeight and applied weighted prefix trees over instruction sequences for the identification of FEPs. This allowed them to recognize longer sequences of up to ten instructions as function prologues. They used normalization on instructions by wildcarding parts of their operands. Shin et al. [176] used recurrent neural networks (RNN) and showed that this allows to notably reduce the time required for training while achieving similar or slightly better results than with ByteWeight. Pei et al. [177] demonstrated the applicability of transfer learning to the recognition of instructions and function borders. They showed that even when only trained on unoptimized binaries, their method XDA still worked well on optimized binaries.

Other research set its analysis scope on code protected against analysis. Kruegel et al. [178] discussed improvements to static disassembly when targeting binaries that are obfuscated. Apart from iterative refinement and conflict resolution of the CFG and basic blocks, they also used gap completion to locate additional code missed by the initial disassembly. Harris and Miller [179] focused on the analysis of stripped binaries. They expanded their model for function representation by adding shared-code and multi-entry functions. After an initial phase of breadth-first recursive disassembly, they similarly scan potential gaps for well-known function prologues. They reported an improvement of accuracy and completeness of recovery over results provided by IDA Pro. Bonfante et al. [9] presented CoDisasm, an approach that is capable of dealing with self-modifying code including overlapping instructions. They combined concrete path execution, used to capture waves of codes, with static disassembly in which they piece the observed pieces together. Scope of their evaluation was the application to malware unpacking and identification of layers in packers.

Further use cases for code analysis and disassembly were also demonstrated. Wang et al. [180] demonstrated their approach Uroboros, which allows to disassemble code in a way that enables assembling it back correctly into executable programs. The challenge specifically addressed in this work is to handle relocation of code accurately, which they solved successfully. Caballero et al. [181] examined possibilities for automated binary code reuse. They proposed a method to identify the interface of self-contained code blocks including instructions and data dependencies and to extract it for external instrumentation. As use case, they showed how to infer and rewrite adapters for the C&C protocols used by the MegaD and Kraken botnets. Chua et al. [182] focused on the application of machine learning for derivation of function type signatures. In their

system Eklavya, they trained neural networks for this task and achieved an accuracy of above 80%.

Our approach SMDA that we present in Section 6.2 builds on several of the presented works [178, 179, 172, 175, 170, 31], reusing ideas for recursive disassembly, using prologue- and call-destination-based heuristics for function entry point detection and filling remaining gaps with linear disassembly. The novelty aspect of our work is that we specifically focus on memory dumps instead of unmapped executables. In the course, we demonstrate that high quality disassembly can still be obtained with these methods without the need to rely on further pointers for code constraints from meta data, such as section, relocation, exception handler, or symbol tables, and also that the output quality of other disassemblers suffer when they do not have access to this information.

3.3.2. Code Similarity Analysis

(Binary) code similarity analysis is a technique that enables the comparison of given pieces of binary code. Its use cases include the detection and analysis of malware but also bug search and differencing of patches, e.g. in the context of vulnerability research. It is usually assumed that source code is not available and methods are applied on either function or whole binary level, less often also on basic blocks. Haq et al. [33] provided a comprehensive survey of research on this topic.

Among the first to describe a code similarity analysis approach were Baker et al. [183]. They proposed a method called Exediff, which allowed to construct delta files, i.e. small change sets of code that can function as patches with low size footprint. Their demonstration was applied to DEC Unix Alpha executables and provided specific methods for handling both text and data segments. Another early work was presented by Wang et al. [184], who introduced BMAT, an approach to match versions of executables with the purpose of propagating existing performance profiling information to new builds. In a first step, code is matched on function level using both meta data information (symbols) and hashing over code. Then the candidate function pairs are further compared on basic block level. Carrera et al. [185] used an adjacency matrix defined over all calls between functions in a program for similarity analysis. They further defined control flow graph and call-tree signatures and used them for matching. Schulman [186] split up the output of a disassembler at the identified function boundaries and used hashing to create a database of known functions. Cohen and Havrilla [187] analyzed code duplicates in a large collection of binaries using code hashes. Working on function level, they compared hashing the binary code directly as given in the program with hashing a processed version in which they wildcarded addresses, leading to position independent code (PIC). They particularly observed that hashing the PIC representation will rarely introduce additional false positives over direct code hashing.

Farhadi et al. [188] defined a taxonomy for code similarity methods, in which they divided them into text-based, token-based, metrics-based, structural-based, behavorial-based, and hybrid approaches. In their taxonomy, text-based approaches are applied on whole binary level, with the system BitShred as introduced by Jang et al. [189] given as example. BitShred uses a bit vector of mixed feature, containing information obtained

through both dynamic and static analysis. The collection of vectors is then processed using co-clustering and Jaccard distance as a similarity measure.

Token-based approaches split a given code representation into subsequences which are then used for comparisons. A very early work using tokens was the study by Goldberg et al. [76]. They observed that for computer viruses that are derived from each other, that they will likely contain sequences of 20 or more identical bytes. Goldberg et al. defined these sequences as characteristics and used a directed acyclic graph to capture the characteristics of certain virus species. Karim [190] compared the utility of n-grams versus n-perms (n-permutations, i.e. sorted n-grams) for matching code and found that n-perms performed better. Walenstein et al. [191] also suggested the use of n-grams and n-perms and introduced weighting based on inverse document frequency. Saedbjornsen [192] normalized instructions by performing abstraction over the operands. Instead of n-grams, a window approach was used to capture instructions in a given region, that were then transformed into a vector suitable for matching using a bagging technique. Upchurch [193] used the locality sensitive hashing (LSH) method Minhash to efficiently index n-grams for code similarity analysis. Tahan et al. [194] described a similarity analysis method using n-grams which involves both benign and malicious software. Here, the benign code is used to filter software to be analyzed, allowing to match only suspected malicious components against each other. Hassen and Chan [195] used the the call graph extracted from malware samples for matching. They also used an intermediate step in which they performed clustering and derived cluster ids using instruction n-grams of sizes one, two, and three. Raff and Nicholas [196] showed that when using n-grams for classification and being interested in extracting the most frequently occuring n-grams, they can be processed using hashing to achieve a speedup of one to two magnitudes.

With respect to metrics-based approaches, Bruschi et al. [197] proposed a method to numerically summarize certain features of functions, e.g. by counting the number of instruction, blocks, or specific instruction types. Miller [198] used metrics such as number of parameters, incoming and outgoing references, and stack frame size to define a signature for functions that can be used for comparisons. Eschweiler et al. [199] thoroughly evaluated the expressiveness of metrics-based features and proposed DiscovRE as a method using kNN clustering for nearest neighbor search to locate candidates of functions to be then compared with other in-depth methods.

A significant work in the class of structural-based methods is the graph-based approach to function matching that was proposed by Dullien [200], which also uses the call graph to determine matching candidates in two programs to be compared. A follow-up work by Dullien and Rolles [201] again focused on graph isomorphism but proposed improvements to the matching on instruction, basic block, and function level. By using the Small Primes Product as a hashing method, the method also introduced robustness against instruction reordering in basic blocks. Kruegel et al. [202] proposed a fingerprinting method for the detection of polymorphic worms. They used colored k-subgraphs, in which the color was based on instruction semantics, defining classes such as data transfer, arithmetics, stack, or floating point instructions. Cesare et al. [203] presented Malwise, which uses a flowgraph representation for functions and set similarity to compare flowgraphs contained in malware samples. Ding et. al. [204] created Kam1n0, an

approach for scalable asssembly code clone search. They used an adaptive locality sensitive hashing scheme to approximate Nearest Neighbors, using instructions represented as mnemonics and operands as well as n-grams of these as features. Exploiting the sparsity of links in control flow graphs, they were able to efficiently implement subgraph clone search in a MapReduce algorithm, working on basic block level and merging upwards into functions. Huang et al. [205] used Minhash to encode features extracted from longest path generation and path exploration for code similarity.

Considering behavior-based methods, Leder et al. [206] proposed to use Value Set Analysis to abstract code semantics and behavior from their concrete implementation and demonstrated the applicability to the detection of metamorphic malware. Jin et al. [207] introduced the concept of semantic hashing. On basic block level, they used pseudo-randomly generated assignments of register states to compute input-output pairs for the behavior of basic blocks. In order to make this semantic representation usable for matching, Jin et al. were the first to show that the concept of MinHashing [208] is well suitable to be applied in the context of code similarity. Lakhotia et al. [209] presented their approach BinJuice, in which semantics of basic blocks are captured by extracting algebraic and type constraints. Egele et al. [210] used blanket execution as a dynamic analysis technique in their PIN [144] tool BLEX to perform semantic feature extraction. They showed that this method can provide higher discriminatory power than other approaches. An extension to cross architecture code matching in the context of bug search was presented by Pewny et al. [211]. They lifted code from ARM, MIPS, and Intel into a common intermediate representation, over which they calculated input-output pairs that were processed using Minhash, similar to Jin et al. [207].

The taxonomy also lists hybrid approaches that combine multiple of the previously listed methods. The approach by Wang et al. [184] as mentioned earlier falls in this category, but also a work by Khoo et al. [212] that uses code abstraction, n-grams, and function subgraphs for matching. Alrabaee et al. [213] used opcodes but also control flow graph walks as features for code similarity analysis. They were also among the first to conduct an extensive study of usage of third-party libraries in malware. Dullien [214] proposed the use of SimHash [215] for code similarity analysis. Features used for their system FuncSimSearch were subgraphs of the control flow graph as well as n-grams of mnemonics.

In addition to these categories, new approaches based on encodings and embeddings, such as the natural language processing techniques *word2vec* by Mikolov et al. [216], were proposed in recent years. Feng et al. [217] created Genius. In their system, control flow graphs annotated with structural and statistical features are first used to generate codebooks. These codebooks are then used for feature encoding of functions, which allows them to apply locality sensitive hashing for search and comparison. Gemini was proposed by Xu et al. [218] as an improvement of Genius, which was the first neural network-based approach for generating embeddings of binary code. They call their embedding structure2vec and process a similar annotated control flow graph structure as defined for Genius. They showed that the method is significantly more accurate than Genius and can be trained three to four magnitudes faster. Massarelli et al. [219] presented their system SAFE, in which they used word2vec to map instructions to vectors,

which are then summarized using a self-attentive neural network. They show that this setup is capable of outperforming Gemini. Ding et al. [220] proposed an adaption of word2vec they call asm2vec. It is tailored to capture syntactic context of instructions by encoding mnemonics and operands for the processed instruction as well as its preceeding and successive instructions. In order to represent full functions, the function's control flow graph is modeled using edge coverage sequences and random walks over sequences of instructions. Ding et al. showed that their technique performs better than approaches based exclusively on graphlets or n-grams.

Our approach MCRIT performs code similarity analysis on function level and builds on several well-proven concepts from prior work. It uses both token- and metrics-based features, whose influence can be weighted to control their impact on the matching. The indexing itself is implemented using the LSH scheme MinHash for which a series of parameterizations are evaluated in detail. We continue the work of Alrabaee et al. [213] by performing an analysis of third-party library usage in malware over a large and representative data set, Malpedia. For the analysis of code overlap across malware families, we incorporate the idea by Tahan et al. [194] to exclude known library code to focus on code intrinsic to malware families.

3.4. Malware Analysis Methodology and Workflows

Chapters 5 and 6 address concrete methodology of static malware analysis. The specific aspects covered in these chapters are chosen on what we – based on personal experience – believe are among the most crucial areas of in-depth analysis, as also outlined in our work on a cooperative malware analysis workflow [38]. The following provides an overview of related work analyzing or proposing malware analysis methodology.

Bryant [221] conducted a fundamental study of the procedures applied by reverse engineers when recovering a semantic interpretation from assembly code. It is specifically highlighted how representational gaps complicate the task of interpretation and abstracted the concepts and procedures as described by multiple subject matter experts into coherent methodology. Pucsek et al. [222] proposed the Integrated Comprehension Environment (ICE), which aims to transfer tools known from IDEs for high-level languages to low-level representations. This includes a series of components, e.g. a Cartographer module that uses the call graph to provide orientation above the function level that many analysis tools usually provide as primary interface. Baldwin [223] extensively studied how analysts achieve program comprehension on assembly code and how to support this procedure. This also included an extensive user study conducted by Baldwin et al. [224] to identify requirements and issues that two expert groups regularly faced during their work. Votipka et al. [47] performed interviews to investigate the decision making procedure of reverse engineers for both vulnerability and malware analysis. They observed a methodology split in three phases across all study participants, divided into overview, sub-component scanning, and focused experimentation. They noticed that static analysis dominates in the first two phases, while focused experimentation often involves dynamic analysis (cf. Section 2.3).

Nguyen and Goldman [225] presented their methodology for Malware Analysis Reverse Engineering (MARE). It covers a full analysis workflow divided in four phases, from detection over unpacking to in-depth analysis including behavior analysis and reverse engineering. Higuera et al. [226] proposed a Systematic Approach to Malware Analysis (SAMA). The method lists a general sequence of tasks typically to be carried out when working on a malware case. It gives detailed workflows including feedback loops to guide the analysis procedure.

Kim et al. [227] proposed a framework to support the procedure of attributing malware families to actor groups by using human and machine collaboration. A number of descriptive features like printable strings or modified registry keys was extracted and used to train a kNN classifier. Then, a human analyst was presented with a visual representation of closeness for outliers that could not be classified automatically. In all cases, the subject matter experts were able to increase the overall classification accuracy with the presented information.

Obrst et al. [228] defined the Malware Attribute Enumeration and Characterization (MAEC) language. MAEC is an ontology that allows structured documentation of malware by referring to standardized behaviors and capabilities.

3.5. Summary

This chapter gave an overview of relevant prior work published on the topics covered in this dissertation. It allowed us to demonstrate how our efforts and results relate to others and which novelties our work provides.

With Malpedia, we provide the most comprehensive data set to date in terms of cleanly labeled, representative samples for 1,136 malware families including an unpacked representation for a large majority of them. Strictly following a set of requirements in line with previous guidelines for best practices [50], it is ensured that this data set can serve as a solid foundation for in-depth analyses of malware. Additionally tracking meta data information for the malware families provides practical context to support research. The usefulness of the data set is demonstrated through an investigation of the applicability of prominent analysis methods on memory dumps.

By expanding an existing approach [105], we first propose ApiScout as a robust method for the recovery of references to the Windows API and show in an evaluation that it outperforms the current state of the art [114, 115]. Using our method on Malpedia, we quantify that dynamic WinAPI imports are found in almost 50% of malware families analyzed. In order to define a representative vector to encode WinAPIs used by malware (similar to [135, 136]), we create a semantic classification scheme more extensive than what was used before [130, 133], providing detailed behavioral characterization of malware. We also show that this vector representation allows a more granular comparison and identification of malware based on WinAPI usage information than comparable methods [159, 160].

With regard to the analysis of code, we show that current disassemblers struggle to recover code from memory dumps and build upon previous work [178, 179, 172, 175,

170, 31] to propose improvements to function entry point recognition and code recovery, producing more robust results. Next, we continue to study code similarity using both exact [187] and fuzzy hashing and propose a scalable, Minhash-based code similarity framework called MCRIT for which we evaluate the effectiveness of mixing token- and metrics-based features. Inspired by [213], we investigate the use of third-party libraries in malware. Using MCRIT on Malpedia and a set of 53 FOSS libraries, we locate about 20% of code likely associated with libraries, which is in line with what was reported by Alrabaee et al. [213].

We will revisit details of selected publications for comparison and discussion in the course of the following chapters.

4. Malpedia: A Representative Corpus for Malware Research

In this chapter, we discuss the perceived lack of solid, quality ground truth for (Windows) malware research and present our contribution in this context: *Malpedia*.

We first start with a motivation and follow up with a definition of related research questions and our contributions in Section 4.1. In Section 4.2 we then define a set of requirements that a malware corpus tailored for static analysis should follow and compare them against the Prudent Practices outlined by Rossow et al. [50]. Next, in Section 4.3 we present Malpedia, a reference corpus following the requirements introduced and give an outline of its contents. Malpedia is valuable for multiple use cases, allowing to study a wide range of different malware families in great detail from different perspectives. In this chapter, we demonstrate this by performing an extensive structural analysis of the unpacked payload malware in order to gain an insight into malware author behavior in Section 4.4. Finally, we conclude the chapter with a summary in Section 4.5.

This chapter follows in large parts our previously published results as presented in [12] but significantly expands in the number of covered malware families, which is more than twice as large as in the original publication.

4.1. Motivation and Contribution

Malware remains a significant threat to the integrity of computer systems and networks. The time around the years 2006-2007 marks a significant changing point in the evolution of malicious software for multiple reasons. First, at this time polymorphic runtime packers were omnipresently adopted, becoming a drastic game changer in the discipline of malware detection, as can be inferred from AV-TEST's collection statistics [35]. Second, from this time on, malware was increasingly economized, introducing specialized services for many aspects of conducting cybercrime operations. Third, fraud using infamous banking trojans promising immediate financial gains was a new attack model that boosted the popularity of financially-oriented malware [34]. Ever since, malware has received increasing attention by researchers.

Reviewing academic malware research of the last 15 years, it becomes fairly obvious that there is still a lack of quality reference data to conduct experiments on. As was summarized in Chapter 3 on related work, many publications that covered Windows malware used data sets consisting of only a few malware families that also appear in parts heavily outdated when comparing their first spotting in the wild versus the publication date. Other reseachers tried instead to compensate the aspect of coverage by using

large data sets with near or completely unknown and undocumented composition. This raises serious questions on the expressiveness of the studies conducted as malware is a fast-paced, everchanging field.

One inherent challenge to research is the omnipresence of packers that immensely inflate the number of uniquely observable malware samples. Bonfante et al. observe 93% packed samples in the data they used [9] and Calvet et al. [10] 98% packed samples. Therefore, packers are an important aspect in the context of detection, and they are an effective hindrance for actual malware research on their carried payloads, which is usually referred to as the actual malware families. Many packers carry great complexity by abusing barely documented operating system intrinsics to thwart detection (e.g. `win.smokeloader` [229]), posing a significant challenge to industry professionals and academic researchers alike. Thus, packers can be seen as a barrier, which has to be overcome in order to perform effective static analysis. This indicates that having a maintained reference corpus of carefully compiled data, focusing on providing unpacked samples would have a huge benefit for research on malware, as it could potentially fill the gap and provide researchers with relevant and understandable data to base their work on.

On the other hand, when looking at practical malware research as for example conducted and published by threat protection vendors or CERTs, their work mostly focuses on single malware families or sets of families as used by specific threat actors. But in consequence, the resulting research and design of tooling currently carried out is of reactive fashion, based on observations of prevalence in the wild.

In summary, the core problems of current malware research are uncertainty in representativeness and expressiveness as well as the reactive short-sightedness they exhibit. This clearly motivates the need for an extensive, realistic, representative corpus of malware. Such a corpus could be used to find answers to questions beyond chasing the current trends that has been shaping the design and optimization of analysis methods for many years now. High value use cases are for example the ability to create context by inferring previously unknown code relationships between malware families or the derivation of generically applicable structural and behavorial features useful for signature generation, i.e. detection and identification.

This leads to the central research question for this chapter:

> RQ_1: How should a malware corpus be composed in order to enable academic researchers to conduct representative malware research while simultanously serving as a relevant resource for practical malware analysts?

In an attempt to answer this question, we have created Malpedia. Malpedia is a reference corpus for malware research, especially optimized for static analysis and providing wide coverage of distinct malware families while being exceptionally accurately labeled. On January 3rd, 2019, Malpedia features 3.469 reference samples for 1.136 malware families of multiple platforms and additionally 2.447 references to analysis reporting, which makes it the most comprehensive malware corpus for practically-orientied malware research at the time of writing. As one central goal is to provide verified unpacked

payloads, it allows a unique insight beyond the shroud formed by polymorphic packers and allows to study the malicious payload programs as produced by their authors.

Having this data at hand, we formulate a follow-up question:

> RQ_2: To which degree is the integrity of original payload meta data and file structure maintained, and based on this data, what can be inferred about tool chains and methodologies as used by the malware authors?

Using our framework `malpedia-analytics`, we show that the integrity of the vast majority of payloads is not affected by packers used. This means that many samples contain seemingly generally valid meta data, among them such interesting fields as the compilation timestamp, or linker version. For example, inspecting the latter, we learn that the most common tool chain is Microsoft Visual Studio in its rather outdated versions 2010 and 6 (dated 1998), showing that malware authors seemingly stick to development environments they are most used to. Furthermore, malware authors spend little attention to data fragments that tell about their development systems, as unredacted PDB paths with user names stored in paths and plausible Rich Header entries show.

Contributions. By answering research questions RQ_1 and RQ_2, we lay the foundations for a realistic, representative corpus useful for malware research. In summary, in this chapter we make the following contributions:

1. We define a set of requirements for a representative malware corpus focused primarily on static analysis and show their harmony with documented best practices.
2. We detail our approach and experiences with composing a reference data set following these requirements, showing that well chosen data can be representative for file collections several orders of magnitude larger.
3. We provide Malpedia as a reference corpus following these requirements to the research community.
4. We demonstrate the applicability and usefulness of the corpus by conducting an extensive study of meta data availability and payload integrity for the 839 Windows unpacked malware families.

4.2. Requirements for a Malware Corpus focused on Static Analysis

After the introductory motivation of the need for a realistic malware corpus, we start by defining requirements that such a corpus should obey. In order to derive concrete requirements that a malware corpus focused on static analysis should fulfil, we first look more closely at what this data set shall enable, especially in the context of this thesis. For this, we quickly recapitulate that the two key challenges identified initially in Chapter 1 were centered around ground truth and situational awareness and derive the following three goals:

1. to create a comprehensive data set to enable a comparative analysis of Windows malware,
2. to derive knowledge about the applicability of static analysis techniques and
3. to apply and improve techniques to investigate relationships between malware families.

We identify requirements that are relevant in all of these goals and which can be categorized into 3 overarching topics: representativeness, accessibility, and practicality. We will first explain the aspects covered by these categories in Section 4.2.1. Afterwards, we examine their relationship to the seminal work on this topic, the "Prudent Practices for Designing Malware Experiments" as defined by Rossow et al. [50], showing that our requirements sufficiently cover the guidelines proposed.

4.2.1. Definition of Requirements

In this section, we define our three requirements for a realistic and usable malware corpus:

- REQ_R: Representativeness
- REQ_A: Accessibility
- REQ_P: Practicality

In the following, we discuss the relevant aspects covered by these requirements in detail.

REQ_R: Representativeness

First and foremost, any data set should be representative so that any experiments and analyses conducted yield expressive and meaningful results. This means that it has to achieve coverage in at least the following (potentially interdependent) dimensions.

Temporal Coverage. To ensure that both general trends in malware can be identified but also the development of code from families over time can be studied in detail, the corpus should provide sufficient temporal coverage. As mentioned in the introduction of Chapter 4, malware has been commoditized since around 2006, so this could serve as a meaningful lower temporal border which the collection should aim for.

Malware Family Diversity. Furthermore, as implied by temporal coverage, the corpus should contain a large number of distinct malware families to support comprehensiveness. However, given the sheer amount of unique observable samples, means of limitation have to be installed. The probably most important feature from a practical point of view that can be used to filter malware is its relevance. A critical problem is that relevance is not defined in a standard way or easily measured objectively. Nevertheless, it can be inferred to a certain degree by studying secondary sources such as analysis reports on malware as published by Antivirus and Threat Intelligence companies. As these companies are driven economically, it can be assumed that their focus is oriented towards what they

perceive as the most threatening malware families or focusing on novel, undocumented specimen. This means that relevance of malware families is indirectly expressed through features such as volume and operative reach, persistence over time, or usage in attacks against high value targets.

Version Coverage. Another dimension is situated within families themselves. It can be assumed that malware authors work similar to the authors of regular (benign) software, i.e. they gradually change their code over time. Example reasons for such changes can be to evade detection, improve or extend the functionality offered by their malware, or to simply fix bugs that impaired their operationality. In consequence, malware families will typically have a series of versions over time that provide a picture of the evolution of their code base.

Platform Diversity. While this dissertation primarily focuses on Windows malware, it should be noted that malware has become increasingly popular for other platforms, e.g. mobile and IoT devices [67, 230]. Therefore, a malware corpus should also consider inventorizing malware for other platforms to reflect the entire spectrum.

REQ_A: Accessibility

Besides representativeness, a second requirement for the corpus should be to ensure accessibility. As shown in Chapter 3, the vast majority of previous works have used collections of malware samples for which the composition could only be estimated. We believe that this is strongly tied to a lack of accessibility as found for many malware collections.

Labeling. While it is not hard to acquire tremendous amounts of malware samples, a dissimilarly harder task is to obtain reliable information about their identity. The primary reason for this is that for defending against malware, detection massively outweighs identification, which is for example observable in detection labels as assigned by Antivirus software [84].

Unpacked Representation. To overcome these issues, we believe that unpacked, clean counterparts for malware samples have to be a core component of the corpus. Identification for these is way easier than for unprocessed malware samples, which allows to assign and verify reliable labels with high confidence. As a result, it becomes possible to perform experiments using static analysis on the actual malware families without being obstructed by packers.

Usability. Additionally, the data set should be structurally organized and made available in a way that it can be easily used by other researchers.

REQ_P: Practicality

On top of representativeness and accessibility, a third requirement should be to aim for practicality, which covers four aspects.

Topicality. In order to remain relevant in a dynamic field such as malware research, a data set should ideally be kept up to date. Ensuring maintenance and keeping the data set topical provides already a great added value and allows it to serve purposes even well beyond academic applications.

Referenceability. In the case of an evolving data set that aims to provide topicality, it is important to provide temporal reference points, to ensure that experiments can be replicated as well as reproduced. The data set should therefore provide mechanisms that allow to create permanent references of its state at any given time.

Documentation. Next, it is important to take care of the documentation of processes employed to create the data set. For example, it is necessary to record the origin of malware samples and to specify any environments and processes used during data creation.

Containment. Finally, as the data set contains purposely harmful software that can lead up to the unrecoverable destruction of data, containment has to be ensured. For this reason, dissemination should be limited to parties that are experienced in handling such data with the necessary amount of care. Additionally, as the contents of the data set may tip off malware authors, even access to meta data such as file hashes should be restricted.

In the next section, we continue by reviewing the Prudent Practices as defined by Rossow et al. [50] and compare them against our requirements.

4.2.2. Review of Rossow's Prudent Practices

In 2012, Rossow et al. [50] published a survey that scrutinized 36 earlier papers in the field of malware research from the years of 2006-2011. They found a number of systematic shortcomings that seriously flawed the experimental results, most of them tied to the handling of malware data sets. Even after the publication of their paper, these flaws are observed in a range of papers (cf. Chapter 3) and it is apparent that quality data sets are not used by or available to academic researchers. In our process of creating a high-quality, accurately labeled malware corpus focusing on static analysis, we use this section to review the guidelines published by Rossow et al. [50] for suitability to our cause and implement them where it appears reasonable.

The guidelines postulated in "Prudent Practices" are generic rules applicable to any kind of experiments involving malware. They are divided into 18 aspects that are grouped into the four categories of "Correctness of Data Sets", "Transparency", "Realism", and "Safety". These categories are not directly applicable to our case that only encapsulates data set design and organization, which is rather a sub-category for which aspects are found in the four categories defined by Rossow et al. In the following, we briefly summarize all of these aspects and then revise their relevance and applicability for our case.

Correctness of Data Sets

For Rossow et al., the topic of correctness questions the composition of a corpus with regard to what shall be shown by an experiment. It is used as well to ensure that the analysis produces meaningful and expressive results.

1. **C1 Inclusion of Goodware:** Especially for techniques that decide for a given software if they are benign or potentially malicious, goodware should be included in the data set.
2. **C2 Balance of Families:** A corpus should be balanced in order to show that a given technique performs well across families and to avoid the impression it is tailored to special cases.
3. **C3 Choice of Training Data:** If it is intended to show that an approach is able to detect unseen malware, it is recommendable to use distinct families in training and in evaluation data.
4. **C4 Privileged Analysis:** The analysis should always be carried out with higher privileges than the malware to avoid tampering.
5. **C5 Artifacts:** Side effects and biases should be mitigated as well as possible.
6. **C6 Blending:** When blending malware into benign background data, this should be carried out with high caution to ensure realism but also produce meaningful results. Especially, it should be ensured that the background data is indeed benign.

Out of these 6 aspects, the aspects of Balance of Families (C2), Privileged Analysis (C4), and Artifacts (C5) are of high relevance. Aiming for balance can be translated into striving for a high coverage in variety of families and below that, versions of these (REQ_R, REQ_P). Privileged analysis should be employed in order to maximize the success for unpacking (REQ_A, REQ_P). Artifacts that may result from the unpacking process on the other hand should be avoided in order to ensure purity in the results. But these artifacts still need documentation (REQ_R, REQ_A, REQ_P).

As goodware is not of concern for a malware corpus, the aspects Inclusion of Goodware (C1) and Blending (C6) can be neglected. Furthermore, since no classification performance is measured, the choice of training data (C3) can be neglected as well.

Transparency

Experiments in research should be replicable which is also covered by by our aspect of practicality (REQ_P). In this vein, Rossow et al. document that it is paramount to describe the experiments in such depth that design ideas and interpretation of results are transparent.

1. **T1 Malware Family Names:** Consistent naming of malware is a persistent problem that regularly causes confusion. Therefore, the choice of naming should be explained in detail, or citation should be given to allow cross-referencing.

2. **T2 Time-dependency:** The time when an analysis was conducted may change the outcome, e.g. when date-based packers are involved or when availability of network connectivity and C&C components plays a role.

3. **T3 Sample Selection Criteria:** The selection process of samples is very impactful as it determines how the resulting corpus is comprised. It also influences many other aspects as outlined here.

4. **T4 Analysis Setup:** The setup of an analysis machine can greatly affect dynamic analysis, as some malware may have dependencies or do not run on certain operating system versions at all. It is also of importance when stating benchmarks for processing speed.

5. **T5 Network Connectivity:** For any experiment, it should be clearly stated if and how a system was connected, i.e. it should also be differed between public facing IPs and NAT.

6. **T6 Thorough Analysis of Results:** When interpreting the outcome of a detection methodology with regard to true/false positives and negatives, reason and diversity of results should be scrutinized.

The aspects on Malware Family Names (T1) and Sample Selection Criteria (T3) are core concerns when compiling a malware corpus, as they massively influence its organization and aspired coverage (REQ_R, REQ_A, REQ_P). While not doing experiments themselves, the Analysis Setup (T4) used for unpacking should be well documented, as this information may later be exploited to augment the results obtainable through static analysis (REQ_A, REQ_P). Since the orientation of our corpus is to support static analysis, Time-dependency (T2) influences how topical the data is and may be relevant when facing date-based packers (REQ_P).

Network Connectivity (T5) should not be enabled because treating samples in isolation benefits their identification. The aspect with regard to analysis (T6) is again not applicable because the corpus is a base for further analyses and not an analysis in itself.

Realism

Rossow et al. also require to show that the implications of the approach impact the real world, and thus the experiment should adhere to realistic conditions.

1. **R1 Relevance of Families:** To support the usefulness of an approach, it should be evaluated against recent and relevant malware families. In consequence, stale malware samples should be avoided.

2. **R2 Significance:** It should be shown that the approach is scalable, e.g. for a network-based approach, a significant number of hosts should be used.

3. **R3 Generalization:** It should be avoided to generalize results when they originate from a single operating system.

4. **R4 Stimuli:** When conducting dynamic analysis, the malware should be appropriately stimulated to near behavior on a real-world system.

5. **R5 Internet Access:** To allow for a realistic environment, the analysis system should be connected to the Internet.

Relevance (R1) of the content is a core facet for a malware corpus and thus very important (REQ_R, REQ_P). The selection of samples should be both broad in coverage while keeping its focus on active and impactful malware families. As a detail, for a malware corpus that aims to preserve and document evolution of malware as a whole (REQ_R), the exclusion of "stale" samples would actually be harmful and is thus not conducted.

Stimuli (R4) can be adapted again in the context of unpacking, as every method available should be considered to excavate the payloads from the packed samples and make them accessible to in-depth analyses (REQ_A, REQ_P).

All the other aspects refer to experiments and can again be discarded.

Safety

While realism is a high goal, Rossow et al. also make clear that there needs to be balance against safety, especially when allowing full network access.

1. **S1 Containment:** Experiments have to avoid at all cost any harm to uninvolved parties. Thus, proper containment has to be installed when running malware.

The aspect of Containment (S1) again addresses provisions to be taken when doing experiments based on dynamic analysis, which is therefore not directly applicable to the cause of creating a malware corpus oriented towards static analysis. However, containment should still be considered in another context: Distribution of the data (REQ_A, REQ_P). While there is a trend towards open and easily reproducible research, we believe that the data set created here should not be openly published, as it poses danger to uninformed users that are not experienced in handling malicious software (REQ_A, REQ_P).

4.2.3. Summary and Mapping to Prudent Practices

Table 4.1 shows how our requirements defined in Section 4.2.1 correspond to Rossow's Prudent Practices.

To begin with, it should be noted that four out of 18 aspects were found not applicable in the context of creating a malware corpus as they address specificities of conducting sound experiments involving malware.

Next, three of the aspects are influential on all of our defined requirements. They address cleanliness of data and labels themselves as well as the procedure of selecting it, which is reasonable. This also highlights their outstanding importance.

Another six aspects relate to more than one of our requirements. They postulate goals to be pursued in the data creation and refinement process as well as how to ensure relevance for the data set and storing it safely.

Finally, one aspect influences practicality of the data and is connected to consider time-dependency, which we interpret as topicality.

In summary, it can be concluded that Rossow's guidelines are still a valid and a helpful orientation even if their context of application is slightly adjusted to in our case. They definitely confirm our own perspective on this matter, as shown by our definition of requirements for a malware corpus focusing on static analysis.

Aspect	**Representativeness**	**Accessibility**	**Practicality**
C1 Inclusion of Goodware	-	-	-
C2 Balance of Families	✓	-	✓
C3 Choice of Training Data	-	-	-
C4 Privileged Analysis	-	✓	✓
C5 Artifacts	✓	✓	✓
C6 Blending	-	-	-
T1 Malware Family Names	✓	✓	✓
T2 Time-dependency	-	-	✓
T3 Sample Selection Criteria	✓	✓	✓
T4 Analysis Setup	-	✓	✓
T5 Network Connectivity	-	-	-
T6 Analysis of Results	-	-	-
R1 Relevance of Families	✓	-	✓
R2 Significance	-	-	-
R3 Generalization	-	-	-
R4 Stimuli	-	✓	✓
R5 Internet Access	-	-	-
S1 Containment	-	✓	✓

Table 4.1.: Mapping of Rossow's aspects to our categories Representativeness, Accessibility, and Practicality.

4.3. The Malpedia Corpus

In this section, we introduce our implementation of a malware data set according to the requirements defined in Section 4.2. The Malpedia corpus is intended to remain an ongoing project that has been pursued since March 2016 and shall yield the most comprehensive corpus of unpacked malware available.

We will first present how the data contained in the corpus is organized. Next, we explain in detail which environment and procedures are used to produce unpacked representations of malware samples, which are the essential and unique feature of this corpus.

Afterwards, we explain the method for choosing samples to be integrated in the data set. Finally, we outline the contents of the corpus at the time of writing, which form the basis for the analyses conducted in the following Section 4.4. To demonstrate that our implementation is in line with the requirements, we mention the respective requirements fulfilled by our design decisions throughout this section. Where possible, we resort to using well-proven industry standards.

4.3.1. Storage and Organization

When considering the conceptual options for designing storage and data organization, we first need to acknowledge the respective aspects of our requirement categories REQ_R, REQ_A, REQ_P that may be relevant in this context. While most of these aspects address content-related issues, Usability (REQ_A) and Referenceability (REQ_P) should be immediately accounted for when thinking about storage and organization. Another aspect that may be relevant is Labeling (REQ_A), as synonyms are known to be common for malware families and have to be potentially treated as well.

As a methodical anchor, we use the definition of a malware family as given in Section 2.2 as our foundation to organize all malware samples to be contained in the Malpedia corpus. Meaning that a single sample will correspond to one family only, this allows us to employ a hierarchical organization of data with the primary criterium being families. Additionally, as one malware sample is only capable of reflecting a single state of development, families can be further divided into versions. In consequence, this observation leads us to the adoption of a straightforward hierarchical mapping, and thus a structure resembling a file-system, using folders for organization. A folder structure is a human-understandable and highly usable format of organization that turned out favorable over a tag-based system, e.g. when sorting emails or files as shown by Bergman et al. [231]. It is also useful to provide convenient and direct access to the contents of the data set. This allows the application of analysis tools directly to the files and having their association reflected in the path.

One commonly raised drawback of a hierarchical system for data organization is the lack of flexibility in data presentation. In order to give this additional flexibility, we propose a method of storing meta data for the family along the malware samples. This meta data specifically allows potential re-organization of the presentation layer at a later time by abstracting from the file system. For the implementation, we use a single, dedicated meta data file per family, and as format a common data serialization format to ensure compatibility. By storing the meta data within the folder of the family, we ensure that the meta data remains available when a family folder is isolated. As serialization format, we chose JSON [232] over other options such as XML [233], as it is just as suited for our intended purpose, while being simple and very compatible with data structures such as dictionaries as offered naturally by modern programming languages.

To ensure consistency, we impose a schema on the meta data files. While the meta data storage is designed to be extensible, to again aid usability we provide facilities to capture the following information by default: a primary family name and aliases, public

analysis references (blogs, reports) as well as information about known threat actors, whenever available.

Before we further detail how families and samples are stored precisely, we first want to address the requirement of Referenceability (REQ_P). As we chose a file-system as basic structure, an overlay such as a version control system (VCS) is a natural choice for tracking changes. Using a VCS, we can ensure that every change is tracked atomically and automatically becomes a referencable state, as commit entries are typically identified by author, timestamp, and a hash serving as unique identifier. Among potential version control systems, we choose Git as it is by far the most popular option, e.g. shown in a 2018 survey conducted among more than 100,000 users by Stackoverflow [234]. As most implementations of version control systems natively support access restriction, we can simultanously address the requirement of Containment (REQ_P).

Considering different options for the requirement Labeling (REQ_A) (and thus primary name and aliases) of malware families, one outstanding option is an industry-wide adopted concept called the CARO naming scheme [235, 236]. As this scheme by design encapsulates both platform and a family identifier, it is perfectly compatible with our intended use of a hierarchical structure. For the identifier of a family, again with usability in mind, we resort to a name as used by the majority of references and AV detections found for the respective family. Because this method of naming is subjective and may even change over time, we use the introduced meta data storage to aditionally assign a universally unique identifier (UUID) [237] when first introducing the family to the corpus.

As outlined, for a malware family, versions are identified and organized as folders below the family level. To aid the hierarchical structure and Usability (REQ_A), we use internal versioning as well as temporal information, if available. Whenever possible, the malware's own specific version information is used (if known), as this provides a natural criterium to characterize its line of development from its author's perspective. Temporal information may additionally be available in file meta data or through external services with exceptional visibility, such as VirusTotal [238]. In case neither information is available, the sample is left in the root folder of the family with the option to be versioned at a later point in time.

For each version of a malware family, only a single representative is kept. This is easily explained with the corpus focusing on malware payloads instead of packers, thus avoiding redudancy. This design choice furthermore results in a drastic data reduction because instead of keeping numerous packed samples containing an identically versioned payload, storage is now reduced to a single representative file. This is further motivated in Section 4.3.3.

In case that the malware family is modular in design, the malware-family specific modules are stored in a dedicated subfolder named "modules". This allows us to separate the core payload from auxiliary files in a meaningful way that reflects the author's design choices for their malware. It additionally accounts particularly for the special circumstance that these modules are typically not observed being distributed through those channels used for initial infection (spam, exploit kits) but delivered through C&C channels. Modules again are versioned in an identical way as explained before.

The samples themselves are stored as found in the wild but renamed to their SHA256 hash, capturing the uniqueness of the data item identified by its content. We decided to use this hashing algorithm as it is not shown to be prone to collisions yet, opposite to MD5 [239] and SHA1 [240]. All processed data related to a sample is stored along them with similar naming. In these cases, additional identifiers are appended to the origin SHA256, divided by underscores. Permissible suffixes are:

- `unpacked:` The payload extracted from the original sample, preferably in its original representation, i.e. not memory-mapped or reconstructed and thus executable.
- `dump:` A memory-mapped representation of the unpacked payload. In this case it is mandatory to record the environment and base address the dump was obtained from in hex representation.

Tracking and pinning the environment provides a beneficial reference for additional context that could be relevant during later analysis. Due to system snapshot consistency, it allows for example to apply memory deduplication techniques [241]. Recording the base address has two benefits: First, it is an important detail for later analysis as well and second, it can be used to immediately indicate the bitness of the hosting process the dump was taken from by formatting the respective address width for 32 and 64bit.

```
[…]
├──── <platform>.<family_name>
├──── win.urlzone
│     ├── 2014-11-08
│     │   ├── 62a19def1dbca132c4e1d53848356be78df6a1f80947ecb0ed7f76f85a94514f
│     │   ├── 62a19def1dbca132c4e1d53848356be78df6a1f80947ecb0ed7f76f85a94514f_dump_0x01e00000
│     │   └── 62a19def1dbca132c4e1d53848356be78df6a1f80947ecb0ed7f76f85a94514f_unpacked
│     └── 2015-04-29
│         ├── 0e7a9a2df9a4db4c537f248ce239aba17bfa3618afcfc30de5d2a460b80b2b55
│         ├── 0e7a9a2df9a4db4c537f248ce239aba17bfa3618afcfc30de5d2a460b80b2b55_dump_0x01e00000
│         └── 0e7a9a2df9a4db4c537f248ce239aba17bfa3618afcfc30de5d2a460b80b2b55_unpacked
├──── win.urlzone.json
├──── yara
│     ├── tlp_green
│     │   └── win.urlzone_g0.yar
│     └── tlp_white
│         ├── win.urlzone_w0.yar
│         └── win.urlzone_w1.yar
[…]
```

Figure 4.1.: Malpedia data set structure as introduced in [12]

An example for this structure is shown in Figure 4.1.

For both `unpacked` and `dump` data, multiple files may be the result. For example, in case the malware itself is exclusively found in a multi-staged form, e.g. realizing execution chains such as

- `dropper` → `loader` → `core_payload` or
- `loader` → `core_payload` → `modules`

It should be noted that it is not always possible to extract a meaningful `unpacked` representation for packed samples. Especially when the payload code is not a proper binary found in PE file format but shellcode instead, it will be often directly mapped to memory and then executed. Consequently and because of the fact that all code has to

be executed at some point, we decide to generally prefer `dump` representations and even consider them as an appropriate normalization format.

The method for creating dumps is explained in detail in the following section.

4.3.2. Environment Specification and Dumping Procedure

In order to be compliant with the requirement of Unpacked Representation (REQ_A), all samples are transformed into a normalized form reflecting this state. As explained in Section 4.3.1, the use of memory dumps is the preferable representation choice in the Malpedia corpus. To further adhere to the criterions of Documentation (REQ_P), this section presents the environments and procedures used to create these dumps.

Environment Specification

All memory dumps should originate from a limited set of well-defined environments. In order to ensure that all memory dumps are created from an identical runtime state of the operating system, the use of a virtualization software makes sense, as these allow to record and restore runtime states. We decide to use VirtualBox because it is open-source, thus offering good potential for hardening against anti-analysis and is also well studied for the application of malware analysis [41]. Using an identical runtime state is necessary to ensure the best possible comparability across dumps.

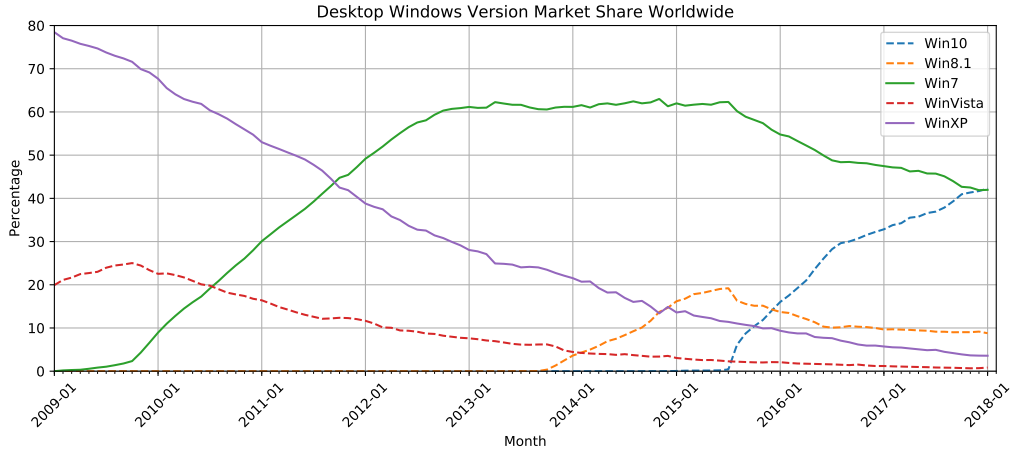


Figure 4.2.: Windows version market share, based on data by StatCounter [242].

To maximize compatibility with the malware, we opt to use the operating system versions that have been most common in the time period to be covered by the corpus. We choose Windows XP and Windows 7 and created hardened Virtual Machines (VM) for them. While Windows XP was first released in 2001 and may appear to be an outdated choice, it has been the most popular operating system until mid of 2011 as shown in Figure 4.2. Due to its popularity, we can safely assume that it was the most targeted and optimized-for operating system for Windows malware authors, which makes

it a natural candidate as a base system. Windows 7 on the other hand has been the most popular Windows version ever since it took over the lead from Windows XP. An argument against using newer versions may be that they introduce additional mandatory security mechanisms which may obstruct the execution of older malware, such as enforced ASLR and DEP. Finally, using just these two environments also allows to cover both 32bit and 64bit programs.

Apart from using these Windows versions as a baseline, additional software should be installed to increase execution and thus unpacking success. Microsoft Visual Studio, the most popular compiler and IDE for C/C++ on Windows [243], will typically compile programs in a way to make use of dynamic loading. This may result in dependencies on the Microsoft Visual C Runtime (short: MSVCRT) which typically has to be installed by users. We therefore collect and install all versions of MSVCRT available for both Windows XP and 7. The same is true for the .NET framework, where we install all available versions as well.

A summary of all relevant configuration details is given in Table 4.2.

	Windows XP	**Windows 7**
CPUs	1	2
RAM	2 GB	4 GB
HDD	120GB	120GB
Bitness	32bit	64bit
Service Pack	SP3	SP1
Version	2600.080413	7601.101119
MSVCRT	up to 14.0	up to 14.1, 32bit and 64bit
.NET	up to 4.0	up to 4.7

Table 4.2.: Specification for the reference virtual machine images.

In order to further increase the unpacking success rate, the environment is additionally hardened against detection by potential anti-analysis methods used by the malware or packers [244]. First, the virtualization software is hidden as good as possible. For this, we exchanged the BIOS emulated by VirtualBox with the parameters of a real system and changed the network interface's MAC address to a plausible neutral vendor prefix. Furthermore, we set the paravirtualization settings to "None", resulting in no detectable hypervisor bit in the CPU and also no hypervisor ID. We also forego the installation of VirtualBox's helper utilities. Second, some emulated "wear" is inflicted onto the system. For this we performed typical system usage activities such as starting and stopping programs, creating and deleting files in various folders.

Dumping Procedure

The highest priority goal for dumping is to extract an unpacked version of the malware, preferably in its naturally encountered state, i.e. a mapped memory image. This makes

the data specifically useful for the derivation of detection mechanisms as it represents the running state in which the malware would be encountered during incident reponse and memory forensics. It has been shown that memory dumps of payload malware can be obtained using automation with a success rate of up to 90% as described and analyzed by Jenke et al. [245]. We therefore apply the following approach, driven by the idea of gradually increasing the effort or degree of intervention necessary to obtain an unpacked version. It is organized in four stages, out of which stages 1 and 2 are automated as explained in [245], while stage 3 and 4 resemble a typical manual analysis workflow [38]:

1. Stage 1 is a snapshotting-based method, for which the operating system's memory state is recorded and compared prior and after the execution of the malware sample. We first execute the sample for 2 minutes, as proposed by Jenke et al. [245]. After taking the second snapshot, we perform a comparison of memory, through which we can identify all changes of executable memory. These candidates are then filtered by removing known benign files originating from the system and are afterwards examined using YARA signatures or manual inspection in order to identify unpacked memory representations of the target malware.
2. Should stage 1 not lead to success, we repeat the method but additionally use system-wide hooking in order to prohibit termination of any processes, as proposed in [245]. This yields additional coverage of samples where the packer or family will terminate their process. Observed reasons for this are e.g. detection of an analysis environment (despite our hardening efforts), missing preconditions (Internet connection or presence of certain system aspects), or completion of the intended functionality (ransomware).
3. In case stages 1 and 2 fail to produce a satisfying result, a manual investigation of the execution behavior is conducted. Now all memory allocations and deallocations are monitored and inspected during the execution phase. This helps in cases where the packer or malware take precautions by minimizing their traces in memory, e.g. by actively removing themselves.
4. As a last resort if none of the three preceding stages lead to a result, it is likely the payload of interest never reached an unpacked and thus dumpable state during the execution. In this case, an in-depth analysis has to be performed. This involves potentially a high degree of intervention, up to debugging and patching code in order to extract an unpacked representation of the payload code.

Without recording exact numbers, it should be noted that the application of stages 1 and 2 leads to success for up to 90% of samples as noted in [245] and manual runtime analysis is only performed if ultimately necessary. As a special case, DLLs should be mentioned. Opposite to executable files, in this case both the DllMain function as well as all exported functions may be the potentially relevant program entry points, which may incur additional runs to produce a dump as observed by Jenke et al. [245].

The dumping in stages 1 and 2 is generally done using low-level mechanisms offered by the Windows API, whereas dumping in stages 3 and 4 is done using a kernel-mode component (offered by the analysis tool ProcessHacker [246]) or the respective debugger used.

We have observed that the outlined methodology may lead to inaccurate representations of the memory layout as used by .NET executables. Here, sections normally will have an alignment of `0x2000` bytes. Because this is not reflected by the tools mentioned above (or other tools tried before), we have addressed this fact by inserting zeroed memory pages into the respective missing gaps if not found in the memory dumps.

As a final processing step, we additionally improve the quality of the created dumps in two steps. First, we clean fragments of packer code trailing the payload, in case they are identified as obvious third party code not belonging to the malware family itself. Second, in case the dump of the malware payload contains a PE header, we check this header for validity and delete potentially trailing zero bytes, which may have been created by a packer as an obvious analysis countermeasure: artificially increasing memory space required to store analysis results and memory dumps. All of this again has the purpose of producing data that provides highest standard of Usability (REQ_A).

4.3.3. Achieving Representativeness

After having explained how the data set is organized and how the unpacked representations of malware samples are produced from a technical perspective, this section covers all aspects dealing with the requirement category of representativeness (REQ_R) while also providing Topicality (REQ_P).

As explained in Section 4.2, a comprehensive malware corpus should aim to provide extensive Temporal Coverage (REQ_R) to capture generals trends within the field of malware, and to offer Malware Family Diversity (REQ_R) as well as Version Coverage (REQ_R) to reflect evolution of families and their intrinsics, which implies also supporting Platform Diversity (REQ_R).

As primary data source for malware samples themselves, we decided to use VirusTotal (VT), a service that has been collecting and archiving scan results of malware since 2004, thus giving sufficient Temporal Coverage (REQ_R). VirusTotal is a popular service and allows free public uploads of suspicious files, resulting in an immense coverage. It has furthermore the advantage of not being an Antivirus vendor by itself, thus being unbiased. As a secondary resource, we have also been granted access to the (non-public) malware archive of the Shadowserver Foundation [247], which has been collecting malware since 2004 as well.

We will now detail the strategy we followed in order to work towards these goals. Our collection approach can be divided into three categories.

Previously collected data. An initial collection of malware samples for a range of well-known and prevalent families was taken from our internal unpacking and malware configuration extraction framework, called Kahou. It is capable of identifying 37 distinct malware families and applying methods to extract their respective configuration parameters. In the period of 2013 to 2016, Kahou has been used to process more than 600,000 unique malware samples that have been kindly provided by the Shadowserver Foundation [247]. We performed a deduplication based on the version numbers and PE compilation timestamps of payloads. This data was then augmented by reviewing the

sharable content of our internal incident and analysis case repository. This category of existing data resulted in the addition of around 120 malware families to the corpus.

Third-party sources. In order to get information about samples associated with certain malware families, we heavily rely on the numerous publications released by the highly active Antivirus and Threat-Intelligence industry [248] as well as private and professional blogs maintained by independent researchers. Most of these entities frequently publish detailed reports in which they describe their obsverations of tracking malicious software and actors, typically accompanied with hashes of reference samples. As these publications remain available in various forms, including whitepapers, blog posts, IOC collections, and information repositories, this allows us to reconstruct the history of malware research to a certain degree. By meticulously dissecting these combined data sources, we were able to add another 746 malware families to the corpus. This data category significantly contributed to Malware Family Diversity and Version Coverage (REQ_R), as most publications discuss the continuous developments of the malware families covered.

Community contributions. In addition to our own collection efforts, we have launched a community-driven web service around Malpedia in December 2017 where vetted members of the malware analysis community can propose additions to the corpus. Since its start of operation and until January 2019, community members have added more than 3,000 content proposals. While the majority of these proposals is centered around extensions to the meta data (references to publications on families), this has resulted in the addition of another 270 malware families to Malpedia as well.

All of these three categories have made significant contributions to the corpus. We have taken a snapshot of the corpus on January 3rd, 2019 that is used throughout this thesis. It contains 1,136 distinct malware families with a total of 3,469 samples, further described in Section 4.3.4. To fulfil the requirement of Topicality (REQ_P), we continue to monitor third-party sources and integrate their referenced malware families and versions and also maintain the community founded for the curation of the Malpedia corpus.

Core Findings

We now summarize observations and experiences we have made during this process.

Data redundancy. A primary observation from our previously collected data processed with Kahou is that for a range of families, a massive data reduction can be achieved when comparing the amount of encountered packed samples versus the uniquely identified versions. For example, we used Kahou to intensely track the builder-based Zeus offspring `win.citadel`. Over the period of three years, the Shadowserver Foundation has provided us with more than 80,000 samples pre-identified as `win.citadel`. Examining the output of our configuration extractor, we identified more than 140 identifiers corresponding to individual builder kits. But even these group into just 21 distinct versions with regard to code of the malware, a finding supported by cursory code similarity analysis. In

consequence, this implies just based on the data processed by us, an observed data-reduction factor of 3,800x for the representation of this malware family is achieved by keeping only one sample per version. Other families where we observed similar ratios are `win.tinba` with a reduction factor of 5,700x, the spam malware `win.asprox` with a factor of 5,500x, and the other two Zeus offprings `win.vmzeus` and `win.kins` with factors of 471x and 105x respectively [12]. Similarly, Haq et al. recently reported an average reduction factor of 26x with regard to unique samples and automatically derived versions for 7,793 samples of 10 families in their malware lineage study [81]. Because there can only be so many malware authors who manually have to create and modify code, we expect similar ratios for a significant portion of malware families, especially commodity malware. This is a very important finding because it suggests that when focusing on unpacked malware, huge numbers of samples can be accurately represented by a fraction of well-chosen samples.

Effectively sourcing malware families. Another observation is that scaled mass data processing as a technique to produce labels of the required accuracy is not self-sustainable. In fact, the creation and maintenance of YARA rules and extractors is only possible by dedicating significant resources in terms of qualified staff to fulfil these tasks. With this realization in mind we shifted our strategy of data collection and primarily relied on third party information. Instead of mass processing samples, we opted to use opportunistic cherry-picking of pre-labeled data from trustworthy sources. One helpful observation in this context is that there is often a redundancy of coverage observable across multiple commercial vendors. This is connected to trends that relate to current and high-profile incidents. As a result the number of samples to be considered for integration in the corpus is typically low enough to maintain the outlined strategy even with low resources. For perspective, in combination with the review and refinement of content proposals by the Malpedia community, this cumulates to 8-10 hours of effort per week at the time of writing.

Collection Bias. While we attempt to have a balanced coverage through our wide choice of third-party sources, it does not go unnoticed that these sources are subjected to following trends. This is expressed in one vendor covering a certain actor or family that is then reactively covered by other vendors as well. Furthermore, the sources monitored are mostly in English language and also Western-focused. We have to assume that this results in a potential underrepresentation of malware families prevalent in regions such as Asia, South America, and Africa.

Impure malware and cross-infections. It also should be mentioned that assigning single labels to packed samples is not always possible. For example, we had frequent encounters where a malware sample of one family would additionally contain one of the notorious families behaving as file infectors, such as `win.virut` and `win.sality`. Another case are orchestrating packers that are capable of dropping multiple payloads. A common combination that we have observed more than once is malware with information stealing capabilities (such as `win.pony`) being run in conjunction with other malware families. This has implications to the design of analysis systems but also the compilation of malware corpora. In particular, a Host Intrusion Preventions System (e.g. Antivirus software) typically aims for detection and will usually produce only a single detection

label, potentially missing presence of another malware family. For Malpedia, we avoid the integration of any samples that may lead to such ambiguities. That said, we do not include packed samples that result in more than one unpacked family when executed if there are alternative samples that do not have this trait.

4.3.4. Data Set Status

In this section, we give a brief overview of the data set status of the Malpedia corpus that will serve as a basis for all following experiments throughout this thesis. As a reference point, we use the Git commit `1639cad`, which has been created on January 3rd, 2019.

	Families				Samples			
Platform	DMP	UNP	PACK	Total	DMP	UNP	PACK	Total
Android	0	1	70	71	0	1	148	149
ELF/Linux	0	11	37	48	0	47	144	191
iOS	0	1	2	3	0	5	2	7
macOS	0	23	21	44	0	52	55	107
Windows	839	41	49	929	2,352	295	308	2,907
other/multi	0	23	18	41	0	39	69	108
Sum	839	100	197	1,136	2,352	439	726	3,469

Table 4.3.: Overview of the corpus state of Malpedia at the snapshot date January 3rd, 2019, commit `1639cad` .

Table 4.3 gives a comprehensive overview of the number of families and samples per platform as well as their processing states. For every sample, only the highest state present is counted, considering dumped, unpacked, and packed (cf. Section 4.3.1). With regard to the two specified environments used for dumping on Windows (cf. Section 4.3.2), 937 of 2,352 (40.67%) dumps have been produced using Windows 7. The minimum file size for a dump in this data set is 4,096 bytes and the maximum 15,687,680 bytes, with a median of 135,168 bytes.

In order to illustrate that the corpus does indeed represent recent malware, we have performed a lookup for all sample file hashes against VirusTotal in order to get an estimate for their date of first appearence. The results are shown in Figure 4.3. As can be seen, the majority of samples originates from recent years with an emphasis on 2015 and later.

We believe that this concept gives a solid answer to RQ_1, which should be sufficiently covered by the sum of these efforts.

4.4. A Comparative Structural Analysis of Windows Malware

After the composition and realization of a representative malware corpus suited for static analysis has been discussed in the previous sections, we now want to use this

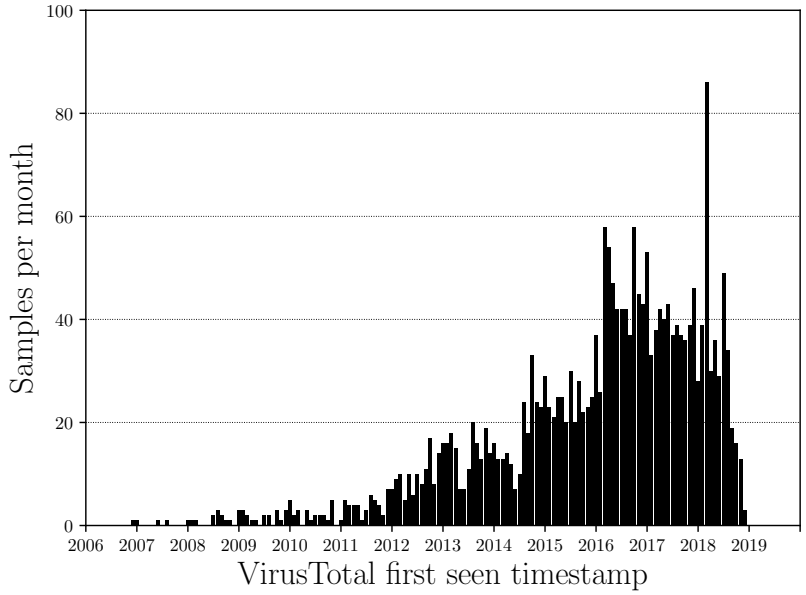


Figure 4.3.: FirstSeen timestamps for 2,290 dumped Windows samples found on VirusTotal, grouped by month.

data set to answer research question RQ_2, addressing payload integrity and malware author methodologies. As stated initially, this dissertation focuses on x86/x64 Windows malware as it is without question the most-targeted and historically richest environment for malicious software. For this reason it also still remains the most relevant platform where most analysis methods and tools focus on.

Even with this limitation, the Malpedia corpus allows us to perform a comparison across several hundred malware families that should yield an unseen impression of the design and implementation choices of the various malware authors over several years. The analysis will be limited to those families of malware for which at least one memory dump (in accordance with the requirement of Unpacked Representation (REQ_A)) of a sample exists, which is a result of RQ_2's goal of targeting payloads only. This leaves us with a comparison of 839 malware families. A full listing of these families is provided in Appendix A.

In some cases where we have dumps for multiple samples of a family available, the observed values will not be in concordance with each other. As some of them are enumerative declarators, we can not average them. Instead, we count the occurrences and present the value observed most often as it is then the most representative for the population of samples analyzed.

As we are interested in meta data, the analysis will primarily focus on the availability of PE header information [21]. This is also possible, because this information is available for a surprising 96.42% of families examined (cf. Section 4.4.2). The analysis is divided along the groups of fields in this layered data structure. In this context, we will conduct

an assessment of the reliability of the information contained in these fields, i.e. if meta data values concerning provenance and structure are genuine.

The outcome of this analysis has different implications. First, structural correctness with regard to section layout is especially of importance to assess the viability of memory dumps as a preferred normalization form in the Malpedia corpus. Second, the outcome influences the applicability of static analysis techniques in general and therefore also for the methodology proposed in Chapters 5 and 6. Finally, the information gathered in this analysis by itself will provide helpful insights into the workflow and tool-chain choices of malware authors.

Before presenting the results of our analysis, we shortly outline the methodology applied for parsing and extracting PE header fields.

4.4.1. Methodology

We perform the analysis of the PE header information in resemblance of the sequence how the addressed structures are encountered in a PE file. A short recapitulation of the PE header is provided in Section 2.1.2. We limit ourselves to only those fields and structures that are commonly inspected in the context of malware analysis and are expressive in early triage (see Figure 2.1), thus helping to characterize the tool chains and methodologies used by malware authors, as is the focus of RQ_2. In the following, we divide the analysis into five parts and examine the reliability of selected fields along their presence: DOS Header, COFF File Header, Optional Header, Data Directories, and finally Section Table.

While the margins in which valid PE files can be crafted are surprisingly wide [249], the PE header itself follows a quite strict offset based structure for the most relevant fields in both 32 and 64bit files that benefits their parsing. We opt to not rely on one of the available libraries for PE parsing because they usually rely on the presence of the iconic file magic `MZ` in order to operate at all. This decision is based on the experience, that these magics may be purposefully removed or altered in malware in order to thwart detection or heuristics used by automated analyis systems. One famous example is the malware `win.plugx`, which changes the magics `MZ` and `PE` to `XV` after injection into memory as observed by Szappanos [250]. Instead, we use the following method (that we will call `pe_check` in the following) to identify the presence and location of potentially modified PE file headers.

We start by scanning the given input file for the first occurrence of all of the following `WORD` values, potentially identifying the start of the Image File Header:

- `4C 01`: I386
- `64 86`: AMD64
- `02 00`: IA64

We then perform a backward search to check if the `DWORD` location prior to the value found (which should be the start of the `NtHeader`, usually marked by the magic `PE`) is referenced by a matching `e_lfanew` field (short for *long file address of New Executable*

	Families			Samples		
Feature	True (%)	False (%)	n/a (%)	True (%)	False (%)	n/a (%)
`has_mz_magic`	805 (95.95)	34 (4.05)	0 (0.00)	2,199 (93.49)	153 (6.51)	0 (0.00)
`pe_check`	809 (96.42)	30 (3.58)	0 (0.00)	2,219 (94.35)	133 (5.65)	0 (0.00)
`pefile`	805 (95.95)	34 (4.05)	0 (0.00)	2,199 (93.49)	153 (6.51)	0 (0.00)
`has_dos_string`	779 (92.85)	60 (7.15)	0 (0.00)	2,060 (87.59)	292 (12.41)	0 (0.00)
`has_rich_header`	570 (67.94)	269 (32.06)	0 (0.00)	1,638 (69.64)	714 (30.36)	0 (0.00)

Table 4.4.: Overall availability and presence of field values in the DOS header.

header). This provides us independence from the explicit file magic `MZ` and header magic `PE`, while still providing sufficient orientation to be able to extract all fields of interest.

The remaining procedure is straight-forward parsing of fields at their respective offsets, as designated by the PE/COFF standard [21]. All aggregation and analysis has been performed with a set of Python scripts in the following referred to as `malpedia-analytics`.

4.4.2. Evaluation of Availability and Reliability of PE Header Information

We now apply the presented methodology to all elements of the PE header and focus on the fields that potentially carry information revealing hints on the methodology used by malware authors.

DOS Header

We first focus on the first header component of the PE header: the DOS Header, including its prominent magic `MZ` and the DOS stub. Here, it is of specific interest if malware authors modify these values in order to make their malware less detectable by heuristics of automated analysis systems. We summarize our results in Table 4.4.

PE Header Availability. Performing `pe_check` on the data set, it can bee seen that a total of 809 (96.42%) of families pass this check. This is a pleasant result, as it implies that in almost all cases data of the PE file header is potentially present that can be examined for further meta information. With regard to the presence of the MZ magic, we can indeed see that at least four families seem to remove these magics only but potentially leave the remainder of the header intact, which was confirmed by manual inspection. We used the popular Python library pefile [251], which was also able to parse data from 805 families, which corresponds with the number of observed MZ magics.

We have furthermore manually reviewed all 153 samples that did not start with the `MZ` magic. The cases identified can be further divided into subgroups where dumps start with a destroyed header (42 samples in 14 families with larger portions or full header nulled, 18 samples in 12 families with only the MZ and PE magics overwritten or nulled, 1 sample overwritten or "encrypted" with a single byte XOR key) or no header at all (53 samples in 24 families starting with (position independent) shellcode, 39 samples in 15 families starting with referenced data instead). Please note that the number of individual families exposing these characteristics is slightly higher than the 34 being

stated not having a `MZ` magic, because some families have a majority of samples without a modified header, triggering the majority decision as explained earlier or the authors having experimented with multiple categories in different versions of the malware.

DOS String. Another prominent feature typically found in PE headers is the so-called DOS string. This structure is part of the DOS stub and contains a series of machine instructions that print the actual DOS string in case the program is executed in a non-compatible legacy MS-DOS environment. Its presence was determined for 779 (92.85%) families, with 3 different variants having been observed:

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

The variant `This program cannot be run in DOS mode` is way more common and appeared 1,930 times in cases when a DOS string was observed in a sample. The second and third variant `This program must be run under Win32/Win64` seem to be an alternative used by Borland compilers exclusively and occur in 131 files. For the cases where a `MZ` magic is preserved but no DOS string observed, we assess multiple case-specific reasons. We have proof for intentional manipulation, e.g. in the case of all Zeus-related families in whose build script the DOS header is scrubbed, as can be inferred from the leaked source code [252]. We also observed the string not being present because of side effects, such as compressed PE header fragments (specifically by the packer `MPRESS`).

Rich Header. The Rich Header is a very interesting structure in the PE file because it is one of the few elements that may contain information about the system environment in which the binary was compiled. According to Webster et al. [25], the Rich header is a proprietary and MSVC-specific structure that contains counter values tracking how many times specific versions of the MSVC compiler and linker have been used on the system in order to produce a given binary. Given knowledge about the release date for these compiler versions, the Rich Header can be used to determine a lower (time) border after which a sample has been most likely compiled. In case MSVC is updated, the combination of tool versions may become very characteristic for a family.

We observe the Rich Header to be present for 570 (67.94%) families. This already inidicates that a majority of the reviewed Windows malware is likely compiled using the MSVC toolchain. The Optional Header (further investigated in Section 4.4.2) contains a MajorLinkerVersion and MinorLinkerVersion field inserted by the compiler tool chain. It can be used in conjunction with the RichHeader to examine common plausibility.

In the following analysis, we use the mapping published by Webster et al. [25] to identify if the respective version of MSVC is contained among the Rich Header entries. As shown in Table 4.4, there are 1,638 samples in which a Rich Header is present. After discarding 36 cases where the linker version field was nulled, we assess that in 1,571 of the remaining 1,602 cases (98%), the linker version is actually found among the Rich Header fields. In nine of the mismatched cases, the linker version corresponds to Visual Studio 2017, which is not covered in the mapping. For the remaining cases, no profound reason for the mismatch could be established.

	Families			Samples		
Feature	True (%)	False (%)	n/a (%)	True (%)	False (%)	n/a (%)
`has_pe_magic`	805 (95.95)	34 (4.05)	0 (0.00)	2,203 (93.66)	149 (6.34)	0 (0.00)
`is32`	793 (94.52)	16 (1.91)	30 (3.58)	2,114 (89.88)	105 (4.46)	133 (5.65)
`timestamp`	761 (90.70)	48 (5.72)	30 (3.58)	2,095 (89.07)	124 (5.27)	133 (5.65)
`dll`	237 (28.25)	572 (68.18)	30 (3.58)	765 (32.53)	1,454 (61.82)	133 (5.65)
`exe`	598 (71.28)	211 (25.15)	30 (3.58)	1,454 (61.82)	765 (32.53)	133 (5.65)

Table 4.5.: Presence of field values in the COFF file header.

COFF File Header

The second group of fields that we want to investigate are associated with the COFF File Header. Again, with author methodologies in mind, fields of special interest are the PE magic, the bitness and file characteristic, and the compilation timestamp. The number of sections is addressed along with other details on this structure in Section 4.4.2 instead. The results in Table 4.5 give an overview of presence for these values.

PE Magic. Observations for the PE magic are mostly similar to what was already established in Section 4.4.2. There are notably only 4 cases in which only the MZ but not the PE magic have been manipulated.

Bitness and File Characteristic. With respect to bitness, the vast majority (739 families or 94.52%) of families inventorized feature primarily 32bit code. We find the following two explanations for this. First, the system from which the majority of dumps have been produced (59.33%) is the Windows XP VM, which is a 32bit exclusive operating system. Secondly, writing malware in 32bit has the advantage that it can be executed on both 32bit and 64bit versions of Windows, in the latter case falling back to using WOW64 (Windows-On-Windows 64-bit). Nevertheless, we have noted that at least 22 families also have code as 64bit version available, in most cases along a 32bit version and shared loader that deploys the appropriate version. It has to be noted that 64bit variants are necessary in order to inject into and manipulate native 64bit processes, e.g. to perform hooking.

Additionally, using a dedicated YARA signature and manual verification, we have determined that at least 36 families make use of the so-called Heaven's Gate technique [253] in the variant offered by ReWolf's x86 helper library [254]. This technique allows to intercept the switching procedure during requests to the 64bit kernel in 32bit WOW64 environments, effectively allowing the execution of 64bit code within the same process. It can be used for example as an anti-analysis measure or to inject code into other (native) 64bit processes. It furthermore implies that some malware will contain both 32bit and 64bit code within the same sample, which is a case not covered by most current analysis tools. The malware family `win.nymaim` is at least one publicly documented case where a malware author intentionally used hybrid and even polyglot (semantically equivalent in 32bit and 64bit) code [255].

Another interesting aspect is that 237 (28.25%) of the families' primary dumps are identified as DLLs. From a pure technical point of view of a malware author, the

difference between EXE files and DLL files is not significant, as both have a designated entry point that can be used to start arbitrary execution. They mostly differ in the operational perspective, i.e. how they are supposed to be started up. Here, DLLs need help for startup (e.g. rundll32.exe, another potentially customized loader, or by making use of DLL side-loading [256]) while EXEs can directly spawn as processes themselves. Therefore, from a methodology point of view, this implicates that many malware authors seemingly have adopted a modularized or multi-staged approach to develop their malware.

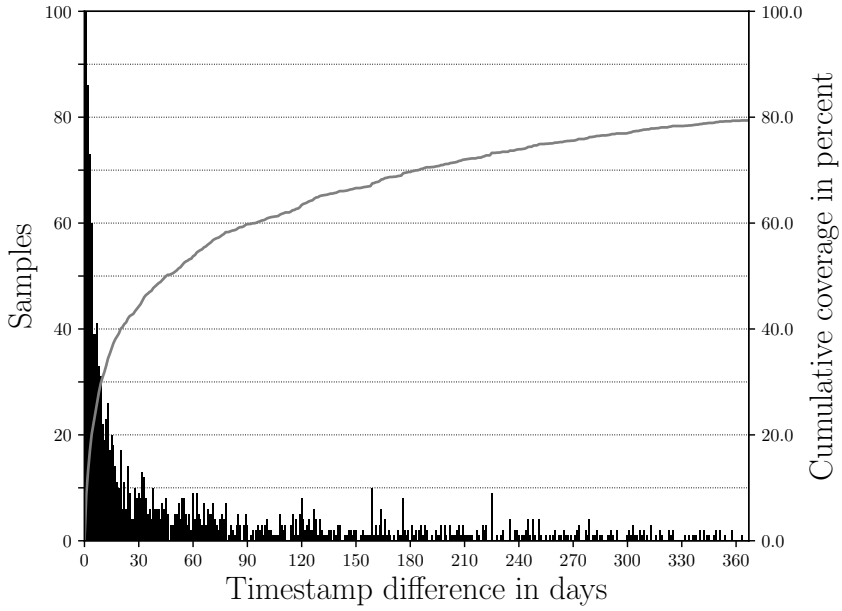


Figure 4.4.: Difference between VirusTotal FirstSeen and PE compilation timestamps.

Compilation Timestamp. A header field that can provide interesting temporal context about a malware author's working habits and/or the provenance of certain versions of a family is the PE compilation timestamp. It is a `DWORD` sized value representing a UNIX Timestamp and therefore given in UTC [21].

As research question RQ_2 addresses how reliable PE header fields in unpacked malware are, we will now compare the values found in the PE headers with an external reference, in this case the FirstSeen timestamps of VirusTotal (cf. Section 4.3.4).

Table 4.5 shows that 2,095 (89.07%) samples have a compilation timestamp that is potentially considered valid, i.e. not zero and not the fixed Delphi Timestamp of 708,992,537 (June 19th, 1992 22:22:17) [257]. In order to compare values with the FirstSeen dates as provided by VirusTotal, they actually have to be found on that platform, which is the case for 2,041 (97.42%) of these samples, that we now denote as T_{vtpe}.

These samples can further be reduced to a set of potentially meaningful values. We first exclude samples with PE timestamp values before January 1st, 2006 (the year in which the earliest sample of our collection has been confirmed) and values beyond the data set snapshot taken on 3rd of January 2019. One more causal constraint is that the

compilation timestamp is situated before the FirstSeen timestamp, which leaves us with 1,911 samples having date pairs of interest, that we denote in the following as T^*_{vtpe}.

Figure 4.4 shows the distribution of differences between FirstSeen and compilation timestamps for T^*_{vtpe} in days, limited to one year.

As many as 28.1% of the samples are observed within one week, and 44.69% within the first 30 days. Overall, almost 79.38% of the samples show a timestamp difference of at most a year. With regard to outliers, 716 families have timestamps in T^*_{vtpe} and 595 of these 716 families also have values within this one year range.

While there is no absolute ground truth available (and we only compare against VirusTotal), we believe that this examination across the Malpedia data set indicates that PE compilation timestamps will more often than not contain reasonable values. This is surprising, as they give away useful information to analysts while being trivially to tamper with. But seemingly, many malware authors and operators do not spend attention to this.

Optional Header

The COFF File Header is followed by the Image Optional Header. While the COFF File Header details how contents are organized within the file, this structure primarily contains information about runtime requirements and how it is supposed to be mapped into memory for execution. Since we are interested in reliability of information, we focus on the Major- and MinorLinkerVersion that provides us provenance information about the program's creation, required OS version and security features enabled during compilation.

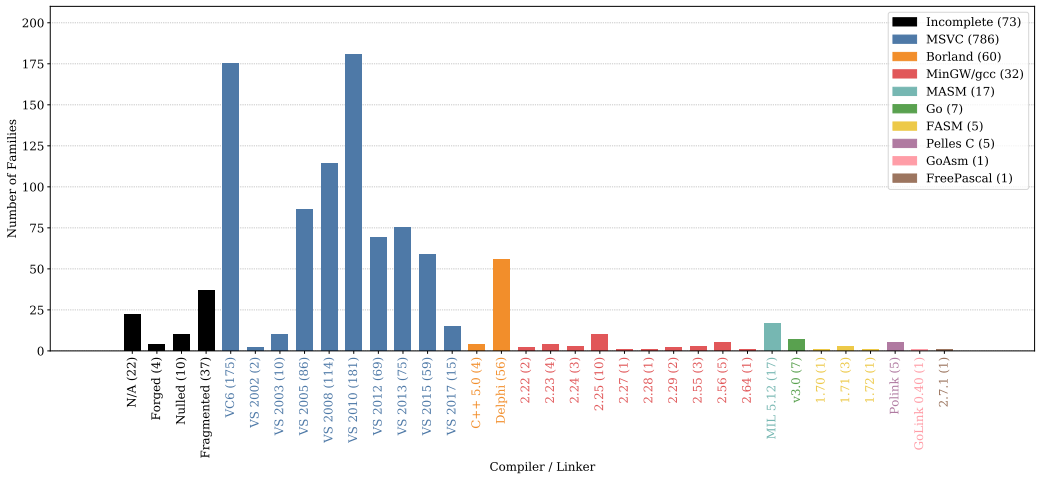


Figure 4.5.: Overview of occurrence frequencies of Linker versions, counted once per family.

Major/MinorLinkerVersion. The Major- and MinorLinkerVersion is typically inserted by the compiler/linker tool chain during creation of the executable. Best to our

knowledge, no standardized registry of these values exists. However, we assume that the creators of the most popular tool chains are aware of each other's work and tend to pick unique values as identifiers.

With regard to mappings from Major- and MinorLinkerVersion fields to the respective tool chains, no official data sources exist. We have used the database of the popular file identifier Detect-It-Easy [258] as a basis and then adjusted and extended it for integration into `malpedia-analytics`. Detect-It-Easy was considerable more granular and maintained as other alternatives, such as libmagic [259] as integrated in GNU/Linux or the outdated tool PEiD.

Figure 4.5 gives an overview of the distribution of values observed across all families, with values per family only counted once. Using this methodology, we observe a total of 987 data points. These can be identified as nine different tool chains in 30 versions which make up 95.54%, and 44 cases where the information was likely manipulated, fragmented (e.g. by overlapping data of a compressed header) or missing.

Microsoft Visual C++ (MSVC) as offered by Microsoft's Visual Studio (VS) product line is by far the most popular tool chain, contributing 82.57% of the data points and making an appearence as compiler in 701 families. Even though the majority of files in the data set has been collected 2013 and later (cf. Figure 4.3), Visual Studio 2010 sticks out as most commonly observed (181) overall. Additionally, Visual Studio 6, although released in 1998, is closely following with 175 occurrences. A primary reason for this appears to be that its library version of `msvcrt.dll` was the one delivered in Windows XP and beyond by default, meaning that programs compiled with VS6 would be compatible out of the box and not be affected by the so-called phenomenon of "DLL Hell" [260, 261]. The other versions between 2005 and 2015 register between 59 and 114 observations, while earlier versions and the most recent ones at the time of writing (VS2017) were observed 20 times or less.

Only MSVC generates Rich Headers, which can be used to cross-check the validity of the values found in the Major- and MinorLinkerVersion. As has been established in Section 4.4.2), in 98% of the cases where Rich Header and linker version information are available, the linker version in fact appears with its corresponding Product ID in the Rich Header.

The remaining tool chains sum up to 12.97%. MinGW appears to be the primary alternative C/C++ compiler being used by malware authors, but is also only found 32 times in 29 families. The code of four families was generated with Borland's C++ compiler, while 56 families were written in Delphi and also compiled with Borland's tool chain for this programming language. This also includes modern versions of Delphi, which are now maintained by the software vendor Embarcado but keep the same linker version (2.25).

Another 23 families appear to be written in Assembler directly and being linked using Microsoft Incremental Linker (MIL, 17 families), FASM (5 families), and GoLink (1 family). Seven families are written in Go and compiled using the corresponding compiler in version 3.0.

	Families			Samples		
Feature	True (%)	False (%)	n/a (%)	True (%)	False (%)	n/a (%)
ASLR	446 (53.16)	363 (43.27)	30 (3.58)	1,321 (56.16)	898 (38.18)	133 (5.65)
NX	414 (49.34)	395 (47.08)	30 (3.58)	1,116 (47.45)	1,103 (46.90)	133 (5.65)
NX&ASLR	393 (46.84)	416 (49.58)	30 (3.58)	1,042 (44.30)	1,177 (50.04)	133 (5.65)
SEH	655 (78.07)	154 (18.36)	30 (3.58)	1,849 (78.61)	370 (15.73)	133 (5.65)

Table 4.6.: Presence of field values in the Optional header.

DLL Characteristics. The DLL Characteristics field is a bitmask that contains information about the compatibility with various security mechanisms. Table 4.6 gives an overview of the availability of the most relevant fields. While Structured Exception Handling (SEH) is supported by 78.07% of the families, only a mere half supports Data Execution Prevention (DEP) as realized through use of the non-execution bit (NX) mechanism, and same holds true for supporting Address Space Layout Randomization (ASLR). Both DEP and ASLR are supported since VS2005 and by default activated since VS2008 [262].

The fact that these numbers turn out lower than what could be expected based on the Linker evaluation may be explained by malware authors wanting to remain able to overwrite and modify code at runtime.

Operating System Version. The Operating System Version field in the PE header indicates the minimum version required to run the given binary. This is one of the strict constraints and in fact, execution will not even start in case the OS version is lower than specified in this field.

For 2,162 out of 2,219 (97.43%) samples with a parsable header, the field assumes a meaningful value. The most common value is "5.1", indicating the very popular Windows XP as its required version. The value "4.0" (Windows NT 4.0) is the second most common value with 792 occurrences, followed by "5.0" (Windows 2000) with 268 occurrences.

To our surprise, 71 samples indicate "5.2", which is the 64bit version of Windows XP. After manual inspection, all of these samples are actually 64bit samples and the version required is the lowest OS version possible matching this bitness. While Windows XP 64bit is amongst the rarest of all Windows versions observed being in use, we believe it has been chosen to enable maximum compatibility to ensure the malware will run on any 64bit target presented. Beyond XP, 124 samples require OS version "6.0" (Windows Vista) and 2 require "6.1" (Windows 7).

For 57 samples, this field has been nulled or set to "1.0", which we consider an irregular or modified value.

Data Directories

Next in sequence after the Optional Header are the 16 Data Directory entries. Results for all directories are shown in Table 4.7, the fields are discussed in thematical groups, generally following the order of appearence.

Feature	Families True (%)	Families False (%)	Families n/a (%)	Samples True (%)	Samples False (%)	Samples n/a (%)
Export	217 (25.86)	592 (70.56)	30 (3.58)	637 (27.08)	1,582 (67.26)	133 (5.65)
Import	774 (92.25)	35 (4.17)	30 (3.58)	2,062 (87.67)	157 (6.68)	133 (5.65)
Resource	561 (66.87)	248 (29.56)	30 (3.58)	1,281 (54.46)	938 (39.88)	133 (5.65)
Exception	25 (2.98)	784 (93.44)	30 (3.58)	102 (4.34)	2,117 (90.01)	133 (5.65)
Security	33 (3.93)	776 (92.49)	30 (3.58)	55 (2.34)	2,164 (92.01)	133 (5.65)
BaseRelocationTable	575 (68.53)	234 (27.89)	30 (3.58)	1,699 (72.24)	520 (22.11)	133 (5.65)
Debug	194 (23.12)	615 (73.30)	30 (3.58)	456 (19.39)	1,763 (74.96)	133 (5.65)
Architecture	1 (0.12)	808 (96.31)	30 (3.58)	3 (0.13)	2,216 (94.22)	133 (5.65)
GlobalPtr	0 (0.00)	809 (96.42)	30 (3.58)	0 (0.00)	2,219 (94.35)	133 (5.65)
TLS	79 (9.42)	730 (87.01)	30 (3.58)	139 (5.91)	2,080 (88.44)	133 (5.65)
LoadConfiguration	269 (32.06)	540 (64.36)	30 (3.58)	655 (27.85)	1,564 (66.50)	133 (5.65)
BoundImport	10 (1.19)	799 (95.23)	30 (3.58)	16 (0.68)	2,203 (93.66)	133 (5.65)
IAT	660 (78.67)	149 (17.76)	30 (3.58)	1,813 (77.08)	406 (17.26)	133 (5.65)
DelayLoadImport	31 (3.69)	778 (92.73)	30 (3.58)	65 (2.76)	2,154 (91.58)	133 (5.65)
COMRuntime	103 (12.28)	706 (84.15)	30 (3.58)	159 (6.76)	2,060 (87.59)	133 (5.65)
Reserved	0 (0.00)	809 (96.42)	30 (3.58)	0 (0.00)	2,219 (94.35)	133 (5.65)

Table 4.7.: Presence of field values in the Data Directories.

The number of families with an Export directory (25.86%) strongly corresponds to the families that are developed as DLL (28.25%). The slight difference results from the fact that a DLL does not necessarily need exported functions as execution can also be taken over through an optionally defined entry point, commonly a DllMain routine [263].

However, a DLL without any exports should immediately raise suspicion as a DLL's purpose is as its name implies to provide additional functionality by serving as library.

With regard to importing functions, almost all families (92.25%) specify an Import directory, meaning that they are dynamically linked against additional libraries or the Windows API. This makes a lot of sense as the Windows API is the central interface for interaction with the Windows operating system (cf. Chapter 5). On the other hand, only 78.67% have an explicit Import Address Table (IAT) directory configured. The IAT is the expected natural runtime counterpart to the Import Table. Its directory exists mostly for reasons of optimization as the region specified here is designated for Copy on Write, thus excluding them from memory otherwise shared across processes (which is the case for most Windows system modules). The Bound Import directory allows a program to be linked against a fixed version of a module. Of the ten families that feature this directory, 9 are written in VisualBasic and require the respective runtime in version 6.0. For `win.slingshot` however, the bound imports might have been used to tailor the malware to blend in with or tie it to a given target, as it was designed against MikroTik routers running Windows 2003, which matches the timestamps in the bound import table entries [264].

Delay Imports are specified for 31 families and enable a program to load additional DLLs at call time. Here, no consistent usage pattern could be identified. The topic of Windows API usage is discussed in detail in Chapter 5.

A Resource directory is found for 66% of the families. It is a very flexible structure and some of its primary use is to carry auxiliary data in pre-defined structures. Many

of its use cases address the appearence of a graphical user element, with elements such as custom program icons, cursors, or images, embedded fonts, defined menu structures and dialogues. It also enables multi-language support in accordance with the system's localization settings. Another prominent data type is version information about the program. As the resource section can be used to store (binary) strings, some malware families embed additional payloads or modules in this data structure, contributing to its popularity.

The Exception directory is found in only 3% of the families, which is explained with the fact that it is only available for 64bit binaries, which are rare in the corpus (cf. Section 4.4.2). The Security directory is similarly uncommon as it is only found in 4% of families. Its presence indicates that the file has been signed with a certificate in order to establish trust in the file's integrity. Signing binaries is a prominent method to enable their execution in environments with heightened security controls, and 63% of the families with signed samples are also attributed to APT activity.

More common is the Relocation Table with 68% presence among families. It is used to allow the binary to be loaded at different base addresses, which is especially important for operating systems of Windows 8.1 and above.

Notably, 23.12% of the families contain a Debug directory. Similar to the Rich header (cf. Section 4.4.2), this data structure can contain data containing information about the computer the binary was compiled on. Here, the location entry of the auxiliary Program Database (PDB) file is of relevance as it contains a local path. In total, 441 PDB paths could be extracted from samples, which is almost 4 times as high as the result noted by Miller [265]. Depending on the location of the project, this path may contain the user name on the system or the self-chosen name of the author. We observe a total of 51 unique user names that were non-standard (i.e. different from arbitrary names such as Administrator, Admin, User, ...). While a majority of them appear to be nicknames, some also are first and last name pairs in the clear. Additionally, the paths contained parts that correspond to the name with which the respective malware families are publicly referred to, meaning that this one of the typical resources used for naming. As we did not pursue this beyond the observations shared here, authenticity of the data is not established but it appears plausible that this field may contain highly interesting data.

While the Architecture field is supposed to be zero according to the PE/COFF standard [21], 3 samples have a non-zero field. All of them are written in Delphi and it appears that under some circumstances, a Borland compiler may set a pointer to the internal project name in the Architecture field. The Global Pointer as well as the Reserved directory were indeed zero for all files.

The TLS (Thread Local Storage) directory, which is present for 9.42% of the families, can be used to implement additional per-thread setup/tear-down functionality including private storage apart from the stack. Notable in this context are TLS callbacks, which can be used to execute code prior to the entry point. In this case, 125 of 139 samples have such a callback, which is also commonly observed in packers.

Finally, a LoadConfiguration directory is found for 32.06% of the families. This directory is tied to the SafeSEH security feature, and the number is only about half as big as families generally supporting SEH (cf. Section 4.4.2).

Section Tables

The section table of the PE file format provides information about the overall program structure. For example, it lists details about code and data regions including their names (which hints towards their purpose), sizes, and how they are generally mapped into memory with access flags read, write, and execute. The availability and consistency of this information is highly relevant for further analysis of the malware. We will first check the section tables encountered in malware against the recommended values from the PE/COFF standard [21] in order to get an impression of information availability. In a second step, we evaluate consistency of table data versus actual content by using CodeScanner (in its version available as of August 2019) by Zwanger et al. [266] to identify if indeed only sections marked as executable primarily contain code.

Number and Characteristics of Sections. We first investigate if the section table contains a meaningful number of sections and well-defined names and flags.

With regard to the number of sections, samples with three (442), four (661), or five (716) sections make up 81.97%, most of them consisting of a combination of the standard section names `.text`, `.rdata`, `.data`, `.reloc`, and `.rsrc`. 57 samples have one or two sections, while 177 have six, 53 have seven, and 60 have eight sections. Another 52 samples have nine to thirteen sections and a single outlier has 255 (manual inspection indicates an intentionally modified value likely to confuse analysis programs).

Out of a total 10,074 section names, 9,117 are equal to one of the reserved section names as given in [21], with 8,819 of them having the required combination of section flags. Furthermore 427 section names are considered the extended standard category, as they simply are an alias to one of the reserved names, e.g. `CODE` instead of `.text` as typically used by Borland compilers. Another 208 section names consistently appear only in conjunction with specific well-known packers or unpackers, such as UPX, ASprotect, Upack, MPRESS, and VMprotect. The remaining 277 section names appear manipulated (e.g. nulled (90), and some even contain non-printable binary data (5)).

On sample level, 1,692 samples (76.25%) have exclusively standard names, another 288 have one or more sections that fall into the extended standard, 94 with packer sections and 143 have one or more non standard sections. Altogether, this leaves the impression that the vast majority of section names as found in the headers are highly consistent with native compiler output.

Consistency of Table Data and Memory Layout. In the second step, we now apply CodeScanner [266] to all memory dumps and compare the results with the PE header's section table information where possible. CodeScanner is a byte occurrence frequency and pattern-based detection tool that can be used to classify a given buffer into regions such as code and ASCII/binary data.

For the following evaluation, we use the executable memory regions as defined by the PE header as expected code region and measure if and how much code is additionally located by CodeScanner outside of these regions. For 133 out of 2,352 samples, no PE header information could be obtained (cf. Section 4.4.1)). For another four samples, the PE header was fragmented in a way that no valid section offsets could be extracted and CodeScanner did also not detect any code.

For 216 samples, CodeScanner did not identify code regions. By manually inspecting these, we found that 163 of these were either .NET or VisualBasic samples, which can not be detected by the version of CodeScanner available to us, which is only capable to find x86 and x64 code regions. Looking at the remaining 49 samples, the code region generally was tiny (potentially smaller than CodeScanner's window size) or obfuscated, also resulting in no detection.

For 43 samples, only CodeScanner produced results. Here, 29 samples had virtual offsets pointing outside of the buffer and through manual analysis we concluded that most likely junk bytes have been inserted to fool analysis programs. In the remaining 14 samples, no executable bit was set, which may be a result of a custom loader reading the PE and not relying on this information (e.g. because all memory is set to read-/write/executable anyway), meaning that CodeScanner correctly identified additional code.

This leaves us with 1,956 samples (83.16%) for which results can be compared and that can be evaluated in the proposed way. Among these, the code regions detected by CodeScanner are fully contained within the section borders for 702 samples. For another additional 1,082 samples, the overlap is between 90 and 100%, which in combination sums up to 91,21%. Upon closer inspection of the remaining 172 samples, we find that for 53 of them, the code section is under 20kB, meaning a single misclassified window in CodeScanner will have the samples already drop below the chosen 90% overlap border. Indeed, misclassification was confirmed for all of these cases, in some cases with an additional CodeScanner window before or after the code section. For another 65 samples, CodeScanner correctly identified an additional embedded PE file, a typical method of malware to carry an auxiliary module, e.g. in the Resource section. For 12 cases, CodeScanner made a detection of code for another architecture, which was confirmed as Heaven's Gate code to dynamically change between 32bit and 64bit execution. This leaves us with 42 uncategorized cases. For these, we identified a mix of correctly identified shellcode in data sections and minor misclassifications by CodeScanner that led to a code section coverage below 90% as outlined before. Overall, CodeScanner has provided very accurate and consistent results that established for us that the section table layout as contained in the PE header of the payload dumps appears to be genuine in a vast majority of cases.

4.5. Summary

In this chapter we examined the availability of quality ground truth for static analysis in malware research and observed a severe need for reliable data.

This led us to RQ_1, asking how a malware corpus should be composed to both enable representative research from academic viewpoints while similarly serving as a relevant resource to practical malware analysis. To answer this question, we specified requirements around the three core categories of Representativeness (REQ_R), Accessibility (REQ_A), and Practicality (REQ_P). These were then carefully validated against Rossow's Prudent Practices, showing that they indeed would yield a data set as prescribed by an accepted standard.

In the following, we presented our implementation of such a malware corpus named Malpedia. We showed that our concept and realization indeed fulfil all aspects of the specified requirements. This included careful documentation of the collection and processing applied, including a summary of observations gathered during the processing. As of January 3rd, 2019, Malpedia consisted of 3,469 representative samples for 1,136 malware families, making it the most comprehensive malware corpus (in regards of verified malware families) available to research.

With this corpus at hand, we approached RQ_2, asking about the integrity of payload meta data and file structure in unpacked Windows malware as well as what can be inferred about tool chains and methodologies used by malware authors. To answer this, we conducted a thorough comparative structural analysis of the 839 Windows malware families with unpacked representations present in Malpedia. In summary, we found that meta data in the form of PE headers was available for 96.42% of the malware families. Furthermore, fields contained in this meta data were plausible in a vast majority of cases, showing only few cases of tampering. This has been both shown by the comparison of Rich Headers with compiler/linker fields and the study of payload integrity with respect to the section table. These findings especially indicate that packers and protectors almost always can be seen as just an initial barrier that by itself does not focus on aggravating static analysis, e.g. by further concealing payloads after deployment.

As for author methodologies, Visual Studio strongly dominates as a development platform, with the seemingly outdated versions VS2008 and VS6 being among the most common. It was surprising to see that for almost a fourth of all samples, hints on debug information were contained in the sample, including filepaths from the author's machine and in more than 50 cases even a username. This shows that ignorance or even carelessness to such easily mitigatable details is not uncommon among authors or operators of malware, unless it is planted intentionally as a false flag as observed e.g. by Kaspersky in the OlympicDestroyer campaign [267].

As a final remark, it should be noted, that the relative values derived in Section 4.4 are very close to what we reported before in the previous publication [12] this chapter is based upon, although more than twice as many families have been covered in this evaluation. We conclude that this stability indicates reliability for the generalization of the results.

5. Robust Recovery and Analysis of Windows API Usage

After explaining our efforts of a structured collection and cursory structural analysis of malware, we continue in this chapter by focusing on a crucial cornerstone for in-depth malware analysis: API usage information. As motivated earlier in Section 4.4, we continue to focus on Windows malware as it is the most relevant environment. We highlight the importance of availability of API information and present our approach for robust recovery and comparative analysis of extracted Windows API usage profiles. The information gathered allows us to study the spectrum of malicious capabilities in great detail and provides better understanding how malware authors interact with the API.

We first start with a motivation and define our main research questions and contributions in Section 5.1. Next, we introduce our proposed approach *ApiScout* for the recovery of Windows API usage information from memory dumps in Section 5.2, reliably automating one of the most relevant tasks in the preparation of in-depth malware analysis. In Section 5.3, we use ApiScout to perform an analysis of Windows API usage across Malpedia. We follow up with a method to store and compare extraction results of ApiScout that we call *ApiVectors* in Section 5.4 and benchmark the performance when using ApiVectors for malware classification. We conclude with a summary in Section 5.5.

This chapter follows in large parts our previously published methodology and results as presented in [12, 13] but it significantly expands the evaluation of the method itself as well as the number of covered malware families. Furthermore, it is in line with the data set status as defined in Chapter 4 and supports the purpose of providing a coherent picture throughout this dissertation.

5.1. Motivation and Contribution

When working on incidents and newly identified malware families, analysts often work under high pressure towards goals such as ensuring protection and mitigation of threats. A core question that usually quickly arises has its origins in risk assessment and asks about the potential capabilities of a given malware. To answer it, analysts typically have to accept trade-offs between the timeliness, depth of analysis, and accuracy of conclusions they are able to make [268]. To save time, a typical first step in analysis is the use of automated dynamic methods such as blackboxing and sandboxing. However, these methods can only give a limited answer because they are typically performed with a short execution time frame. In consequence, they will only capture a limited number of

behaviors exhibited in that time frame. Moreover, because malware often acts reactively and in response to an external control entity, it may depend heavily on external resources such as network communication and availability of the malware's C&C servers.

For results of more depth and accuracy, reverse engineering has to be applied. Effective reverse engineering requires situational awareness in order to maintain orientation [38] and one central cornerstone for this are Windows API interactions. Following these interactions is among the most effective ways to locate code tied to certain behaviors of interest [38]. For example, in order to investigate the communication with a C&C server, an analyst will typically first identify API functions that are tied to networking, and then investigate nearby areas in the code where these functions are referenced. As we have learned in Chapter 4, malware is typically encountered packed and proper analysis requires unpacking. Luckily, Chapter 4 has also shown that packing in modern malware can be considered a mere barrier that can be successfully circumvented, e.g. by an approach differencing memory dumps, yielding a success rate of up to 92% [245]. This means that memory dumps can be easily produced and serve as an effective approximation of unpacking that can be used to jumpstart in-depth analysis.

However, this does not address additional methods used by malware authors to protect their code against analysis. In the context of this chapter, these protections include concealing interactions with the Windows API, e.g. by avoiding the PE header's natural method for resolving references into the Windows API (and other dynamically linked libraries), the Import Table [21]. In case malware uses so-called dynamic imports or even more sophisticated methods of obfuscation, reconstruction of the WinAPI usage information is required to enable proper analysis.

The current state of the art for import usage reconstruction from memory dumps is represented in tools such as *Scylla* or *ImpScan* as found in the Volatility memory forensic framework. Both require access to the full memory layout as well as contents of the target process from which a memory dump was taken of. Scylla also relies on manual interactions and adjustments that can not be executed automatically. Additionally, these tools assume that all import references are stored in a single structure that resembles the Windows-native Import Address Table (IAT) format, which does potentially not hold true for custom methods as observed in malware. With especially the last argument in mind, no formal evaluation of these approaches is available, leaving uncertainty about their effectiveness and accuracy.

Longing for improvement, the first research question for this chapter is the following:

> RQ_3: Using static analysis, how can Windows API usage information be robustly extracted from memory dumps?

Using Malpedia as reference corpus, we quickly follow up with a second research question:

> RQ_4: How frequently do malware authors apply obfuscation schemes to their WinAPI usage?

As an answer to both, we present ApiScout, a fully automated method that is capable of identifying and recovering both static and dynamically created references to

the Windows API from memory dumps. Because ApiScout uses an approach divided in two stages, it does not require a live environment and can be conveniently applied in post-mortem scenarios for malware analysis and memory forensics. As consequence of the principles explained in Section 4.3, this also allows us to apply ApiScout to all memory dumps contained in Malpedia. This way, we can study WinAPI usage in detail and derive an answer for the second question.

Now, given the ability to robustly extract WinAPI usage profiles, we specify a follow-up question:

> RQ_5: How characteristic are Windows API usage profiles for malware families and can they be used in the context of malware identification?

To investigate this question, we need to be able to measure the similarity of usage profiles. We propose the concept of ApiVectors, a vector representation of the most semantically relevant and commonly encountered WinAPI functions. Apart from being an efficient storage method, these vectors can be compared to each other using similarity metrics, serving as an effective classification method for malware.

Contributions. In summary, in this chapter we make the following contributions:

1. We present ApiScout, a method that effectively recovers WinAPI references from memory dumps. With a near perfect F1 score (0.999), ApiScout produces more reliable results than the comparable approaches Scylla (0.893) and ImpScan (0.933).
2. We define a taxonomy for API usage obfuscation. Using ApiScout, we provide the first extensive analysis of the usage spectrum of the Windows API across 726 malware families. We learn that 48% of these families do make use of dynamically imported WinAPI references but only less than 4% use complex custom obfuscation that is not extractable by our method.
3. Based on the API usage data, we create a comprehensive semantic classification scheme, covering 4,994 WinAPI functions. Being almost twice as extensive as schemes from previous works, it allows more precise characterization of capabilities in malware.
4. We propose ApiVectors, a method that allows measuring similarity based on API usage profiles and show that it outperforms current state of the art methods: ImpHash and ImpFuzzy.

5.2. ApiScout: Recovery of Windows API Usage from Memory Dumps

As motivated earlier, it is a common task for analysts to rapidly identify points of interest tied to key behavior aspects such as persistence, network communication, and functional capability in a given analysis target [38]. Revisiting that we have already explained in Chapter 4 why we consider memory dumps equivalent if not superior to classic 'clean' unpacking, this allows us to consider new methods to optimize an analyst's workflow.

State of the art approaches such as Scylla IAT Search and Volatility ImpScan are able to extract API information from memory dumps but have shortcomings. Scylla can not be automated and requires manual interaction and both approaches generally require that the full memory layout of the process for which API import information shall be reconstructed is available, i.e. information about all modules being loaded in a given process context.

We now introduce our method *ApiScout*, which is tailored to memory dumps and intended to ease an analyst's task of robustly reconstructing Windows API usage of malware in a given memory dump. In this context, using memory dumps as the only input data provides flexibility by being generally decoupled from the dynamic analysis execution environment once created.

We define the following requirements for our method. First, it should provide a complete and accurate result, providing coverage for all references to the Windows API. Second, the method should generalize well and especially be applicable for malware analysis. Third, it should have high usability as this is not only important to find acceptance among practicioners, but also to allow for easier integration with other tools and workflows.

In the following, we first explain the ApiScout methodology in detail. Because our later analysis focuses on Windows malware, we present inventarization results for four major releases of this operating system, showing how its API maintained compatibility while also growing significantly in size over time. To show that our approach provides accurate and robust results, we evaluate it against Scylla IAT Search and Volatility ImpScan, using a selection of benign and malicious programs in both 32 and 64bit.

5.2.1. Methodology

ApiScout is a generalization of and inspired by the method proposed as part of the Eureka framework by Sharif et al. [105]. In Eureka, disassembly is used to identify call instructions that likely interact with the Windows API, which are then compared against a database of target function offsets. The database is built dynamically by extracting exported functions from modules, with analysis triggered when a call to the `ntdll.dll!NtMapViewOfSection` WinAPI function is executed.

To fulfil the requirement of generalization, we do not make assumptions about how the Windows API is used as long as valid references to it exist. This means that ApiScout is not based on the context of disassembly for resolving call targets. Instead we show that the method can be used with arbitrary buffers without any additional structural information being required. The methodology is generally divided into two phases: an initial setup phase and its actual application phase. This division specifically enables the decoupling from a runtime environment as explained in the following. A limitation of this approach is that only those function offsets can be located, that are explicitly stored within the target memory and that correspond to actual export offsets from inventorized modules as stored in the database.

The phases are now described in detail. A reference implementation of ApiScout has been written in Python and made publicly available on GitHub [269].

Inventarization Phase

The inventarization phase is a mandatory setup step that is required to capture the specifics of the environment from which the memory dumps originate that are later to be analyzed in the application phase.

In this phase, the target system's file system is crawled recursively in order to identify every executable and Dynamic Link Library by their file extension. These files are then analyzed in order to collect their preferred ImageBase address and all of the exported functions including their Relative Virtual Addresses (RVAs). The RVA is the value at which these exported functions are located once the DLL has been mapped to memory, relative to the ImageBase. Both of these data points are easily located by parsing the PE header in a similar methodology as outlined in Section 4.4, this time focusing on the Export data directory. It has to be noted that the approach has to be capable of treating both 32bit and 64bit systems, meaning that the address length format can be 4 bytes or 8 bytes respectively. The extraction result for every DLL is stored individually within one configuration profile for the given OS and its current state. By storing results individually, this allows to resolve potential address conflicts that may arise through identical ImageBase addresses at a later time. Incorporating every executable and DLL ensures that we achieve the requirement of completeness, as we have data for every potentially imported standard module. Malware authors can generally not make assumptions about their target systems (unless dedicated reconnaissance was performed) and have to rely on these standard modules. In Section 5.3.3, we will later see that in practice, only a small number of DLLs (and no executable) are used to interact with the Windows API.

Because the above described method of database creation is based on full enumeration of the file system, it gives a complete result and by definition is as good as potential alternatives. It follows mostly the method proposed for Eureka [105], described in the section on "Handling DLL obfuscations: DLLs loaded at standard virtual addresses". It only deviates in order to additionally address the specifics of modern OS versions of Windows, namely effects of Address Space Layout Randomization (ASLR).

In this context it is important to note that during the setup phase, every DLL inventorized is loaded once into a stub process to set and identify its individual adjusted load offset. This way, even in case ASLR is activated for the environment, we will still obtain the actual ImageBase and RVA for all DLLs. Loading the modules is also the only known way to derive the ASLR offset, which makes it a necessity. However, this means that the database has to be updated after every boot sequence as this is the moment ASLR offsets are initialized randomly. For the application of ApiScout, this drawback is negligible as it is common practice in malware analysis to have a snapshot of a running system state (e.g. using a virtualization software such as VMware or VirtualBox) to speed up analysis [41]. The same holds true for most sandbox analysis systems. However, full rebuilding of the database is necessary in case the system configuration changes, i.e. when patches are applied, as the memory layout within relevant DLLs may change, shifting the export offsets.

```
Offset    Hexdump                                                          Offset    Value     WinAPI function

008DE000: ❶AC9BDE77 519CDE77 7E9ADE77 11C8DF77 8042DE77 C3BCDF77 D7EADD77 B8EFDD77   008DE000: 77DE9BAC  advapi32.dll!CryptDestroyHash
008DE020: 9651DE77 8F9BDF77 A454DE77 5FD7E077 4278DD77 AB7ADD77 176CDD77 D5ECDD77   008DE004: 77DE9C51  advapi32.dll!CryptCreateHash
008DE040: 1D79DE77 40E3DE77 00000000❷2527A977 2B45AB77 C41FA977 00000000❸1724807C   008DE008: 77DE9A7E  advapi32.dll!CryptHashData
008DE060: 01FE907C D109837C 462C817C 85DE807C 1A1E807C C706817C 64A1807C 9C32817C   008DE00C: 77DFC811  advapi32.dll!CryptVerifySignatureA
008DE080: A7A0807C E917807C C51E837C 4624807C FD0C817C 1218807C B099807C 0F29837C   008DE010: 77DE4280  advapi32.dll!RegDeleteKeyA
008DE0A0: 6B23807C 281A807C 170E817C D79B807C 51AC807C A400917C 7CAC857C 0DFF907C   008DE014: 77DFBCC3  advapi32.dll!RegCreateKeyA
008DE0C0: 31B7807C E19A807C 2E93807C 749B807C 7B1D807C 30AE807C CFE9807C 00000000   008DE018: 77DDEAD7  advapi32.dll!RegSetValueExA
008DE0E0: ❹8BFC177 07C4C277 F075C477 16DEC177 BD67C477 80D3C177 607CC477 1BC2C277   008DE01C: 77DDEFB8  advapi32.dll!RegOpenKeyA
008DE100: 706FC477 00000000❺A24B1277 80481277 00000000❻F699AB7C 00000000❼10A6417E   008DE020: 77DE5196  advapi32.dll!RegEnumKeyExA
008DE120: 2398427E EA07457E 00000000❽EA821C77 5A5A1C77 8E571C77 A1601C77 F92A1C77   008DE024: 77DF9B8F  advapi32.dll!RegEnumValueA
008DE140: 52341C77 8C4D1C77 00000000❾556AAB71 ED3FAB71 E12EAB71 C145AB71 00000000   008DE028: 77DE54A4  advapi32.dll!GetUserNameA
008DE160: ❿32A5077 7E055077 00000000 00000000                                    [...]

❶ advapi32.dll     ❸ kernel32.dll     ❺ oleauth32.dll     ❼ user32.dll     ❾ ws2_32.dll
❷ crypt32.dll      ❹ msvcrt.dll       ❻ shell32.dll       ❽ wininet.dll    ❿ ole32.dll
```

Figure 5.1.: Example of an IAT for `win.asprox`. API references (thunks) are typically organized by their respective DLL.

Application Phase

After a database of potential API offsets has been created, the recovery technique can be applied to examine memory dumps for potential references to the WinAPI. Before the examination begins, the database has to be prepared for usage. Because we require our approach to have high usability, we want to optimize for speed.

Our method is based on queries against a database of addresses, so we use a data structure that allows fast lookups. While there are several possibilities, we choose a hash-map as it allows efficient lookups in average $O(1)$, which is the fastest possible in this case. For the concrete implementation that is written in Python, this means we can use dictionaries, as they are internally implemented as hash-maps. As a result, the keys of the hash-map will be absolute memory addresses and values are the respective WinAPI information (DLL and API function name) as this is what we want to resolve. For construction of the hash-map, the respective RVA export offsets are added to the preferred ImageBase address of every DLL, taking the additional ASLR offset into concern if necessary. This resulting value corresponds to what would be expected as a data reference in a live process environment, i.e. as an IAT entry.

Now using this data structure for the recovery of data references to the Windows API is straightforward. Given a memory dump, all consecutive data n-grams that correspond to a possible address length (i.e. 4 or 8 byte, resulting in `DWORD` or `QWORD` values) are extracted and then used in a look-up against the database. In case we hit an element, we consider the n-gram as a candidate for a WinAPI reference and store it along its offset in the memory dump. This method is illustrated in Figure 5.1, showing how WinAPI references are identified and extracted as candidates.

After all candidates have been identified, it seems advisable to apply optional filtering mechanisms to increase the precision of the approach as required initially by sorting out potential false positives. We propose the following methods for filtering.

F_S **Self-Filter:** We can filter out all references towards addresses that are situated within the address space of the currently analyzed memory dump. Obviously, it is impossible that a module has been mapped to the same ImageBase as inhabited by the memory that was dumped, meaning this will never result in the removal of true positives.

Name	Version/Build	All APIs	All DLLs	Unique APIs	Unique DLLs
Windows XP (32bit)	NT5.1/2600	128,408	1,597	101,710	1,584
Windows 7	NT6.1/7601	251,186	3,828	168,176	2,215
Windows 8.1	NT6.3/9600	282,802	5,154	183,424	3,024
Windows 10	NT10.0/17134	338,456	5,971	234,528	3,751
Total				323,851	5,686

Table 5.1.: Number of DLLs and exported API functions found in different vanilla installations of Windows, as presented in [13]. Unique columns show data with deduplicated DLL and API names. The "Total" is calculated as all unique values observed across all versions. Windows XP is 32bit, all others are 64bit versions.

F_N **Neighbor Filter:** We can use the distance between candidates as a criterion. This approach is based on an observation from practical malware analysis. Typically, malware authors structure their programs logically, resulting in the fact that WinAPI references are often located near to each other. This naturally holds true for the construct of an Import Address Table in which entries are stored immediately next to each other with a zero `DWORD` or `QWORD` as divider between DLLs (see Figure 5.1). However, it is our experience that this is similarly true for many custom import schemes used by malware authors. Thus, the neighbor filter discards all WinAPI reference candidates that do not have another candidate within a certain distance. This can help to reduce potential false positives, but as will be shown in Section 5.2.3, the proposed technique is highly accurate even without this filter.

Additionally, we propose the following method for estimating the number of data references to a candidate offset. It does not require any disassembly by using a similar method as for scanning for candidates. For this technique, we exploit the fact, that most data references to WinAPI references in the code will be made using instructions for which we can easily derive their targets, given the dump's base address and the offset where the reference occurs. The two instructions in question are these: `call dword ptr <offset>` and `jmp dword ptr <offset>`. Please note that a differentiation has to be made for 32bit and 64bit systems as offsets for 32bit are absolute while they are relative for 64bit systems. The binary representation of these instructions is `(48)FF15<offset>` for a call and `(48)FF25<offset>` for a jump, with the `48` prefix indicating a 64bit instruction. By simply searching through the memory dump and inspecting every occurrence of these byte sequences and the respective offset, we can perform address calculations using the given offset and analyze if it corresponds to one of the previously found candidate offsets, tracking the potential references with a counter. A limitation to this is that we do not get visibility into references using other data reference methods (e.g. indirection through `call <register>`, with the register holding a WinAPI offset). An advantage of this approach is that it does also not require any disassembly or data flow analysis, as used by both Scylla and ImpScan, while still providing decent accuracy as shown in the earlier work [13]. Obviously, non-zero counters for candidates drastically increase their chance of being an actual WinAPI reference.

5.2.2. Inventarization of the Windows API

Before we evaluate ApiScout, we first study the development of the Windows API over several versions of Windows, focusing on the most popular editions (cf. Section 4.3.2): Windows XP, 7, 8.1, and 10. This will give us insights into the population of addresses in the memory address space being potential WinAPI offsets, and thus potential for collisions while constructing the hash-map and doing lookups against it. As explained in [13], the inventorization step was applied to vanilla installations of these operating system versions to serve as a baseline for the respective OS version and ensure reproducibility.

Table 5.1 gives a summary of the results. As can be seen, the Windows API has noticably grown over the progression of the versions. The drastic difference of Windows XP to the others is explained by the fact, that only a 32bit version of Windows was available to us for XP, while for all others a 64bit version was analyzed. In 64bit Windows, a mechanism called *Windows-on-Windows 64-bit* (WOW64) is available that affects the majority of core DLLs. Because of WOW64, many DLLs exist in both 32bit and 64bit versions to ensure downward compatibility to older software compiled for 32bit systems.

Inspecting the columns with unique DLL and API names we can infer that over time not only new DLLs and APIs have been added but some older ones had also been removed. Nonetheless, the hash-map to be used by ApiScout typically consists of more than 100,000 entries, with additional entries from any software that may be installed along the Windows system DLLs. This means the chance for collisions of random DWORDs with our function offset database is very small. Using Windows XP as example, with up to 2,147,483,648 addressible bytes in user-space, only a tiny fraction (0,006%) could be occupied by valid function offsets in a scenario where all 1,584 DLLs would be mapped to memory. In practice, the number of mapped modules in processes is far smaller. Again for the Windows XP example, the most "crowded" process in a vanilla system is Windows Explorer with 138 DLLs mapped to its memory space.

As additionally discussed in [13], we investigated potential collisions of address offsets that may occur during the construction of the hash-map. For Windows XP and Windows 7, only 1 and 178 collisions have respectively been identified, which occur in DLLs sharing their base address. While a common base address rarely occurs in system libraries of these operating systems, collisions are even less likely as export offsets would have to match as well, which explains this low number of collisions.

However, the situation is completely different for Windows 8.1 and 10, where 55,181 and 115,022 collisions are respectively found. The reason for this is that ASLR is a mandatory feature in these operating system versions and all of the DLLs have been compiled with this in mind, resulting in them defaulting to standard base addresses `0x10000000` for 32bit and `0x0000000180000000` for 64bit. In practice, these theoretical collisions are even less impactful than for Windows XP and 7 and do not negatively affect the operability of ApiScout. In a running system state, ASLR offsets are initialized and will cause the DLLs to be spread in the memory space, resolving all collisions. Furthermore, as Windows employs a shared memory concept for processes, the offsets will be identical across process boundaries.

5.2.3. Evaluation

We now proceed to evaluate the capabilities of ApiScout in terms of WinAPI usage recovery by measuring the accuracy achieved with the method outlined in Section 5.2.1. We begin with explaining the data set used for the evaluation, followed by an overview of comparable methods as identified in Section 3.2, namely Scylla IAT Search and Volatility ImpScan method that will serve as a state of the art reference against which ApiScout is then evaluated in the following. For ApiScout, we also examine the influence of the proposed filters F_S and F_N, as well as the proposed method for reference counting.

Data Set

To achieve reasonable results, ApiScout has to be tested on both benign as well as malicous software. To cover all aspects mentioned previously, it should also be tested with 32bit and 64bit and activated ASLR.

We therefore use a vanilla Windows 7 32bit and 64bit system as a baseline. Because all WinAPI references found in memory have to be manually annotated to serve as ground truth, we limit ourselves to 15 benign system programs (cf. Table 5.4) chosen by personal experience. These have a large variety of characteristics and functional spectrum, e.g. including GUI, console tools, and services used for tasks such as multimedia processing, networking, or system management. We furthermore randomly select 15 malware families from the Malpedia data set (cf. Section 4.3). With regard to bitness, we select ten 32bit and five 64bit samples each.

To create a ground truth out of these binaries, we executed all of the programs and created memory dumps of their running state, which we assumed to be an idle state after one minute of execution or upon their WinAPI call to `ntdll.dll!TerminateProcess`. For the goodware, we parse all entries of the `ImportTable` and `DelayImportTable` as they designate the offsets where `ImportAddressTable` entries referencing WinAPI functions can be found. For the DelayImportTable, we then reduce the set to those references that are actually created delayed, which is typically just a subset, depending on the execution paths covered during this runtime of the program. For the malware, we similarly parse the entries from these structure but additionally perform a manual analysis to ensure that all additional (dynamic) WinAPI reference are covered as well. Dynamically resolved WinAPI references are observed for nine of 15 families: `win.citadel`, `win.geodo`, `win.h1n1`, `win.matsnu`, `win.reactorbot`, `win.redalpha`, `win.snatch_loader`, `win.trickbot`, `win.vawtrak`. In the 30 memory dumps, we identify a total of 8,132 WinAPI references across all samples.

Impact of Filtering

Before we compare ApiScout against other approaches, we examine the effectiveness of the proposed filters. As was stated earlier, the Self-Filter F_S will never lead to the removal of true positives but has the potential to filter false positives, which means it should be always active. On the other hand, the Neighbor Filter F_N can affect the result

Filter Size	4	8	16	32	64	128	256	512	1,024	2,048	4,096
TP	5,419	7,918	8,034	8,101	8,112	8,119	8,120	8,120	8,122	8,125	8,127
FP	7	20	28	44	53	60	66	68	70	71	85
FN	2,713	214	98	31	20	13	12	12	10	6	5
PPV	0.999	0.997	0.997	0.995	0.994	0.993	0.992	0.992	0.991	0.991	0.990
TPR	0.666	0.974	0.988	0.996	0.998	0.998	0.999	0.999	0.999	0.999	0.999
F1	0.799	0.985	0.992	0.995	0.996	0.996	0.995	0.995	0.995	0.995	0.994
Filter Size	8,192	16,384	32,768	65,536	131,072	262,144	524,288	1,048,576	2,097,152	4,194,304	8,388,608
TP	8,127	8,129	8,130	8,130	8,130	8,131	8,132	8,132	8,132	8,132	8,132
FP	89	98	101	116	124	126	126	126	126	126	126
FN	5	3	2	2	2	1	0	0	0	0	0
PPV	0.989	0.988	0.988	0.986	0.985	0.985	0.985	0.985	0.985	0.985	0.985
TPR	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
F1	0.994	0.994	0.994	0.993	0.992	0.992	0.992	0.992	0.992	0.992	0.992

Table 5.2.: Effect of different neighbor filter sizes without self-filter

Filter Size	4	8	16	32	64	128	256	512	1,024	2,048	4,096
TP	5,420	7,919	8,035	8,102	8,112	8,119	8,120	8,120	8,122	8,125	8,127
FP	6	6	6	6	6	6	7	8	8	8	19
FN	2,712	213	97	30	20	13	12	12	10	6	5
PPV	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.998
TPR	0.667	0.974	0.988	0.996	0.998	0.998	0.999	0.999	0.999	0.999	0.999
F1	0.800	0.986	0.994	0.998	0.998	0.999	0.999	0.999	0.999	0.999	0.999
Filter Size	8,192	16,384	32,768	65,536	131,072	262,144	524,288	1,048,576	2,097,152	4,194,304	8,388,608
TP	8,127	8,129	8,130	8,130	8,130	8,131	8,132	8,132	8,132	8,132	8,132
FP	23	32	35	50	58	60	60	60	60	60	60
FN	5	3	2	2	2	1	0	0	0	0	0
PPV	0.997	0.996	0.996	0.994	0.993	0.993	0.993	0.993	0.993	0.993	0.993
TPR	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
F1	0.998	0.998	0.998	0.997	0.996	0.996	0.996	0.996	0.996	0.996	0.996

Table 5.3.: Effect of different neighbor filter sizes with self-filter

in both ways, potentially reducing both True and false positives, in case true positives are isolated from other imports, e.g. when found in a Delay Import Table or being the result of dynamic imports.

Table 5.2 shows the results for a range of filter sizes. The sizes are chosen as powers of two, up to a size exceeding the largest memory dump in the data set, equivalent to the filter being deactivated. Table 5.3 shows the same filter sizes but with activated Self-Filter F_S.

First, it can be noted that ApiScout achieves very good results of F1 scores (i.e. the harmonic mean of precision and recall) above 0.99 for all configurations except with a F_N of size 4 and 8. This is due 64bit data references already having a size of 8 bytes, meaning that they alone technically can not fulfil the minimum distance for a filter size of 4 and even in the case of 8 require immediate neighbors, which leads to a number of missed WinAPI references.

It can be further seen that the Self-Filter F_S indeed improves all results in terms of false positives as expected. In some configurations, e.g. neighbor filter of size 2,048 bytes, the effect is quite drastic, resulting in a reduction to 8 FPs from 71 FPs without the filter.

In general, the Neighbor Filter also balances True and false positives as expected, and the best result (highest F1 with lowest FPs and FNs) is achieved for F_N with 2,048 bytes

in size. Without neighbor filtering, ApiScout is also capable of avoiding false negatives entirely, thus recovering all WinAPI references found in the test data set.

Comparison with other Approaches

In order to be comparable to ApiScout, we only consider methods that fulfil the following requirements:

- applicable to memory dumps.
- work entirely statically to reconstruct WinAPI usage.
- (publicly) available for evaluation.

As evaluation candidates, this leaves us with Scylla [114] and Volatility ImpScan [115]. It was not possible to acquire the code of Eureka as it is not publicly accessible or otherwise shared by the authors. Note that Scylla by itself does not offer a fully automated application but can be adapted to perform in that way. Therefore, we resort to only comparing ApiScout against Scylla and ImpScan. We conducted a code review for both Scylla and ImpScan and outline their methods briefly.

Scylla: IAT Search. Scylla is implemented in C++ and focuses only on IAT recovery, meaning that it aims only for extracting one block of consequent WinAPI references. This is originally done using the method `findAPIAddressInIAT` and relies on diStorm3 [270] as disassembler. The approach of `findAPIAddressInIAT` begins by disassembling 200 bytes, starting at the Original Entry Point (OEP) of the program. If it encounters a `jmp` or `call` instruction, it potentially extracts an IATPointer that is checked for being an actual WinAPI reference. In case it is a WinAPI reference, it infers the IAT start address and size by performing a backward and a forward scan for as long as additional WinAPI references are found. If no WinAPI reference is found by this approach in the first 200 bytes, another function is searched by following any call and branch instructions in the disassembled buffer. This is repeated up to 8 times.

Scylla: Advanced IAT Search. Scylla also supports a second method that improves its original scanning, called `findIATAdvanced`, which also relies on diStorm3. Instead of starting at OEP and disassembling forward, Scylla obtains the BaseAddress and size of the executable memory in which the OEP resides. Now, this whole buffer is disassembled, searching for all potential IATPointers that can be found. To derive IAT start and size from these candidates, a verification step starts in the middle of the list and checks for distances between the pointers of longer than 256 bytes. If such skips are found, it is checked if any of the two pointers creating the skip is invalid and if yes, all following candidates are deleted. The second part of the verification consists of iterated validation rounds, always starting at the beginning of the list and similarly checking neighbored candidates in the pointer list. In case a jump of more than 256 bytes is found, the entries are checked for being valid WinAPI references and deleted if this is not the case. The purpose of this procedure is to keep only valid WinAPI references and continually delete bogus references and eventually the borders of the derived IAT are returned. Note that this leaves room for false positives, as entries are only validated in case they are part of a jump.

Volatility ImpScan. ImpScan is a plugin for Volatility and written in Python2. The method is similar to the Advanced IAT Search of Scylla and works as follows. It starts by creating a database of possible WinAPI reference offsets by extracting exported functions from modules loaded into the process (similar to ApiScout). Over the full memory buffer which shall be analyzed, it disassembles all instructions and examines call instructions if they reference a call target that is listed in the database. Now, for the lowest and highest candidate, a forward/backward search called *VicinityScan* is executed, in which consecutive IAT entries of up to 0x2000 bytes distance are recorded. This search is aborted if 5 consecutive resolution errors occurred, meaning that either an invalid WinAPI reference, or a duplicate IAT entry, or a reference pointing into the own memory space is found, with the latter being identical to the Self-Filter F_S of ApiScout. As all entries are checked for being valid WinAPI references, ImpScan can only produce FPs in very borderline cases.

Because both Scylla and Volatility derive their candidates from fully scanning the memory, their candidate lists should be near identical. Yet, their filtering differs which is why we evaluate against both approaches, reimplementing Scylla's Advanced IAT Scan in Python and adapting ImpScan. For all three, we use the same WinAPI reference databases, emulating the dynamic acquistion methods used for Scylla and Volatility.

Evaluation of WinAPI Usage Information Recovery

We now discuss the evaluation of Scylla, ImpScan, and ApiScout. A summary of the results is shown in Table 5.4.

We first assess that both Scylla and ImpScan produce very similar results because of the noted relationship in their methodology, relying on linear disassembly. Both of them also have a significantly lower true positive rate (TPR) than ApiScout. Scylla is able to produce 86.9% of the TPs and ImpScan produces 87.4%. Only in three cases are these approaches capable of reconstructing all WinAPI references (ApiScout: 25/30). In two cases, both methods even do not produce a single WinAPI reference. This has two reasons. One the one hand, `win.matsnu` uses a number of anti-analysis techniques that aggravate disassembly. Furthermore, its reference offsets are not located on multiples of 4, which breaks another assumption in ImpScan. For `win.snatch` (and essentially also `win.h1n1`) the lack of identified references is a consequence of the linear disassembly using diStorm3, which is not able to identify any calls into the IAT correctly, meaning that even the initial candidate lists are empty.

The ImpScan method does not result in any false positives which is a result of the strict validation used in the vicinity scan. Scylla on the other hand produces a moderate amount of false positives but many for `winlogon` and malware family `win.citadel`. Here, the discovery of `winlogon's` Delay Imports and `win.citadel's` dynamic imports violate Scylla's assumptions of a single IAT. The FPs are thus created by misinterpreting gaps between these imports wrongly as WinAPI reference offsets without further checking. As for overall results, Scylla ends up with a F1 score of 0.893 and ImpScan achieves 0.933 due to its higher precision.

File/Malware	Malware	64bit	N_{GT}	ImpScan TP	ImpScan FN	ImpScan FP	Scylla TP	Scylla FN	Scylla FP	ApiScout TP	ApiScout FN	ApiScout FP
SoundRecorder	False	True	218	214	4	0	214	4	0	218	0	1
calc	False	True	377	373	4	0	372	5	0	376	1	1
cmd	False	False	197	184	13	0	179	18	0	196	1	0
explorer	False	True	890	855	35	0	847	43	0	890	0	1
iexplore	False	True	167	163	4	0	159	8	0	167	0	1
mmc	False	False	1,138	1,067	71	0	1,064	74	0	1,136	2	0
mspaint	False	False	929	876	53	0	874	55	1	928	1	0
notepad	False	False	201	187	14	0	187	14	0	201	0	0
nslookup	False	False	93	85	8	0	85	8	0	93	0	0
osk	False	False	191	177	14	0	177	14	0	191	0	0
regedit	False	False	315	296	19	0	296	19	0	315	0	0
svchost	False	True	102	100	2	0	100	2	1	102	0	0
taskmgr	False	False	275	262	13	0	258	17	0	275	0	0
winlogon	False	False	642	562	80	0	562	80	541	642	0	0
wscript	False	False	215	200	15	0	200	15	0	215	0	1
win.citadel	True	False	386	303	83	0	301	85	62	386	0	0
win.conficker	True	False	184	171	13	0	171	13	0	184	0	0
win.elise	True	False	104	101	3	0	101	3	0	104	0	0
win.geodo	True	False	147	95	52	0	95	52	0	147	0	0
win.globeimposter	True	False	59	58	1	0	58	1	12	59	0	0
win.h1n1	True	False	94	1	93	0	1	93	1	94	0	0
win.homefry	True	True	40	40	0	0	40	0	0	40	0	0
win.matsnu	True	False	172	0	172	0	0	172	0	172	0	0
win.reactorbot	True	True	344	142	202	0	142	202	0	344	0	0
win.redalpha	True	True	164	164	0	0	164	0	1	164	0	1
win.rockloader	True	False	34	29	5	0	29	5	0	34	0	1
win.snatch	True	False	40	0	40	0	0	40	2	40	0	0
win.trickbot	True	True	96	93	3	0	79	17	0	96	0	0
win.vawtrak	True	False	190	185	5	0	185	5	0	189	1	0
win.xagent	True	True	127	127	0	0	127	0	1	127	0	1
Totals			8,131	7,110	1,021	0	7,067	1,064	622	8,125	6	8

Table 5.4.: Results for Scylla, ImpScan and Apiscout. ApiScout uses the Self-Filter F_S and Neighbor Filter F_N with a filter size of 2,048 bytes.

ApiScout on the other hand produces very stable results and is not affected by the effects of scattered imports or imports not residing on an address that is a multiple of 4 or 8. Only in 5 cases does ApiScout miss any WinAPI references. For all of the benign programs, these are single, isolated entries of their respective Delay Import Tables that get removed by the Neighbor Filter F_N. In case of `win.vawtrak`, the false negative is also an isolated reference that results from a single cached dynamic import slightly outside the filter window. Other than that, ApiScout produces a near perfect result of F1 0.999.

A notable difference between ApiScout and the other methods is that ApiScout makes less assumptions about how the Windows API references are encountered. While this may lead to a negliable amount of false positives, it drastically improves the completeness. We therefore believe that the ApiScout method fulfils all requirements defined initially. It is a decent answer to Research Question RQ_3 and does achieve the goal of robustly statically extracting WinAPI usage information from memory dumps.

5.3. Analysis of Windows API Usage in Malware

After having presented and verified ApiScout as an effective method, we can proceed to apply it to the Malpedia corpus as described in Chapter 4. This has the goal of answering RQ_4, which targets the frequency with which malware authors apply obfuscation schemes to their WinAPI usage. We additionally investigate the occurrence frequency of different WinAPI functions in general and propose a classification scheme of semantic categories, intended to serve as an outline for the characterization of potential behaviors.

The section is structured as follows. We first review to which parts of the corpus the ApiScout methodology is applicable.

Because it has been documented on several occasions [103, 271, 109, 255, 104] that malware authors make use of obfuscation methods to conceal their program's interactions with the Windows API, we conduct an analysis to measure API information availability. We define a taxonomy of three classes of WinAPI usage with regard to obfuscation and examine how common and effective these obfuscation methods are against ApiScout.

Finally, we introduce a semantic classification scheme for WinAPI functions and based on the results of the API usage recovery, we conduct a survey on the occurrence and usage frequencies for DLL and APIs.

5.3.1. Data Set

As motivated in Section 4.3.4, we use the same repository snapshot of Malpedia (Git commit `1639cad`, created on January 3rd, 2019) to provide a consistent picture throughout this dissertation. Because ApiScout operates on memory dumps only, the 929 Windows malware families are reduced to 839, for which we have 2,352 memory dumps of samples inventorized.

At this point, it has to be noted that ApiScout is oriented towards the recovery of direct WinAPI references, which limits its applicability to native code. Among the 839 malware families for which dumps are available in Malpedia, 8 families have been created using VisualBasic and/or another 105 using the .NET framework. Both primarily do not make direct use of the Windows API but provide their own proxied API interface. In consequence, it does not make sense to apply ApiScout to these families, as the recovery method of ApiScout aims for native Windows API only. As there is no mapping available between the custom APIs used by these frameworks and the Windows API, we exclude them from the following analysis. This leaves us with 726 Windows malware families with 2.155 memory dumps present in native code, on which we will now focus in the analyses following in this chapter.

5.3.2. WinAPI Information Availability

Before we dive deeper into API usage, we first assess the general availability of WinAPI usage information. As previously mentioned, other analyses highlighted aggravation of analysis through obfuscation, e.g. O'Meara's analysis on API hashing as found in

malware family `win.heriplor` [104] or Suenaga's analysis on multiple WinAPI usage obfuscation techniques [103]. Given the importance of WinAPI information to analysts [38], concealing API interactions are certainly an aspired goal in this regard. It remains to analyze if these techniques also have an impact on the applicability and results of ApiScout.

A common approach is to avoid usage of the programmatically intended way, i.e. having the compiler specify WinAPI references in the PE header's Import Table, and instead delay the resolution of references to the WinAPI for all or just certain specific API functions until a self-chosen point in time during execution. This is typically achieved through so-called *Dynamic Imports*, often expressed through the use of two WinAPI functions from `kernel32.dll`:

1. `LoadLibrary(lpLibFileName)` to ensure that target libraries have been loaded and to identity their base addresses and
2. `GetProcAddress(hModule, lpProcName)` in order to obtain the concrete virtual address of a target API function, which is in many cases cached for later on-demand use.

While being a very popular method, more sophisticated schemes exist that even avoid using those two API helper functions [104] or otherwise complexify recognizing API interactions. For this reason, we now first define a taxonomy of API usage obfuscation schemes and then evaluate their appearance frequencies in our data set, using ApiScout and manual analysis as refinement.

A Taxonomy of API Usage Obfuscation Schemes

After extensive literature review we found only a single categorization for classes of API usage obfuscation. Sharif et al. [105] distinguish between standard API and obfuscated API resolution. Suenaga [103] gives a fine-grained enumeration of techniques observed in the wild but does not categorize them into groups.

Based on these previous works and additional manual analysis of a wide range of obfuscation schemes, we decided to use the following three classes into which API usage obfuscation can be divided.

1) Native Import Table usage (no obfuscation) As a default case, we define absence of obfuscation. In this case, the respective malware author has not conducted any steps to aggravate analysis and had their compiler generate a default Import Table. Import information for this class is usually extractable with many existing analysis tools focusing on PE header and structure analysis.

2) Dynamic Imports The classic scheme of Dynamic Importing relies on one or more dedicated functions, which are typically executed at the beginning of the program's start. These functions will resolve all desired WinAPI references that are potentially used throughout the program and store them in a "pseudo" Import Address Table format as chosen by the author. A variant of this scheme is to use Dynamic Imports only for a subset of the API functions, usually those that the author deems more suspicious or

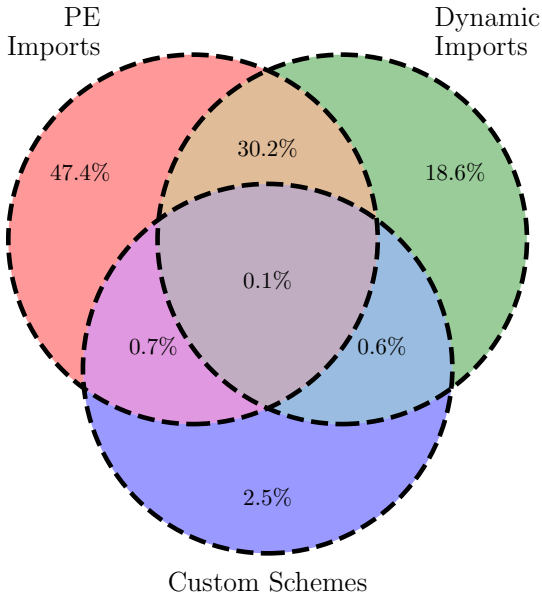


Figure 5.2.: Distribution of API Usage Obfuscation Schemes across 726 families (113 VisualBasic and .NET families excluded).

connected to potentially malicious behavior [13]. Regardless of this, their common aspect is that the direct reference offsets are cached and thus generally available to analysis.

3) Custom Schemes (obfuscation) The third class are all schemes that in some way deviate from how imports of WinAPI functions are typically used in programs, i.e. cached reference tables. These schemes are synonymous for us with obfuscation and express themselves in many unique ways as explained in the following evaluation.

Through summarizing the different custom schemes, we believe that our taxonomy better reflects the situation which analysts experience during practical work in which they are potentially confronted with obfuscation. Regardless of the actual scheme, custom methods used by malware authors always require varying amounts of tailoired efforts to effectively recover a complete WinAPI usage profile.

To summarize, our defining criterion is the ease of access for an analyst to WinAPI usage information. The division of dynamic imports and custom schemes is also motivated by the methodology of import recovery as introduced with ApiScout (cf. Section 5.2), which fully addresses both classes 1 and 2.

Evaluation of Usage Frequency for Obfuscation

After having defined three classes of usage obfuscation schemes, we now analyze how often they appear in malware. As for methodology, we apply ApiScout to all memory dumps of the 726 families considered and additionally reconstruct and parse their PE header's Import Tables (cf. Section 5.2.1).

In case all entries recovered by ApiScout are associated with an Import Address Table entry, we assign class 1 (PE Imports). If ApiScout produces results that are outside of the natural IAT, we assign class 2 (Dynamic Imports). Should ApiScout produce suspiciously few or no results, we assume class 3 (Obfuscation) and conduct manual analysis to confirm our hypothesis. Obviously, this procedure allows memory dumps to fit into multiple classes at once. A malware author may decide to use the Windows API normally for most of his code but use Dynamic Imports or Custom Schemes to protect certain WinAPI functions they deem might raise suspicion.

The observed distribution among all classes is shown in Figure 5.2. We first note that ApiScout is capable of recovering WinAPI references from almost all of the families (96.2%) as it fully covers import classes 1 and 2.

With 47.4%, almost half of all families make use of exclusively compiler-generated, native imports through the PE header. This is in line with observations made in Chapter 4, in which the analysis of PE headers indicated that many families of malicious code exhibit a "natural" program structure once unpacked. It also further strengthens the impression that aggravating analysis may not be a primary target of most malware authors and that malware often is simply supposed to enable the execution of malicious activities while otherwise having a very similar appearance to benign programs. As was noted in [12], the frequent usage of APIs tied to dynamic imports (`kernel32.dll!LoadLibrary`, `kernel32.dll!GetProcAddress`, ...) may also raise suspicions when observed in automated dynamic analysis systems. Therefore, it can definitely make sense to avoid noisy protection schemes.

A total of 18.6% families make use of Dynamic Imports only. Here, the malware authors have taken full control over how their program interacts with the Windows API. Cursory inspection of examples shows that these families regularly feature a single or a few functions that are dedicated to resolving all WinAPI dependencies at once. By definition, the resolved references are cached in some way. As the layout of the cache can be chosen freely, we note 1,231 cases where they are IAT-like (consecutively stored, sometimes missing the typical dividers of a zero DWORD/QWORD), 641 cases with two blocks, 143 with three blocks, and 75 with more than three and up to 41 blocks (each at least 2,048 bytes apart, the setting of F_N as explained in Section 5.2.3). In 82 samples from 37 families (e.g. `win.matsnu` or `win.tinba`), we furthermore observe that the offsets are also not multiples of 4 or 8.

Almost a third (30.2%) of the families make use of a mix of Native and Dynamic Imports. In this case, it is of high interest which criteria malware authors may have followed in order to treat certain WinAPI functions differently. We postpone this reasoning until Section 5.3.4 because we can additionally consider semantic contexts that are introduced in this section.

Finally, only 28 (3.9%) malware families in the data set make use of class 3 custom obfuscation schemes that are not easily recovered, some in combination with regular or dynamic imports. We were not able to identify a common pattern by which these obfuscations are designed and implemented, instead it seems malware authors typically create their own schemes around a general idea of how they want to obfuscate WinAPI usage.

	APIs	DLLs
Minimum	0	0
25%	79	5.27
50%	114	7.66
75%	169	10
Maximum	717	26
Mean	139.51	8.06
STD	106.36	3.99
Count	4,994	80

Table 5.5.: API and DLL Occurrence Frequencies per Family.

The most commonly observed scheme are variants of *API name hashing* [103, 104] which we found in at least 10 families, e.g. `win.formbook` and `win.nymaim`. Almost similarly common was the technique to create an *Extra-Modular Function Table* [103], with variants on the stack (e.g. `win.dorshel`) or heap (e.g. `win.cryptowall`). We also found cases where multiple techniques were combined, most notably in `win.andromeda`. Here, a level of indirection is created using *Staged API Obfuscation* [103] which however consists of *Immediate Jumps* [103] combined with a *Jump-in* [103], skipping the first 5 bytes or more with the most likely intention to avoid hooks (a technique also observed in `win.chthonic`).

5.3.3. DLL and API Occurrence Frequency Analysis

After a close examination of how malware performs its imports of references to the Windows API, we will now focus on how much and which parts of the Windows API are most commonly used. We again use the ApiScout results with active F_S and a F_N size of 2,048 bytes as it turned out as the best configuration in our previous evaluation in Section 5.2.3. In case we have multiple dumps for a family, we again create an average over the result values and keep those that appear in the majority of cases, using the same arguments for this procedure as used in the evaluation of the PE header data presented in Section 4.4.

Table 5.5 provides a characterization of the distribution of API and DLL counts used across the 726 families. We first note that the spectrum appears comparatively small with 4.994 APIs and 80 DLLs, given that we earlier counted about 20 times as many unique APIs and DLLs available on a vanilla Windows installation (cf. Section 5.2.2). Individually, about half of all malware families in the data set make use of between 79 and 169 API functions originating from about 5 to 10 DLLs. The most API functions (717) are used by a sample of family `win.cutlet`, a malware used for the manipulation of Automated Teller Machines (ATMs). The extensive amount of functions is a result of this malware featuring a GUI, because programming GUIs using the respective frameworks provided by the Windows API requires reliance on a variety of specialized functions for different graphical objects, thus inflating the individual count. For additional context, this GUI requires the malicious user to enter a validation PIN that they receive from the developer upon payment of a certain amount of Bitcoin and it then offers buttons to issue commands to the ATM in order to force complete dispenses of its cash containers [272].

In the following, we will now examine the frequency with which references to the individual API functions and DLLs occur. For better comparability, we summarize entries by performing three processing steps:

1. We account for the fact that WinAPI functions that process string parameters typically exist in two variants, one for ANSI and one for Unicode [273]. Treating them as one in the following can be justified by the fact that they will yield the same result (oftentimes even resorting to the same underlying WinAPI functions after conversion of the parameters) and also have joint documentation pages in the MSDN.
2. We also summarize identically named functions that are provided by different versions of the MSVC runtime, using `msvcrt.dll` as a replacement DLL label for its respective versions such as `msvcr10.dll`, `msvcr110.dll`, etc. This allows us to be more independent from concrete IDE and compiler choices of malware authors which have been already discussed in detail in Section 4.4.2.
3. In case the name appears in mangled format [274], we perform demangling and reduce the API to the primary function name, dropping additional information on types and arguments.

These adjustments reduce the total number of observed API functions from 4,994 to 4,562.

Table 5.6 gives an overview of the 50 most commonly imported APIs and DLLs. As the counts have not been adjusted for the class 3 obfuscation, we find that at least one reference to an API of `kernel32.dll` is found in every family not using API obfuscation. However, no WinAPI function is used in every of these family, as the disparity between the top entries in API and DLL columns reveals.

Looking closer at the listing of APIs, the distribution even among the first 50 entries exhibits a steeply falling occurrence frequency. The entry at position 33 already occurs in less than half of the families. The overall distribution indicates that this trend continues as shown in Figure 5.3. In fact, every API beyond `msvcrt.dll!strstr` in position 307 occurs in less than 10% of the families, meaning that 93.27% of all APIs observed occur this rarely.

What has not been accounted for is that multiple API functions may achieve the same effect intended by a malware author. For example, instead of using a higher level API offered by `kernel32.dll` a malware author may use a direct call to the respective function in the underlying `ntdll.dll`. Another example would be to achieve network capability by interacting with sockets using `ws2_32.dll` directly instead of using the high level `wininet.dll`.

The table also lists another column N_{MSVC}, indicating in how many versions of Microsoft Visual Studio (we examined all 10 versions, ranging from VS6 to 2019) the respective WinAPI function will automatically appear as standard import when using static or dynamic linking with this framework. Because of the high frequency of usage of MSVC as described in Section 4.4.2 on Linker versions as denoted by the PE header, we found adding this information a useful addition. Our available methods do not allow

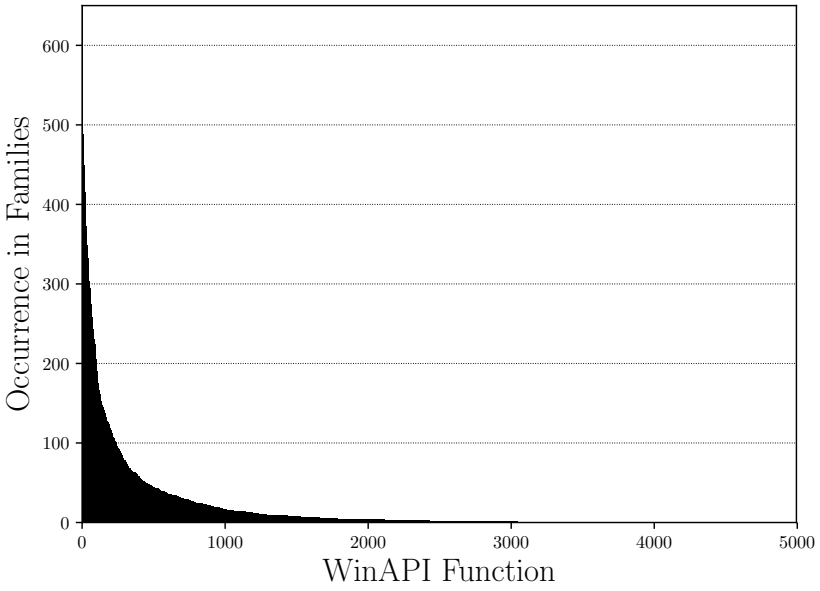


Figure 5.3.: Occurrence Frequency Distribution of individual WinAPI Functions.

us to accurately filter out the occurrences purely originating from linked code but we have to assume that they will influence the overall distribution to some degree.

In order to make API usage more comparable and also to enable faster capability assessment, we introduce a semantic classification scheme for API functions in the next section.

5.3.4. A Semantic Classification Scheme for WinAPI Functions

Having another look at Table 5.6 and focusing on the semantic aspects, we note that the vast majority of these APIs appears to be centered around core functional aspects of interacting with the Windows operating system. Many of these can be used to interact with properties relevant for execution such as obtaining and manipulating handles and modules, thread management, but also error handling. We further observe WinAPI functions that allow file system manipulation, managing memory, or reading from the Windows registry.

Further analysis of this data leads us to the conclusion that while revealing interesting data points, the presentation is not suited to get deeper insights in how the Windows API is used on a broader level. For example, we can assume that it is probably unlikely that less than half of all families will make use of network, as only a single network API function is listed with `ws2_32.dll!WSAStartup` in position 49. One reason for this is that there are typically several ways to achieve the same or similar effects when using the Windows API, as already hinted by the simultanous existence of otherwise identical API functions except for processing ANSI and Unicode arguments.

	N_{MSVC}	API	Occurrences	DLL	Occurrences
1	10	kernel32.dll!CloseHandle	654 (89.10%)	kernel32.dll	699 (95.23%)
2	5	kernel32.dll!Sleep	629 (85.69%)	ntdll.dll	652 (88.83%)
3	9	kernel32.dll!GetModuleHandle	601 (81.88%)	advapi32.dll	570 (77.66%)
4	10	kernel32.dll!WriteFile	600 (81.74%)	user32.dll	519 (70.71%)
5	10	kernel32.dll!GetModuleFileName	599 (81.61%)	shell32.dll	397 (54.09%)
6	8	kernel32.dll!CreateFile	598 (81.47%)	ws2_32.dll	368 (50.14%)
7	6	kernel32.dll!LoadLibrary	569 (77.52%)	wininet.dll	265 (36.10%)
8	10	kernel32.dll!GetProcAddress	553 (75.34%)	ole32.dll	248 (33.79%)
9	10	kernel32.dll!ExitProcess	536 (73.02%)	shlwapi.dll	235 (32.02%)
10	2	kernel32.dll!ReadFile	529 (72.07%)	oleaut32.dll	191 (26.02%)
11	10	kernel32.dll!GetCurrentProcess	528 (71.93%)	msvcrt.dll	181 (24.66%)
12	10	kernel32.dll!WideCharToMultiByte	489 (66.62%)	gdi32.dll	162 (22.07%)
13	10	kernel32.dll!MultiByteToWideChar	482 (65.67%)	crypt32.dll	97 (13.22%)
14		ntdll.dll!RtlAllocateHeap	481 (65.53%)	psapi.dll	90 (12.26%)
15	5	kernel32.dll!GetTickCount	480 (65.40%)	urlmon.dll	86 (11.72%)
16	10	kernel32.dll!TerminateProcess	471 (64.17%)	version.dll	64 (8.72%)
17	3	kernel32.dll!CreateThread	467 (63.62%)	netapi32.dll	60 (8.17%)
18	10	kernel32.dll!GetStartupInfo	449 (61.17%)	iphlpapi.dll	57 (7.77%)
19	9	kernel32.dll!GetCurrentThreadId	443 (60.35%)	winhttp.dll	54 (7.36%)
20		ntdll.dll!RtlEnterCriticalSection	442 (60.22%)	msvcp60.dll	46 (6.27%)
21		ntdll.dll!RtlLeaveCriticalSection	442 (60.22%)	mpr.dll	45 (6.13%)
22	8	kernel32.dll!GetCurrentProcessId	435 (59.26%)	dui70.dll	43 (5.86%)
23	10	kernel32.dll!GetCommandLine	434 (59.13%)	GdiPlus.dll	42 (5.72%)
24		kernel32.dll!WaitForSingleObject	430 (58.58%)	wtsapi32.dll	41 (5.59%)
25		kernel32.dll!DeleteFile	430 (58.58%)	userenv.dll	35 (4.77%)
26		advapi32.dll!RegCloseKey	417 (56.81%)	wsock32.dll	35 (4.77%)
27	5	kernel32.dll!SetFilePointer	415 (56.54%)	dnsapi.dll	34 (4.63%)
28		ntdll.dll!RtlDeleteCriticalSection	391 (53.27%)	mfc42.dll	32 (4.36%)
29	10	kernel32.dll!UnhandledExceptionFilter	389 (53.00%)	comctl32.dll	31 (4.22%)
30		ntdll.dll!RtlReAllocateHeap	385 (52.45%)	rpcrt4.dll	25 (3.41%)
31	10	kernel32.dll!GetStdHandle	373 (50.82%)	winmm.dll	24 (3.27%)
32	4	kernel32.dll!VirtualAlloc	369 (50.27%)	winspool.drv	24 (3.27%)
33		kernel32.dll!CreateProcess	363 (49.46%)	msvcr110.dll	22 (3.00%)
34	8	kernel32.dll!SetUnhandledExceptionFilter	360 (49.05%)	secur32.dll	22 (3.00%)
35		ntdll.dll!RtlGetLastWin32Error	356 (48.50%)	msvcr120.dll	19 (2.59%)
36	4	kernel32.dll!VirtualFree	355 (48.37%)	comdlg32.dll	14 (1.91%)
37	10	kernel32.dll!GetACP	353 (48.09%)	sspicli.dll	12 (1.63%)
38		advapi32.dll!RegOpenKeyEx	353 (48.09%)	uxtheme.dll	10 (1.36%)
39	9	kernel32.dll!GetProcessHeap	351 (47.82%)	imm32.dll	9 (1.23%)
40		kernel32.dll!GetFileSize	349 (47.55%)	msvcr90.dll	8 (1.09%)
41	8	kernel32.dll!TlsGetValue	347 (47.28%)	mfc42u.dll	8 (1.09%)
42	8	kernel32.dll!TlsSetValue	343 (46.73%)	powrprof.dll	7 (0.95%)
43		advapi32.dll!RegQueryValueEx	343 (46.73%)	setupapi.dll	7 (0.95%)
44	4	kernel32.dll!lstrlen	343 (46.73%)	oleacc.dll	7 (0.95%)
45	9	kernel32.dll!QueryPerformanceCounter	340 (46.32%)	imagehlp.dll	6 (0.82%)
46	10	kernel32.dll!GetCPInfo	338 (46.05%)	samcli.dll	6 (0.82%)
47	10	kernel32.dll!GetFileType	333 (45.37%)	msimg32.dll	6 (0.82%)
48	9	kernel32.dll!FreeLibrary	333 (45.37%)	netutils.dll	6 (0.82%)
49		ws2_32.dll!WSAStartup	333 (45.37%)	srvcli.dll	5 (0.68%)
50	2	kernel32.dll!GetVersionEx	332 (45.23%)	cfgmgr32.dll	5 (0.68%)

Table 5.6.: Most common APIs and DLLs across all families. N_{MSVC} indicates how many versions of MS Visual Studio's statically compiled libraries will induce presence of the given WinAPI function.

For this reason, we decide to aggregate the observed Windows API functions by their semantics. We first define a scheme and then evaluate the occurrences according to this scheme.

Definition of a Semantic Classification Scheme

In order to abstract from these concrete choices of certain API functions that may be made by malware authors, we resort to using a classification scheme of semantic categories to describe in which context these WinAPI functions are used for, centered on a (malware) programmer's viewpoint. We first considered to build on related work (cf. Section 3.2.2) as multiple authors had introduced similar schemes in previous works. Some of them even categorized more than 2,000 API functions, for example Ki [130] sorted 2,727 APIs into 26 classes and Anderson [133] assigned 2,460 APIs to 94 classes. However, we were not able to obtain any of their schemes in full detail as they were either not contained in the publication or not made publicly available by the authors. Reviewing the methodology they used, we note that their collection of APIs was extracted from dynamic traces with tagging applied to the API information, mostly for the purpose of detecting malware. Because of that we have to assume that the dynamic traces were created from packed files, which means that the classification schemes may be biased by API calls encountered primarily during the runtime of these packers and are also limited to functionality actually observable during runtime of the analysis.

As a consequence of these circumstances, we decided to create our own semantic classification scheme for Windows API functions. For the procedure, we started out by consulting the MSDN as a guideline to derive expected categories for the DLLs previously observed. The classes chosen by us generally resemble the classes used in other mentioned works, as far as this can be determined from their descriptions.

Now starting out with 11 prototype classes, we iteratively assigned all observed Windows API functions into them, marking the entries with a low/high confidence label. After the first pass, we had to make slight adjustments and extended the original scheme with an additional class covering "Other" APIs to address the API functions with a low confidence label. It turned out complicated to assign them a single semantic aspect, so we refrained from this and stayed with the "Other" group label as they can be generally considered helper constructs (like data structures or math routines) which can be used more generally than in only one semantic context.

In a second round, we then reviewed the APIs within the primary classes in order to create subgroups that capture them more accurately. This categorization was done entirely manually and primarily lead by our personal experience in reverse engineering. It involved tedious review of the results obtained through ApiScout and consulting the MSDN whenever necessary. The subgroups defined address either specific aspects of Windows system internals (e.g. COM objects, RPC, and DDE in the Process group) or categorize basic principles of data processing, such as creation, read, write, and delete operations found in the three primary classes Memory, File, and Registry.

Ultimately, we ended up with a scheme that covers all of the 4,994 APIs we observed, sorted into 12 primary and 113 secondary classes. This makes our categorization scheme

almost twice as large as the other schemes and by far the most comprehensive collection known to date. An additional advantage of our scheme over the others mentioned is that it captures all WinAPIs resolved in our entire data set, independent from whether they are observed during runtime or not because they were statically extracted.

Our scheme is publicly available and maintained through the ApiScout repository [269].

Evaluation of Occurrence Frequencies of the Semantic Classes

An overview of the classes and their frequency of occurrence is shown in Table 5.7. The presentation of summarized classes gives a much better characterization of Windows API usage across the 726 malware families than the absolute occurrence of individual API functions.

Looking at the distribution of APIs within the classes, we can see that some classes contain an extensive amount of functions, most notably GUI-related functions with 1,564 entries. The latter is in line with our earlier observation that GUI programming in Windows results in having to use many different specialized functions. We can further see that the usage of frameworks such as GDI or MFC incurs a signficant increase in functions of this semantic class that have to be covered. On the other hand, other classes such as Time (44 entries), Registry (93 entries), and Memory (119 entries) are way smaller because they do not feature the same degree of redundancy but instead a simple and tightly defined interface.

With regard to occurrences, we see that 5 classes appear in more than 90% of the families: Execution, Memory, Files, System, and handling of Strings. This is easily explained with the fact that they encompass the most basic operational aspects of information processing. Without surprise, we can also see a frequent use of writing and creating files, most certainly tied to achieving persistence oftentimes in a less suspicious file system location after initial compromise.

On the other hand, it is surprising that more than a third of the families makes use of Component Object Models (COM) objects. Usage of COM objects enables a wide range of different functionalities through convenient access interfaces. However, our personal impression is that their apparently frequent use is not covered as often or not as explicitly outlined in malware analysis reports. At this point, we do not further investigate which COM objects are actually used as it is out of scope for our analysis of Windows API usage. Instead, we consider it an interesting aspect for future work.

Even beyond those top 5 classes, we note that almost 80% of the families make use of some kind of network functionality. Inspection of the subclasses also confirms our earlier impression that aggregating by semantics provides a more meaningful picture: All three ways of using socket (`ws2_32.dll` and `winsock.dll`), Internet (`wininet.dll`), and HTTP (`winhttp.dll`, impersonating the Internet Explorer's methods) are well-known and popular for achieving connectivity to a C&C server. Also interesting is the presence of non-HTTP protocols, such as Samba (Share), Windows Terminal Services (WTS), FTP, and LDAP.

Class	APIs	Occurrences	Subclass	APIs	Occurrences	Subclass	APIs	Occurrences
Execution	601	712 (97.00%)	Process	116	695 (94.69%)	Message	27	281 (38.28%)
			Handles	25	686 (93.46%)	COM Objects	26	262 (35.69%)
			Synchronization	85	684 (93.19%)	Pipe	15	215 (29.29%)
			Modules	26	677 (92.23%)	Service	39	181 (24.66%)
			Errors	65	656 (89.37%)	RPC	42	9 (1.23%)
			Thread	68	649 (88.42%)	ActCtx	6	6 (0.82%)
			Arguments	18	546 (74.39%)	DDE	13	5 (0.68%)
			Debug	14	339 (46.19%)	Jobs	10	5 (0.68%)
Memory	119	695 (94.69%)	Allocation	24	681 (92.78%)	Management	32	543 (73.98%)
			Free	16	674 (91.83%)	Read	7	415 (56.54%)
			Write	23	574 (78.20%)	Search	17	209 (28.47%)
File	368	693 (94.41%)	Management	132	678 (92.37%)	Path	71	319 (43.46%)
			Write	26	654 (89.10%)	Volume	20	193 (26.29%)
			Create	20	619 (84.33%)	Resource	23	188 (25.61%)
			Read	23	562 (76.57%)	Open	12	119 (16.21%)
			Delete	8	448 (61.04%)	Compression	5	3 (0.41%)
			Search	13	408 (55.59%)	Link	15	1 (0.14%)
System	888	682 (92.92%)	Environment	76	663 (90.33%)	Clipboard	20	86 (11.72%)
			User	48	351 (47.82%)	Shell	26	80 (10.90%)
			Authorization	158	350 (47.68%)	Version	11	53 (7.22%)
			Console	54	329 (44.82%)	Setup	21	39 (5.31%)
			Events	33	314 (42.78%)	MSI	413	4 (0.54%)
			Control	19	280 (38.15%)	Communication	9	1 (0.14%)
String	491	681 (92.78%)	Format	144	665 (90.60%)	Convert	17	365 (49.73%)
			Search	140	556 (75.75%)	Management	4	333 (45.37%)
			Modify	89	402 (54.77%)	Uncategorized	60	197 (26.84%)
Network	390	581 (79.16%)	Sock	43	381 (51.91%)	Share	27	61 (8.31%)
			Winsock	36	376 (51.23%)	WTS	26	60 (8.17%)
			Internet	63	334 (45.50%)	User	23	54 (7.36%)
			Name	18	298 (40.60%)	FTP	11	9 (1.23%)
			HTTP	42	287 (39.10%)	LDAP	33	7 (0.95%)
			Management	62	79 (10.76%)	WLAN	6	1 (0.14%)
Time	44	550 (74.93%)	Read	30	537 (73.16%)	Control	1	2 (0.27%)
			Convert	11	250 (34.06%)	Modify	2	2 (0.27%)
Registry	93	491 (66.89%)	Management	29	468 (63.76%)	Write	22	376 (51.23%)
			Read	28	389 (53.00%)	Delete	14	194 (26.43%)
GUI	1,564	444 (60.49%)	Window	111	357 (48.64%)	Font	16	57 (7.77%)
			Management	243	292 (39.78%)	DirectUI	9	49 (6.68%)
			Dialog	48	215 (29.29%)	Desktop	13	48 (6.54%)
			Icon	19	127 (17.30%)	MFC	489	42 (5.72%)
			Mouse	7	100 (13.62%)	Scroll	22	33 (4.50%)
			Geometric	31	98 (13.35%)	MDI	3	30 (4.09%)
			Text	31	87 (11.85%)	AcctBl	9	27 (3.68%)
			Menu	50	74 (10.08%)	Caret	8	18 (2.45%)
			Image	32	68 (9.26%)	Theme	45	11 (1.50%)
			GDI	364	63 (8.58%)	OpenGL	14	1 (0.14%)
Other	134	390 (53.13%)	Variants	34	157 (21.39%)	Datastructures	24	110 (14.99%)
			Anonymous	18	148 (20.16%)	UUID	16	107 (14.58%)
			Math	39	126 (17.17%)	Compression	3	16 (2.18%)
Device	170	317 (43.19%)	Display	9	175 (23.84%)	Audio	69	53 (7.22%)
			Keyboard	52	169 (23.02%)	Network	5	46 (6.27%)
			Mouse	15	106 (14.44%)	Printer	11	28 (3.81%)
			Driver	5	61 (8.31%)	HID	3	1 (0.14%)
Crypto	132	292 (39.78%)	Encryption	60	212 (28.88%)	Random	9	102 (13.90%)
			Hash	23	126 (17.17%)	Certificate	40	42 (5.72%)

Table 5.7.: A semantic classification scheme of 12 primary and 113 secondary classes, covering 4,994 Windows APIs. Also listed are occurrences of the classes across 726 malware families.

Despite its importance and power in the general system context of Windows, the Registry is only accessed by about 66% of the malware families. Only about 51% actually write to the registry and we assume that this happens mostly in order to achieve autostart persistence.

While the GUI class is large with respect to the number of contained API functions, the occurrences are mostly tied to functions that are connected to extracting information instead of actually producing an interface. The Window subclass contains functions used to create screenshots or to read from Window titles. The latter is for example used as an alternative to process enumeration (as otherwise enabled through the System class API calls) in order to identify targets for injection or to detect if certain malware analysis software may be running.

The least frequent class Cryptography is still present in almost 40% of the families. We specifically note that the most prominent among the subclasses is actually use of encryption, which is certainly also a consequence of the rise of ransomware in the last years. A cursory review of descriptions for the malware families showed that at least 80 families in the data set can be associated with this capability.

5.4. ApiVectors: Storage and Comparison of WinAPI Usage Profiles

In the previous sections, we defined a method for the robust recovery of WinAPI information, which is essential for effective in-depth program analysis as it allows the quick localization of relevant areas in a binary. Along the way, we noted that strong obfuscation is not widely found and for the majority of malware families the recovery procedure will result in a rather complete picture of the families' API usage spectrum.

In this section, we now address RQ_5 and want to analyze how characteristic these usage spectrums are for families and if they can be used to reliably identify malware. As motivated before, the classification and identification of malware is very important as it allows to quickly assess threats and to enable the reuse of existing analysis results, allowing analysts to efficiently focus on new aspects or malware that is potentially unknown.

For this purpose, we propose a concept called ApiVectors, which doubles as an efficient storage method for recovered API information and also allows the comparison of captured API usage spectrums. The concept presented in this work is comparable to the binary WinAPI presence vectors used by Lu et al. [136] for studying applicability of machine learning to malware detection. However, apart from notably expanding the number of considered WinAPI functions, we also show in the following that using just a carefully selected subset of them can actually improve classification results.

Because ApiVectors are supposed to be used in a similar settings as ApiScout, the same requirements as explained previously in Section 5.2 can be demanded: accuracy and usability.

In the following, we first introduce the general methodology of ApiVectors. We then explain the parameters defined for this method and evaluate their effects, focusing on the trade-off between storage space and coverage. Finally, we evaluate the classification performance of the approach and compare it against two related, well-known methods: ImpHash [159] and ImpFuzzy [160].

5.4.1. Methodology

In order to investigate how characteristic API usage spectrums are and how well-suited they are to identify malware families (RQ_5), we need to define a method able to capture and compare these spectrums. We first discuss the available data and then choose a representation.

Generally, ApiScout reports contain all recovered occurrences of API references and additional meta information about them. Entries consist of the offset where the reference was found, the DLL/API name, an estimate for the number of uses for this API reference, and whether it was part of the import table or not. Obviously of highest value to us are the actual DLL/API names observed and potentially also the number of uses for an API reference. Both in combination can be considered as a direct representation of the usage spectrum.

For the other values, we find meaningful reasons why to discard them. Whether an API originates from the import table or not may provide an inconsistent picture because the parsing of the import table from memory dumps may not be possible at all or fail due to various reasons, as was observed in Section 4.4. The fact that dynamic imports are used is also rather an implementation detail that in itself can be influenced by a malware author, opposite to the need of having to use a concrete API to achieve certain functional aspects. Additionally, the offset and inferrable from that, the order of appearence for the API references should be discarded as well. These values are instable even for different compilations of the same source code, which would likely negatively impact classification results.

Related work in this domain relied on hashed representations, with ImpHash [159] using MD5 [275] over the import table entries and ImpFuzzy [160] using ssdeep [132]. ImpHash using the cryptographic hash MD5 [276] results in only being able to do exact matches, while ImpFuzzy using the rolling hash ssdeep also allows approximated matches. However, in both cases, the source information is completely lost and both approaches also do not consider using weights, a method that could take advantage of the drastically skewed distribution of API occurrences (cf. Section 5.3.3).

Given these considerations, we choose a dedicated vector representation for our features, with each dimension representing one WinAPI function. For choices of a classification approach, we have learned in Chapter 4 that there is not as much variety in unpacked malware versions opposite to observed unique packed files and acquiring ground truth beyond the quality of Malpedia is near impossible. Because even in Malpedia many families have only single representatives, we opt against an approach of (semi-)supervised machine learning because no meaningful k-fold cross validation or otherwise split into training and test data is possible and thus any evaluation would be seriously flawed.

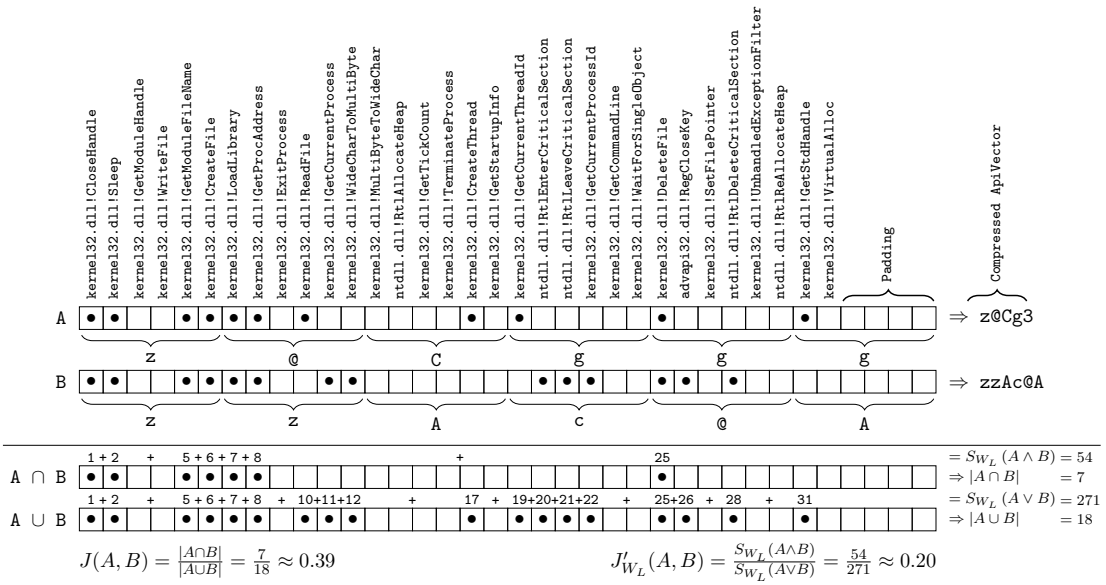


Figure 5.4.: A full example for the construction, compression, and similarity calculation of two ApiVectors A and B. The ApiVectorBase has length 32 and contains the most common Windows API functions as found in the Malpedia datatset. Compression is achieved by using Base64 with a custom alphabet (cf. Table 5.8) and applying run-length encoding for repetitive symbols. Similarity between the vectors is calculated using Jaccard similarity with optional weights for vector offsets. Diagram as initially introduced in [13].

Instead, we decided to define a similarity measure to compare individual vectors, and then use a threshold, in order to classify similarity between vectors and thus samples.

In the following, we describe the concrete construction of ApiVectors, a compression scheme for efficient storage, and the similarity measure used for the evaluation of its classification capabilities.

ApiVector Construction

We will first describe how an ApiVector is constructed. Based on the considerations discussed in the previous section, we introduce a series of abstraction and normalization steps that bring the data extracted with ApiScout into a format that can be used for further processing. We perform the same three steps of preprocessing as used for the occurrence frequency analysis (cf. Section 5.3.3). These are intended to even out implementation and compilation specifics and should increase the potential of comparisons.

First, we do not keep track of ANSI and Unicode implementations for Windows API functions. As explained in Section 5.3.3, the Windows API provides duplicate implementations that can ingest arguments given as either ANSI or Unicode. Because this is mostly an implementation specific and does not alter the semantics of a given API function, we normalize the names of respective API functions by dropping the `A` or `W` suffix.

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

Table 5.8.: The custom Base64 alphabet used for compression of ApiVectors [13].

The second step addresses the fact that many (malware) programmers make use of the Microsoft Visual Studio C compiler (cf. Section 4.4.2). Because different versions and compilation options (static vs dynamic linking) lead to use of different versions of the respective MSVCRT standard libraries, we conduct additional normalization to increase the stability of comparison results with regard to toolchain upgrades. This is easily achieved by replacing the DLL name for specific MSVCRT versions (e.g. msvcr80.dll or msvcr90.dll) with a generic name, for which we choose "mscvrt.dll".

Finally, in a third step we address name mangling that is found for some API names (mostly from MSVCRT as well). For normalization, we conduct demangling [274] and limit ourselves to using the function name only, discarding additional information such as class names or arguments. This again increases the robustness of name representations and summarizes multiple semantically similar or even identical functions into single representatives.

With regard to the 4,994 individual API functions recorded in Section 5.3.3, these normalization steps result in a reduction to 4,562 API functions. We can see that the effects are not excessive but certainly result in a meaningful generalization.

After this input normalization, the data has to be converted into the desired vector representation. As stated earlier, we want to use one dimension per API function in the vector. To be able to compare vectors, the mapping of API functions to dimensions (i.e. vector offsets) has to be consistent. For this reason, we introduce the term ApiVectorBase to describe an instance of such a mapping. We do not strictly require that every entry in a normalized ApiScout result is represented in an ApiVectorBase, e.g. an ApiVectorBase as shown in Figure 5.4 could focus on 32 entries only. In this case, entries from an ApiScout result that are not represented in the ApiVectorBase are simply discarded.

For value representation, we consider either boolean values (indicating a WinAPI function is used or not), effectively resulting in a bit vector, or integer values (API function is referenced n times). For a more convenient implementation, we restrict the usage count range to $0 - 255$, allowing us to use exactly one byte per dimension. This has only minimal impact, because only for 26 out of 298,163 observed WinAPI entries across all previously created ApiScout results exceed a reference count of 255. In these cases, we use 255 as value.

To further exemplify usage, Figure 5.4 introduces 2 example bit vectors with a ApiVectorBase of the 32 most commonly found WinAPI functions (cf. Table 5.6). The figure also explains the compression and comparison of vectors, as explained in the next two sections.

ApiVector Compression

Revisiting Figure 5.3, we note that the distribution of occurrences across families is heavily skewed towards a few very common WinAPI functions and otherwise extensively sparse as about 93% of APIs appear in 10% of the families or less. This means that it is likely possible to compress their representation to save storage space. Because we intend the ApiVector concept to be of high practical value and thus initially defined the requirement of usability, we need to consider a practical representation for the vectors just introduced.

First, we want users to be able to handle these vectors easily in their procedures of analysis and documentation. As a result we want them to be as compatible as possible and representable in ASCII printable characters only, which leaves us with 95 usable character symbols. Some symbols are used as control symbols (e.g. `"` or `|`) by terminals such as the Bourne-again shell (Bash), which implies that we should in fact use fewer than the available char set. Because of these constraints, we can not use the highly efficient Base91 (14-23% overhead) or Base85 (25% overhead) encodings and resort to the popular Base64 encoding [277] which leaves us with 33% encoding overhead.

Second, as with all meta data for files, we want them to be stored in a space-efficient manner to minimize overhead. For this, the observed fact about the sparse occurrence distribution is likely beneficial. We decide to use a run-length encoding (RLE) [278], which is compatible with our print-only requirement by introducing a straightforward modification to the standard Base64 implementation.

The concrete methodology works as follows. In order to achieve a printable representation, we first convert the given vectors (boolean or byte) into a continuous binary string. Next, we apply the Base64 encoding [277] to this binary string, adding zero bits as padding beforehand if needed. However, because we want to reserve the number literals as used by the original Base64 for the run-length encoding, we replace them with a set of the other remaining printable ASCII characters, carefully avoiding control symbols as used by shells such as the Windows commandline or Bash. The custom alphabet is shown in Table 5.8. For compression, we now scan the resulting Base64 encoded sequence of multiple consecutive appearences of the same symbol. In case we find more than 2 identical symbols, we replace them with one instance of the symbol and a number indicating the repetitions, e.g. “DDDD” becoming “D4” and so on. Figure 5.4 features two examples for this encoding and the compression.

ApiVector Similarity Measures

The final conceptual aspect to be discussed is the method of measuring similarity between ApiVectors. Similarity measures have been extensively discussed in previous works. For this work, we will consult the comparative survey of 76 binary similarity measures published by Choi et al. [279] as a guideline for the selection of an appropriate method. Thinking about what our vectors represent, we do not consider absence of WinAPIs as expressive. We therefore do not further consider negative match inclusive measures, which already eliminates 38 candidates.

The further hierarchical clustering and correlation analysis performed by Choi et al. indicates that many measures produce highly similar results. From the remaining groups, we select the Jaccard [280] similarity measure as a representative because it is extensively studied and has performed well in numerous use cases [279].

Given two Binary ApiVectors, the Jaccard similarity measure is defined as:

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

As stated earlier, we can represent A and B as bit vectors, which allows an efficient calculation through Boolean logic as follows:

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

Here, S is a function that counts the number of bits set to one in a given vector. A similar method has been successfully used by Jang et al. [189] for their BitShred approach.

Another consequence from our observation about the distribution of occurrence frequencies is that we might want to be able to use weights in our similarity measure. It is safe to assume that lesser frequently appearing WinAPI functions carry more decisive information and consequently we might want to boost them. One way to achieve this is by introducing a modification to our function S in that way that a Hadamard product (i.e. element-wise multiplication) is carried out with a second vector of the same length. Let this weight vector be denoted as W, the resulting modified function S shall now be called S_W and similarly the modified Jaccard function using S_W will be called $J'_W(A, B)$:

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

Again, a full example of this calculation is given in Figure 5.4.

After discussing the case of Binary ApiVectors, we now focus on Frequency ApiVectors. Application of Jaccard similarity is not possible because we no longer necessarily have equal values at matching vector offsets. We decided to use the continuous form of the Tanimoto [281] similarity measure:

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

The Tanimoto similarity is closely related to the Jaccard similarity [279], in fact, calculating the Tanimoto similarity for a bit vector gives an identical result to $J'(A, B)$. Again, as we want to be able to apply weights to vector entries, we extend $T(A, B)$ to become $T_W(A, B)$, but have to adjust the element-wise multiplication. Instead of applying the weights directly, we have to use the square root of each weight because the following multiplications between vectors A and B will restore the original weight values:

Size	10%	20%	30%	40%	Median	Mean
64	14.9%	19.6%	23.4%	27.0%	31.5%	33.3%
128	25.4%	33.0%	38.2%	43.8%	49.0%	50.8%
256	42.9%	50.5%	57.3%	62.9%	69.0%	67.4%
512	57.9%	71.1%	76.2%	80.5%	84.6%	80.8%
1,024	78.4%	87.0%	90.5%	92.6%	94.9%	91.1%
2,048	92.6%	96.5%	97.9%	98.9%	99.4%	97.3%
4,096	96.6%	99.4%	100.0%	100.0%	100.0%	98.8%
C-1024	67.2%	77.9%	82.6%	85.7%	89.9%	86.1%

Table 5.9.: Lowest Windows API function coverages by vector size.

$$T_W(A,B) = \frac{(\sqrt{W} \circ A) \cdot (\sqrt{W} \circ B)}{\left|\sqrt{W} \circ A\right|^2 + \left|\sqrt{W} \circ B\right|^2 - (\sqrt{W} \circ A) \cdot (\sqrt{W} \circ B)}$$

5.4.2. Evaluation of ApiVector Parameterization

In the previous section, we introduced ApiVectors as a general concept for storing and comparing ApiScout results. Before this concept can be applied effectively, we need to examine the configuration of the ApiVectorBase, especially the influence of the vector length and entry composition.

We first have to note that we do not have much information available that could aid us directly in the selection process. Our previous analysis provides us with the occurrence frequency of individual WinAPI functions and their respective semantic categories. As was concluded in Chapter 4, we can assume that Malpedia is representative for a solid part of the malware family universe. Thus, using the aggregated occurrence frequencies derived from the corpus should serve as a decent base for decision making while not adjusting too much to the individual contents of the corpus, as we are still aiming for a generalizing solution.

However, we have to take into account that the occurrence frequencies have still to be considered impure. This is primarily the consequence of the presence of additional code introduced through the different code generation frameworks used, as was already observed for the very popular Microsoft Visual Studio (cf. Table 5.6). Similarly, other frameworks with less popularity that also introduce significant amounts of statically linked library code like Delphi or Go will have an impact.

In the following two sections, we will first analyze the ability of differently composed ApiVectorBases derived from the occurrence frequency to cover ApiScout results and to capture semantically relevant WinAPIs. Next, because we claim our method to be space-efficient, we will also examine the space consumption by these different vectors, for both binary and frequency vectors, and compare them to the space consumption of other methods used for similarity analysis based on Windows API information. As data set for the evaluation, we continue to use the reference snapshot of Malpedia, as described earlier in Section 5.3.1.

	64	128	256	512	1,024	2,048	4,096	C-1024
Execution	26	43	81	129	200	318	440	229
Memory	8	16	29	38	51	83	95	68
File	12	19	39	63	98	162	225	114
System	7	18	26	51	95	188	353	150
String	5	10	13	36	78	199	300	52
Network	1	14	27	53	95	172	271	192
Time	1	2	7	10	13	24	35	22
Registry	4	5	8	13	16	36	57	32
GUI	0	1	18	85	276	574	1,267	27
Other	0	0	3	13	44	71	106	24
Device	0	0	1	8	31	82	129	66
Crypto	0	0	4	13	25	51	90	48
unknown	0	0	0	0	2	88	728	0

Table 5.10.: Different Vector sizes and their coverage of semantic categories, sorted by occurrence of classes (cf. Table 5.7)

Analysis of Coverage

We first conduct an analysis of how many ApiScout result entries are covered when using ApiVectorBases of different length. To address a meaningful range of vector lengths we revisit the average number of WinAPIs found per malware family (139 WinAPI functions) and decide to evaluate lengths of vectors that are powers of 2 between 64 and 4,096. The choice for powers of two is arbitrary but allows us to capture a good range of lengths with just 7 configurations. For WinAPI entries within these ApiVectorBases, we construct them by using a single list of the respectively most occurring WinAPI functions across all families and cut it off at the given vector length. Furthermore, we note that we do not have to conduct separate measurements for binary and frequency vectors because measuring coverage means we count one or more occurrences, which yields the same result for binary and frequency vectors.

To measure how effectively the different ApiVectorBases perform, we conduct a coverage analysis for all samples. This is done by evaluating how many of the APIs found in the malware are represented in the vectors of different lengths. We focus our analysis on the 10-40% least covered samples. The results are shown in Table 5.9. We immediately note that short vectors yield definitely an undesireably low coverage. For vectors of length 64 and 128, even the median coverage remains below 50% and for a length of 256 the median coverage reaches only about 67%. For lengths of 512 and 1,024 we definitely see significant improvement in coverage. Here, median coverage of 85% and 95% are achieved but the vector of length 1,024 also has almost 80% WinAPI entry coverage for even the ApiScout results with the lowest 10% of coverage. As expected, vector lengths of 2,048 and 4,096 give almost complete coverage. Nonetheless, we need to keep in mind that these will very likely consist almost entirely of zero entries as we only expect about 139 WinAPI function being used on average (cf. Table 5.5).

For comparison, Lu et al. [136] concluded that for their use case, a vector length of 500 entries derived from WinAPIs observed in 800 benign and 400 malicious programs yielded the best results, which is comparable with our findings.

As a second aspect, we will now have a closer look at the distribution of semantic categories being represented by the vector entries of these different sizes. Table 5.10 provides an overview of the semantic categories covered for the vector lengths chosen.

Revisiting our preceding analyses, we remember once again the highly skewed general distribution of occurrences. For the effects previously explained, purely going by occurrence frequency may not be an optimal criterium when trying to select entries that are also relevant to a human malware analyst. Then again, for the lack of available resources, it is also non-trivial to estimate the informational "value" of WinAPI functions to analysts. Especially, since different analysis tasks require different scope and thus focus on different semantic categories of the Windows API.

As a proposal for a practical solution, we manually reviewed the full list of WinAPI functions and used our personal experience to craft a custom vector we denote as `C-1024`. This vector has a length of 1,024, as this size appeared to be a favorable compromise between size and coverage. The vector `C-1024` shares 653 entries with the first 1,024 entries from the list of WinAPIs with the most occurrences. For the remaining 371 entries, we selected other WinAPI functions to provide a more nuanced spectrum of semantic categories as shown by Table 5.10.

When comparing to the regular 1,024 vector, we can see that 9 categories have been extended, while 3 have been reduced in representation. The four most occurring semantic categories Execution, Memory, File, and System have been granted more space in the ApiVectorBase. Other semantic aspects that are often of high interest to analysts have been extended as well: Network, Time, Registry, Device, and Crypto. On the other hand, we reduced entries from the categories String, GUI, and Other. Especially GUI functions that have been previously explained to be having a highly redundant API design (cf. Section 5.3.4), are now represented with only 27 instead of 276 WinAPI functions, most of them centered around initialization routines that would still indicate to an analyst the presence of GUI usage. With another look at Table 5.9, we also can see that the coverage offered by `C-1024` falls between the natural occurrence frequency vectors with sizes 512 and 1,024.

Analysis of Vector Space Consumption

After analysing the composition, we now analyze the space consumption of the different configurations. The results are shown in Table 5.11. Looking first at the maximum values, we see that the space consumption first increases proportional with the vector size but slows down as vectors grow. Opposite to that, the mean values do not grow nearly as fast and cap out starting with the vector of length 1,024. This is a direct reflection of the sparsity in vector entries that has been mentioned several times throughout this chapter.

We next look at two other methods used for similarity analysis of WinAPI usage: ImpHash and ImpFuzzy. Both methods work based on the information extracted from a given program's Import Table.

ImpHash (as the name implies) uses hashing (MD5) to convert the information into a comparable format. It maintains the order of the entries as found in the Import Table.

Vector	Min	25%	50%	75%	Max	Mean
Binary 64	4	10	12	12	12	10.2
Binary 128	4	19	23	23	23	19.0
Binary 256	4	32	41	44	44	33.9
Binary 512	4	43	64	78	87	56.0
Binary 1,024	5	48	82	115	172	79.8
Binary 2,048	5	52	92	145	306	99.9
Binary 4,096	5	53	95	155	440	108.3
Binary C-1024	5	47	77	102	166	72.4
Frequency 64	4	38	72	82	87	58.2
Frequency 128	5	58	122	144	171	100.1
Frequency 256	5	84	171	225	325	155.1
Frequency 512	5	110	213	301	584	211.4
Frequency 1,024	6	119	238	357	1,036	256.8
Frequency 2,048	6	129	252	396	1,391	287.4
Frequency 4,096	6	120	255	412	1,594	294.4
Frequency C-1024	6	115	229	330	780	232.6
ImpHash	32	32	32	32	32	32
ImpFuzzy	6	60	71	86	100	69.9

Table 5.11.: Value distributions for different ApiVector sizes as well as ImpHash and ImpFuzzy.

Because this data structure's layout is potentially randomized upon re-compilation, ImpHash may not find samples although they have an identical set of WinAPI functions used. As MD5 is used as hash, its space consumption is constant at 32 (hex)bytes.

ImpFuzzy uses fuzzy hashing to derive a representation for WinAPI information. Similar to ImpHash, it uses the Import Table information in its original sequence. However, the hashing method used here is ssdeep, which allows for approximate matches. In consequence, the storage consumption is not constant but depends on ssdeep and we observe a space consumption between 6 and 100 bytes with an average of 69.9 bytes.

When comparing ApiVectors to both other methods, we have to keep in mind that them using hashing leads to a significant loss of source information. It is impossible to reconstruct the input data, while information from ApiVectors is easily restored, minus the entries not found in the ApiVectorBase. Given that the space consumption is almost similar to that of ImpFuzzy, the only drawback is that ApiVectors may not have captured all WinAPI functions used by the input program and that some generalization is applied (normalization of API function names and DLLs).

5.4.3. Evaluation of Classification Performance

After introducing the ApiVector concept and discussing its properties of API coverage and vector sizes under different parameterizations, we now want to examine its capabilities for estimating similarity and thus its usability for malware identification. In Section 5.3.3, we noted that the distribution of occurrences for WinAPI functions across families is very scattered and that the majority of observed WinAPI functions is only found in few families. This generally implies that the individual distribution per family has a high potential for being characteristic for the respective family which should favor the WinAPI usage composition as a feature for identification.

After outlining the methodology, we first evaluate three types of parameters for ApiVectors and follow by comparing its classification performance against concepts from related work.

Methodology

For consistency, we again use the reference snapshot of Malpedia, as described in Section 5.3.1. In order to measure the classification performance, we compare all ApiVectors against each other and use a threshold to determine whether the vectors match. We note a true positive if an ApiVector from family A matches another vector of family A and a false positive if a mismatch between families occurs. Self-matches are ignored.

Revisiting the concept of ApiVectors explained in this Chapter, can identify three types of parameters whose influence should be evaluated:

1. Vector length: We noted that vector length has influence on how many APIs are captured by an ApiVector. We use the same 7 vector lengths as before, resulting from powers of 2 and giving values between 64 and 4096, as well as the manually composed configuration C-1024.
2. Binary and Frequency: Having additional information about the number of references to a certain WinAPI function may allow to better distinguish between vectors than just information about their presence. We compare boolean values and occurrence counts for WinAPI functions to see if the greater detail provided by frequency counts provides an advantage.
3. Weights: Because the presence of lesser frequent WinAPI function may help distinguish between malware families, we use three different weighting methods: equal weights, linear scaled weights, and a sigmoid function to derive weights.

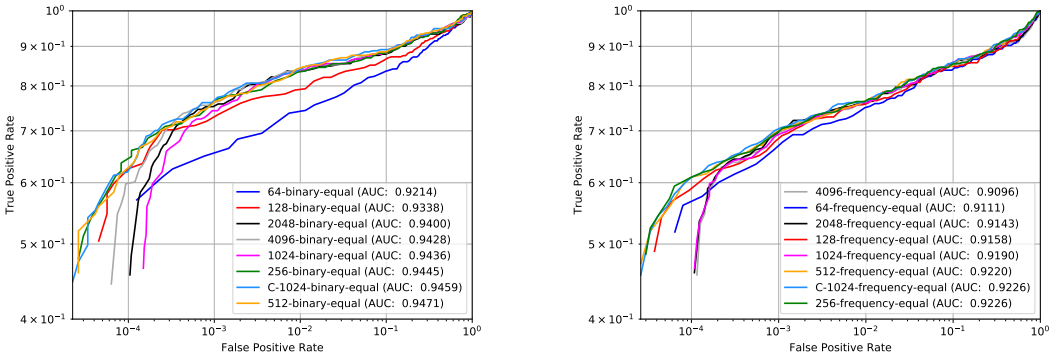
As ApiVectors do not allow further parameters, our evaluation is thus comprehensive. Finally, we use the best configuration and compare against two other related methods: ImpHash [159] and ImpFuzzy [160].

Analysis of Vector Length

We first analyze the impact of ApiVector length on classification performance because it can be considered the most significant parameter as it determines the amount of information generally available to be used when comparing vectors.

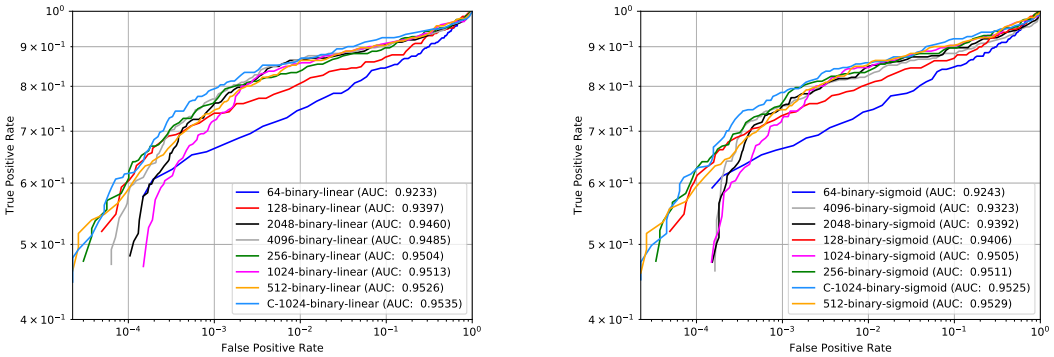
Figure 5.5a presents the results for binary ApiVectors of all considered lengths. We have chosen logarithmically scaled axis to highlight the range of smaller values which is of generally higher interest when considering application in practical context.

We first observe that ApiVectors seem generally useful as a component for the classification of malware families. All vector configurations except for the smallest two with lengths of 64 and 128 entries produce results of generally similar quality. Medium-sized vectors of 256 to 1,024 entries also slightly outperform the larger vectors with 2,048 and 4,096 entries. Notable values for true positive rate and false positive rate pairs of the crafted vector C-1024 are a TPR of 0.76 at FPR 0.001, 0.84 at FPR 0.01, and 0.89 at FPR 0.1.



(a) Binary ApiVectors (b) Frequency ApiVectors

Figure 5.5.: Comparison of ROC curves for Binary and Frequency ApiVectors with all considered vector sizes.



(a) Linear-weighted Binary ApiVectors (b) Sigmoid-weighted Binary ApiVectors

Figure 5.6.: Comparison of ROC curves for weighted Binary ApiVectors of all considered vector sizes.

Analysis of Binary and Frequency Counts

Figure 5.5b shows the results of using Frequency ApiVectors and Tanimoto distance instead (cf. Section 5.4.1). Here, the different vector lengths are even closer together than in the case of Binary ApiVectors. Again, the medium-sized vectors perform better than the others. This shows that short vectors (that notably also contain WinAPI functions with highest occurrence frequency) apparently capture not enough information to distinguish families as successful as larger vectors. Larger vectors on the other hand appear to pick up noise by capturing lesser frequent WinAPI functions that may then lead to lower matching scores. For Frequency ApiVectors in particular, a change in the number of references to a WinAPI function reduces the similarity score while it does not affect the score for Binary ApiVectors.

Analysis of Entry Weights

Apart from ApiVector length and binary/frequency contents, the method introduced in Section 5.4.1 also allows us the application of weights for entries. The general idea behind using weights is to boost the signal available from WinAPI functions with lesser occurrence frequency. These in particular are expected to be more characteristic for a given malware family when present.

For this evaluation, we consider three options for weights. First, equal weights of value 1, as used in the analysis of the other two parameterization dimensions. This is a baseline result as it values all API functions with the same degree of importance. Second, linearly increasing weights. Because our vector is sorted by occurrence frequency, we can simply use the entry position in the vector to determine the weight. This will cause API functions that are less likely encountered to have higher importance in the comparision of vectors. Third, weights determined by a non-linear sigmoid function. This allows us to express even stronger favor for entries with lower occurrence frequency. The formula used for weight deriviation is $W_S(i,n) = 50(1 + \tanh\left(3\frac{2i-n}{2n}\right))$, with n as vector length and i designating the entry's offset in the vector. These values have been chosen to remodel the *tanh* function to assume values in the range of 5 to 95.

We only consider Binary ApiVectors this time as they generally performed better than Frequency ApiVectors and the results for the weight parameterizations are shown in Figure 5.6. Comparing with Figure 5.5, we see that the introduction of weights does improve the results. While the Area under Curve (AUC) values improve only slightly, we can see that the TPR for FPR ranges of 0.01 and below do increase by several percent. This FPR range is typically considered of high practical relevance [65] which means that weights do indeed increase accuracy and usability as required for the method. As before, medium-sized vectors perform best, with smaller vectors performing slightly worse than large vectors. More importantly, we can also see that the manually chosen C-1024 vector composition dominates the other curves for almost the whole FPR range and that the best configuration is found for linear weights.

We now perform an analysis in greater detail of false positives and false negatives for this configuration to better understand the flaws of the method. For this we use a matching threshold of 0.57, which results in a TPR of 0.795 and FPR of 0.001.

For false positives, we note 176 families with a total of 271 unique wrong classifications. Similar to the previous analysis in [13], we can identify classes of false positives that explain the majority of the occurrences.

A number of families have a documented code relationship, e.g. a cluster of families derived from the leak of the Zeus source code [252]. This cluster alone involves 12 families with 30 pairs. Furthermore, we find other famous relations within the false positives, such as `win.hlux` being the known successor for `win.kelihos`, `win.evilpony` as a derivative from `win.pony`, or `win.friedex` as a ransomware spin-off from `win.dridex`. We also find a case in which a family is repurposed as a module for `win.trickbot`. In total, we explain 46 families causing 41 FPs because of these relationships.

For 74 families, the false positives are not caused because of family intrinsic code overlaps but because of their extensive use of third party and/or statically linked code

that introduces many similar WinAPI functions. Families with small intrinsic code footprint and statically linked MSVC runtime account for 44 families leading to 66 wrongly classified pairs. On top of that, we find 24 families written in Delphi alone that cause 94 FPs among each other and additionally 6 families written in Go that generate another 11 FPs.

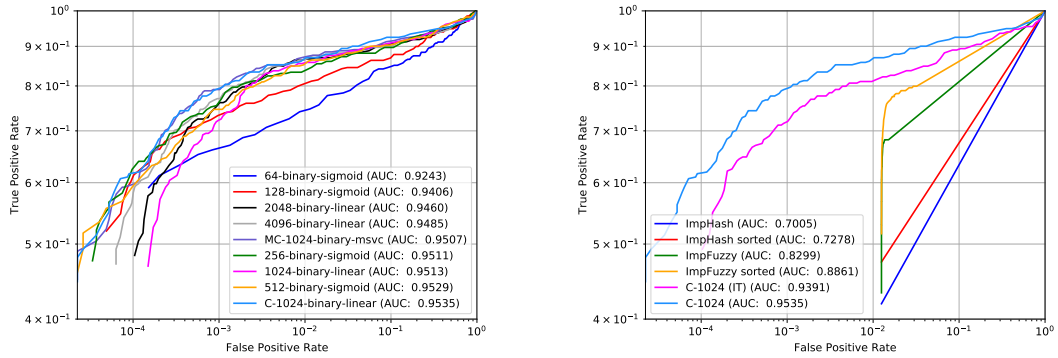
As ApiVectors group by similar WinAPI usage spectrum, we also find a few cases where (assumably) independent families are grouped because of similar behavior. Here, 10 families characterized as ransomware, 9 downloaders, 9 Remote Access Tools (RATs), and 5 families that are capable of Point of Sale (POS) manipulation create a total of 20 wrongly classified pairs among each other.

The remainder of wrong classifications are a result of multiple effects. We notice a very small number of APIs for some of the pairs, potentially due to a mix of native/dynamic imports mixed with import obfuscation, leading to presence of only a few essential WinAPI functions extractable with ApiScout. Additionally, we find some interesting pairs across all classes in which the families are attributed to the same threat actor group. For example, several malware families used by Lazarus have significant overlaps in how they use the networking aspects of the WinAPI, which is potentially a result of their intensive code reuse [282]. Similar is observed for first stage malware used by APT1 as well as noticable overlaps in WinAPI usage spectrums for those APT28 malware families compiled with MSVCRT. While we assume that static linking does have an effect on this, we still think that it is safe to assume that malware authors have a certain fingerprint in their choice of preferred APIs for certain purposes.

For false negatives, we find 68 of 331 families where no positive classification among samples was found although possible. In 29 of these cases, we identify that they are a result of the corpus structure, in which droppers and loaders, as well as modules are sometimes inventorized along the core payload. Because the grouping also happens in many cases based on third party reporting, families may include samples of varying similarity. All of these do not necessarily have a matching WinAPI usage spectrum and thus do not result in matches.

For others, the evolution within the family has lead to a spectrum whose change exceeds the threshold. That is because these families may have only spotty coverage in Malpedia, with observed samples being years apart or for example API obfuscation being introduced at a point in their development timeline. Others may have generally few imports for the same reasons as stated in the analysis of false positives, e.g. API usage obfuscation schemes.

We also analyzed the results for a variation of linear weights for C-1024. In this variation we denote as MC-1024, entries of WinAPI functions that we identified as being introduced through MS Visual Studio's statically linked code (cf. Table 5.6) receive a weight reduced to 1 instead of their normal linear weight. The result for this is shown in Figure 5.7a, which also lists the best performing parameter setting among all combinations of binary/frequency vector and weight for all vector lengths. Opposite to our expectation, this weight adjustment targeting to lessen the effect of common WinAPIs not tied to actual user defined code did not yield an improvement. It appears that this modification interferes too much with the WinAPI spectrum of



(a) Best of Class Binary ApiVectors

(b) ApiVectors, Imphash, and ImpFuzzy

Figure 5.7.: Comparison of ROC curves across all possible parameterizations and versus the related approaches ImpHash and ImpFuzzy.

families not containing the statically linked code portions. We further note that sigmoid weights perform better with smaller vectors and linear weights perform better with larger vectors.

We did not consider machine learning for further optimizing a custom weight vector. The primary reason for this is that as stated before the amount of data organized in Malpedia is insufficient for these techniques to be properly applied. In fact, a total of 758 of 839 families are represented by only five samples or less. This simply allows not to appropriately split the data into training, validation, and test/holdout sets and still yield expressive results. However, a short experiment with intentional overfitting by using an SVM classifier with the full corpus as both training and validation set resulted in a perfect classification result (i.e. every sample assigned with the correct family, no false positives). This additionally indicates that the space of WinAPI functions generally seems well divisable into classes per family and can be considered a strong feature usable in compound classifiers.

Comparison against ImpHash and ImpFuzzy

We finally compare ApiVectors against two other established methods for malware identification based on Windows API import information: ImpHash and ImpFuzzy. The results are shown in Figure 5.7b.

ImpHash (short for import hash) was created by FireEye in 2013 in order to track and cluster similar malware families. The method essentially collects all entries in a given sample's Import Table and calculates an MD5 hash over this API listing. In its original version, this API listing contained DLL and API names (if available) as well as the virtual address where the resolved import would be placed in the mapped binary (i.e. its position in the IAT), one per line. ImpHash was then incorporated into the popular Python library PEfile [21], for which the API listing was slightly modified to no longer contain the IAT position and use a comma as separator for entries. It additionally

resolved API names for a few popular Windows DLLs through lookup tables for their ordinals.

ImpFuzzy was then introduced in 2016 by JPCERT/CC as an alternative or improvement for ImpHash. It uses a mostly similar method for construction of its API listing (it incorporates additional API information in certain edge cases) but then uses ssdeep [132] for derivation of a fuzzy hash. This allows ImpFuzzy to perform approximate matching within the capabilities of ssdeep, opposite to ImpHash that only allows exact matching.

To our surprise both methods do not sort the API listing before application of the hashing. From a functional point of view, the order of DLL and API entries in the PE header's data structures can be considered irrelevant because the correct links can be easily established through code references anyway. However, this order may impact the effectiveness of the techniques just explained. We conducted a cursory analysis of Import order both in Import and Import Address tables for our data set, limiting ourselves to the most common compilers: MSVC, Borland Delphi, and MinGW (cf. Section 4.4.2). Across versions, for MSVC, the Import table appears randomized for both DLL and APIs while the Import Address Table is then apparently sorted by DLL names. For Borland Delphi, no apparent ordering can be easily identified. For MinGW, entries are consistently sorted by first DLL and then API name.

Because of these observations, we assume that sorting the API listing before hashing may positively impact the chance to produce matches for ImpHash and ImpFuzzy at the cost of some potential FPs. Consequently for our comparison, we used ImpHash and ImpFuzzy in their original form, as well as in a modified form in which the API listing contains fully sorted entries similar to MinGW, and then two variants for ApiVectors, the original C-1024 vector with linear weights applied to results provided by ApiScout and a variation that only uses the information from Import tables to ensure direct comparability with ImpHash and ImpFuzzy. We exclude empty import tables from the generation of representations for all three methods.

Analysis of Figure 5.7b reveals that ImpFuzzy does hold its promise and outperforms ImpHash, but we easily note that ApiVectors yield the best results. We also note that sorting the API listings for ImpHash and ImpFuzzy does indeed improve results significantly for both techniques. Furthermore, only ApiVectors are able to produce results with false positive rates generally lower than 1.3%.

ImpHash with its binary decision for equality seems generally too restrictive to provide generalization across different versions of the same malware family. We conclude that it is still a very good method especially for clone detection. This is commonly encountered when considering builder-based malware, in which the same binary template (also called stub) is only adjusted with a set of configuration parameters. Sorting the imports before hashing results in a net gain of 5.8%, so we recommend modifying the method in this way.

ImpFuzzy has a capability to generalize due to its use of approximative similarity but it still suffers from using a suboptimal data representation. Relying on string similarity and ssdeep in particular may lead to disadvantagous situations. For example, a variety of WinAPI functions exist as both a normal and "Ex" version (e.g. `LoadLibraryA` vs.

`LoadLibraryExA`, the latter allows passing additional parameters) or with a "Get" and "Set" variant (e.g. `GetThreadContext` vs. `SetThreadContext`, which read or modify a thread's state), which may be considered very similar by the fuzzy matching but carry different semantics that could help perform accurate matching. Additionally, since ssdeep determines block sizes depending on the input size, two API listings leading to different block sizes because of their lengths may potentially affect matching performance. ImpFuzzy suffers from the effect of library-introduced standard imports (e.g. through MSVC, Delphi, Go) as described before more than ApiVectors because it does not use weights to adjust the impact of entries. Sorting API entries before hashing also improves the technique's results by up to 7.1 percentage points.

In conclusion, we assess that ApiVectors definitely benefit from their specific data representation and ability to use weights for entries, allowing it to significantly outperform the other approaches. Generally, WinAPI information appears to be a very useful feature that, when applicable, should always be considered when determining potential similarity of programs on a cursory level.

5.5. Summary

In this chapter, we set our scope on the Windows API as it is one of the most relevant aspects to enable the situational awareness required for effective in-depth analysis of programs and malware in particular.

In the previous chapter, we identified memory dumps as a favorable form of unpacked malware (cf. Section 4.3.1). In consequence, we now defined our first research question of this chapter, RQ_3, as looking for a robst method of extraction of Windows API usage information from memory dumps. As a solution, we presented ApiScout in Section 5.2. ApiScout is a revised and generalized adaption of a method first presented by Sharif et al. [105] in the Eureka framework. Split in an inventarization and application phase, ApiScout can be applied at later stages and thus decoupled from the respective dynamic analysis environment. We continued examining the development of the Windows API from Windows XP to Windows 10 and then evaluated ApiScout's performance against two related approaches, Scylla IAT Search and Volatility ImpScan (cf. Section 5.2.3) in which ApiScout proved to be superior.

Having shown the robustness of ApiScout, we then applied it to Malpedia in order to answer RQ_4, which is concerned with how frequently malware authors actually apply obfuscation schemes to their WinAPI usage. We defined a taxonomy of three classes for WinAPI usage: Native Import Table usage (no obfuscation), Dynamic Imports, and Custom schemes (obfuscation). Across 726 malware families, we observed that Native Import Table usage can be found for almost half (47.4%) of these families, while Dynamic Imports are found more frequently in combination with Native Imports (30.2%) than on their own (18.6%). Custom schemes are surprisingly rather uncommon and were only found in 3.9% of the families. This falls in line with our assessment made in the previous chapter, that malware payloads are often programs delivered without extensive efforts to aggravate analysis.

A closer inspection of the occurrence frequencies for DLLs and APIs in malware hinted that apart from a core set of very common WinAPI functions, the majority might only appear in few families respectively, therefore potentially leading to characteristic combinations. As a result, we focused with RQ_5 on how characteristic these usage profiles for malware families actually are and if they can be used in the context of malware identification. We defined ApiVectors as a concept for storage and comparison of WinAPI usage profiles and showed that it can indeed be used for measuring malware similarity. The evaluation conducted gave general insights in the usefulness and limitations for using WinAPI usage profiles for similarity assessments. ApiVectors also significantly outperformed two other widely used methods, ImpHash and ImpFuzzy.

6. Code Recovery and Similarity Analysis

In Chapter 4, our work addressed the design of a representative corpus for malware research. Focusing on static analysis and using our reference corpus Malpedia, we then conducted a cursory structural analysis and followed in Chapter 5 with an in-depth analysis of Windows API usage. In this chapter, we continue with an examination of analysis methods for the actual code found in malware, using disassembly and code similarity to identify 3rd party library usage and code sharing among more than 650 malware families.

We first start with a motivation and define our main research questions and contributions in Section 6.1. In Section 6.2, we present our approach SMDA for robust disassembly of memory dumps and evaluate its primary heuristic for function entry point detection and the overall result quality. Next, in Section 6.3 we define a method for efficient estimation of code similarity and show that it produces accurate results. We then apply both methods together in Section 6.4 on a combined data set consisting of reference code for 3rd party libraries and Malpedia. This allows us to analyze the popularity of these third party libraries and to identify and isolate code that is intrinsic to malware families. The chapter is rounded off with a summary in Section 6.5.

6.1. Motivation and Contribution

Looking past cursory structural analysis, a clean representation of unpacked code is required to conduct effective in-depth analysis on malware [38]. With this representation as a foundation, in most cases accurate disassembly can be created, which is essentially a basis for all other techniques to build on, including reverse engineering and code-based similarity analysis.

Historically, obtaining this clean form had to be done using tedious manual or tool-assisted procedures. This was partially caused by a lack of methodology but also mainly a result of threat actors relying on packers that employed a wide array of sophisticated anti-analysis techniques.

Nowadays, the dynamic economization of the malware landscape [34] seemingly has caused a shift in how malware is protected with a stronger focus on avoiding detection. Especially the widespread use of machine learning for malware detection [65] and interconnectedness of systems has reduced the time frame in which actors can assume their campaigns to remain undetected. With techniques at hand that can process huge amounts of data to identify characteristics suitable for detection, the use of anti-analysis techniques appears to become less relevant or even less attractive, as they may stick out as suspicious. In 2011, Lindorfer et al. [283] found that about 60% of 1,686 samples

in their case study exhibited deviating behavior when executed in a virtualized environment. More recently in 2019, Ugarte-Pedrero et al. [92] dissected a representative daily intake of suspicious files as seen by a security vendor. In their study, only 9.5% of 172,000 samples did not show behavior sufficiently classifiable as malicious in a sandbox. This is in line with our observations that memory dumps serve as a good approximation of clean unpacking (cf. Chapter 4), producing a successfully unpacked result for up to 92% [245] of samples.

Sadly, disassemblers are typically not tailored for the processing of memory dumps. While they typically support processing of raw binary files as input, they will not produce results as good as on cleanly unpacked files. This is primarily because they seem to heavily rely on the structural information extractable from the meta data of an executable file. When told to ignore these data structures, it seems to cause them to use only a subset of their heuristics and analysis capabilities. However, in recent years we see an increasing reporting on fileless and memory-only malware [284], malware with custom file formats [285], or plain shell code [286].

In theory, disassemblers should not perform significantly worse when not having structural meta data from a header available. Interestingly, it has not been researched properly yet how the alternative input data representation of memory dumps impacts disassembly quality. We therefore define our first research question of this chapter as follows:

> RQ_6: How can code and Control Flow Graph information for Intel x86/x64 code be robustly recovered from memory dumps, without making further assumptions about the structural properties of the given file?

As an answer, we present our approach SMDA, a recursive disassembler that specializes on memory dumps. The approach recombines best practices known from previous works [178, 179, 172, 175, 170, 31], for which we evaluate the effectiveness of their heuristics, e.g. the ability of finding function entry points and gap functions in detail. We then evaluate the method against a selection of state of the art disassemblers, including IDA Pro and Ghidra, and show that SMDA outperforms them with regard to CFG reconstruction on memory dumps.

With the ability to robustly recover code and control flow, we now want to study how malware authors create and maintain their software. An essential tool in this context is the ability to measure similarity of code. This topic has become immensely popular and has been addressed by several works in recent years [33]. A core observation from this range of different approaches is that there is no one best solution as several provide similarly (good) results.

A primary application of these approaches is patch analysis and vulnerability research. Fewer publications focus on use cases like malware clustering and lineage, as well as library recognition which are more relevant for this thesis. Most notable is the work by Alrabaee et al. [213]. They created the approach FOSSIL, which they apply to analyze the use of about 160 FOSS projects in 17 malware families.

Given a collection of malware families as comprehensive Malpedia, we want to conduct a similar experiment and study code reuse, asking the following two research questions:

RQ_7: How frequent is third-party library usage as shared code in Windows malware?

and

RQ_8: Apart from libraries, what is the actual overlap of intrinsic code in Windows malware families?

To answer these question, we first introduce MCRIT, a method for efficient one-to-many code similarity analysis based on exact and locality-sensitive hashing. Similar to SMDA, MCRIT is based on findings of previous works but tailored specifically for RQ_7 and RQ_8: It supports tagging contents with project names (such as a malware family or a FOSS project name), versions, and a flag to specifically indicate library code. To validate the approach, we show that MCRIT performs well on a large ground truth data set composed of commonly used libraries. We then continue by creating a data set of FOSS libraries and apply MCRIT to this collection in combination with Malpedia.

Contributions. In summary, in this chapter we make the following contributions:

1. We present SMDA, a method to effectively recover code and control flow graph structure. We show that SMDA is capable of producing robust results for memory dumps, a task with which otherwise reliable disassemblers struggle.
2. We introduce MCRIT, a framework for efficient one-to-many code similarity analysis.
3. Using SMDA and MCRIT, we perform an analysis of a combined data set consisting of 2,476,837 functions from FOSS reference code as well as Malpedia, which allows us to conduct an extensive in-depth analysis of code sharing across 663 malware families and their library usage frequencies.

6.2. SMDA: Effective Code and Control Flow Recovery from Memory Dumps

Without question, accurate disassembly is an absolute foundation, upon which basically all methods of reverse code engineering build upon [167]. Andriesse et al. [30] showed that many disassemblers and especially IDA Pro, which is considered the industry standard among recursive disassemblers, achieve great results of an average 96-99% instruction recovery rate across a wide range of tested compilers and optimization levels. Their study however also revealed that all 7 disassemblers tested produced mixed results with e.g. false negative rates of 20% and above for function start detection, indicating room for improvement. This is in line with the observations of Meng and Barton [169] who closely looked at complex constructs like jump tables, non-returning functions, and tailcalls. We have not found any prior work specifically addressing memory dumps as input data for disassembly.

In this section, we present SMDA, our proposed approach for effective code and CFG recovery. As argued earlier, we specifically focus on the importance of producing robust results when processing memory dumps and with emphasis on Windows as operating

system as throughout this dissertation. For this reason, we limit ourselves in the assumptions we make about the code to be disassembled. The core objective is that SMDA should be able to achieve the same results independent from whether it is applied to unmapped PE files with full context available or shellcode without any context at all.

We define the following requirements for our method. First, it should provide a complete and accurate result for Intel x86 and x64 code and CFG recovery, taking complex constructs as mentioned in [169] and [30] into concern. We limit ourselves to this task only because we will not need more details in the following context of this work and there are established methods available in the literature [181, 180, 182], e.g. for reconstruction of further details such as all data references or function arguments. Second, while the method should generalize well, it should be especially applicable for malware analysis. In consequence, the method should focus on binaries compiled with Microsoft Visual Studio, which is highly common in Windows malware development as shown in Section 4.4.2.

In the following, we first explain the methodology behind SMDA in detail, discussing the different phases of code and CFG recovery. We then evaluate the approach, focusing specifically on the heuristic for the detection of function starts and compare the disassembly quality against other well-known disassemblers.

6.2.1. Methodology

Disassembly approaches are traditionally divided into linear and recursive methods (cf. Section 2.1.3). We will first recapitulate these basic principles and then explain how SMDA is designed.

Linear disassembly [161] is based on the assumption that functions are non-overlapping and ordered linearly one after another, which holds true almost always except in rare cases where e.g. code obfuscation is applied. For this method, the analysis is preferably started at the beginning of a code segment and then executed instruction by instruction to maximize coverage. Branches in the code flow as caused e.g. by jumps and calls are ignored as instructions are only recognized sequentially on the fly. This technique is very successful when the code is well structured, e.g. in Linux ELF binaries, it often leads to 100% accurate instruction recovery [30]. It is less applicable for Windows binaries compiled using MSVC, as this compiler makes use of inline jump tables, mixing code with data [30].

Recursive disassembly on the opposite is usually started at the Original Entry Point (OEP), and driven by the assumption that all code is supposedly reachable from this location. Similar to linear disassembly, instructions of given code sequences are processed one after another. The central difference is that targets of branches and calls are usually managed in a queue that provides further starting points of sequences, mostly basic block and function entry points (FEPs), as function starts are alternatively called. Sophisticated disassemblers often combine both methods, organized in multiple passes with application of further heuristics [287, 288, 289] in order to achieve the highest code and CFG recovery rate possible.

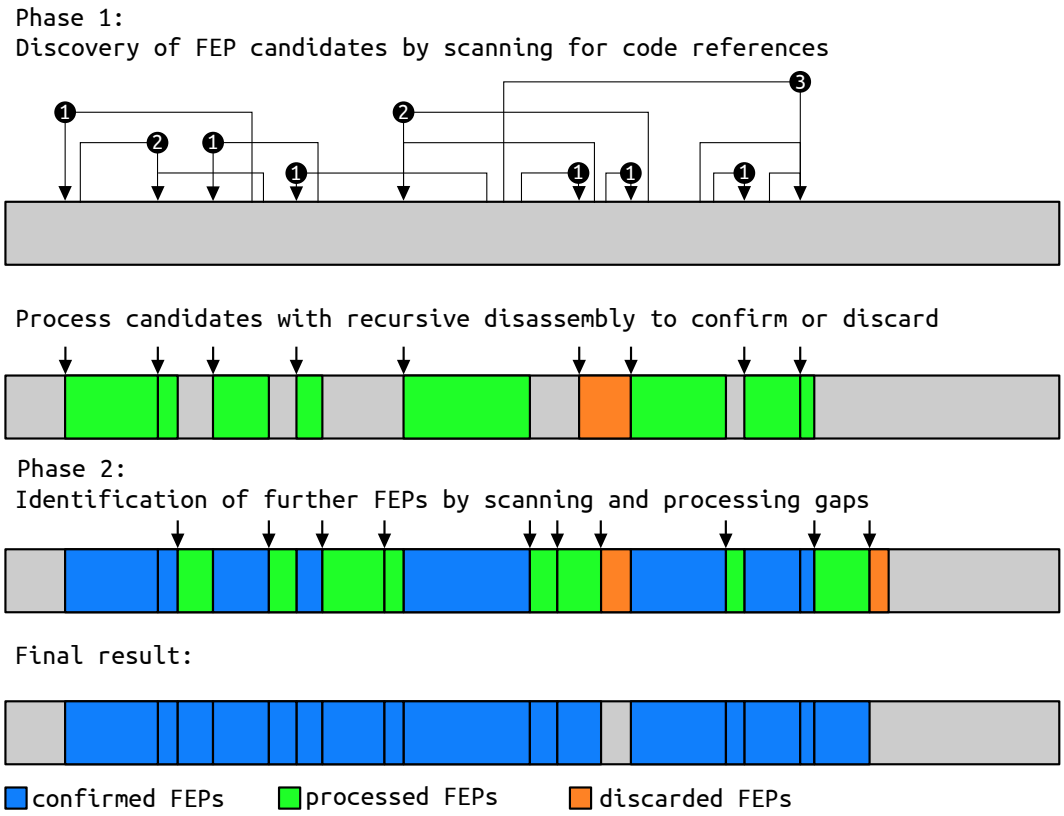


Figure 6.1.: The methodology of SMDA is structured in two phases. First, a given input buffer is scanned for code references. The resulting function entry point candidates are processed sorted by the number of references pointing to them, using recursive disassembly. In a second phase, the gaps are scanned and processed using recursive disassembly as well.

As mentioned before, reliable and complete identification of function entry points is proven as a key requirement for accurate disassembly [30, 169]. Because of this, we make optimization of FEP identification the core principle of SMDA. Our method generally incorporates ideas and approaches previously defined by Kruegel et al. [178], Harris and Miller [179], Rosenblum et al. [172], Bao et al. [175], Di Federico et al. [170], and Andriesse et al. [31].

SMDA's operation is organized in two phases:

1. Function entry point candidate discovery and reconstruction.
2. Gap function discovery and reconstruction.

Treating the input buffer agnostically, we first perform one sequential scan over all data, locating typical byte sequences that either indicate function starts or can be interpreted as jump and call instructions to derive their respective target addresses. Next, this initial list of function candidates is ordered by the number of identified references and then recursively disassembled on a per-function level. During this first round of disassembly, indirect calls via CPU registers are resolved and the queue of FEP candidates is continuously updated.

x86	x64	Opcode	Instruction
✓	✓	`E8 ?? ?? ?? ??`	call [relative offset]
✓		`FF 15 ?? ?? ?? ??`	call dword ptr [offset]
✓		`FF 25 ?? ?? ?? ??`	jmp dword ptr [offset]
	✓	`FF 15 ?? ?? ?? ??`	call qword ptr [RIP + relative offset]
	✓	`(48) FF 25 ?? ?? ?? ??`	jmp qword ptr [RIP + relative offset]

Table 6.1.: Opcodes for code references, identified through global heuristic search. *Opcode* describes the opcode bytes and operand length. *Instruction* is the corresponding instruction when disassembled.

The second phase has the goal of finding all functions that have not been identified through the processed candidates. For this, we linearly scan the buffer for a second time, treating every gap between the already identified function candidates as potential FEP candidates and again perform recursive disassembly on a per-function level.

The methodology is outlined in Figure 6.1, and both phases are now described in detail. A reference implementation of SMDA has been written in Python and is publicly available [290]. It uses capstone [291] as single instruction disassembler backend and LIEF [292] to optionally interpret ELF/PE file structure and meta data.

Function Entry Point Discovery

The core idea of our function entry point discovery is based on two observations:

Code References: The purpose of structuring a program into functions is to enable their referencing as subfunctions. As a consequence, we expect that a majority of functions is referred to by either interprocedural jumps or calls found in other parts in the code. This means in particular that those code references will point to locations within the buffer under analysis.

Common Starts: The concept of functions and calling conventions [293] implies that usually a local context has to be created to ensure that execution integrity of the calling function is not violated. As a consequence, the first instructions found in a function are typically only a subset of all possible instructions, due to necessities such as creating a stack frame or clearing registers. These instruction sequences can be used to identify potential function starts.

The x86/x64 architecture generally allows to transfer control flow beyond the immediately following instruction e.g. via the call, jump, loop, return, and interrupt instructions. In the context of our approach for FEP discovery, interprocedural call and (unconditional) jump instructions using explicit offsets are of special interest. We can derive all of their potential occurrences through scanning the input buffer and matching on the respective opcode byte sequences shown in Table 6.1. In all cases, the referenced destination can be calculated just given the match location, the instruction's operand, and the address pointed to by the operand. A slightly special case is the sequence starting with `48 FF 25`, which has a theoretically redundant `48` prefix byte (`REX.W`).

Opcodes	Instructions
`8B FF 55 8B EC`	mov edi, edi; push ebp; mov ebp, esp
`89 FF 55 8B EC`	mov edi, edi; push ebp; mov ebp, esp
`55 8B EC`	push ebp; mov ebp, esp
`55 89 E5`	push ebp; mov ebp, esp

Table 6.2.: Opcodes for function prologues identified through global heuristic search. *Opcode* describes the opcode bytes and operand length. *Instruction* is the corresponding instruction when disassembled.

The reason for its presence in some cases (especially tailcalls) is an internal Microsoft convention and it is used as a signal for unwinding [294].

For the sake of brevity, we demonstrate this calculation for a x86 relative call instruction (`E8`). Assume this instruction is found in a buffer with a given base address B, at a relative virtual address R, and with an operand specifying a relative offset O (which can be positive or negative). Then the destination D_{rc} of the call can be calculated as:

$$D_{rc} = (B + R + O + 5) \mod n$$

The addition of values is straightforward, but we have to add 5 bytes to account for the fact that the reference originates from after the call instruction and 5 bytes is the size of the call instruction.

Now, we can check if D_{rc} is located within the boundaries of the buffer under analysis (i.e. spanning from B to B + size) and if yes, treat this referenced location as a FEP candidate. The modulo operation simply ensures we perform a correct calculation with regard to the bitness' register size, with $n = 2^{32}$ for 32bit and $n = 2^{64}$ for 64bit respectively.

If D_{rc} is in fact pointing within the buffer, it is already highly likely that we found a valid call instruction and destination. This is because the operand length allows addressing 4 bytes (2^{32}) but the typical buffer size is magnitudes smaller. For example, assuming a buffer size of 1 MB (2^{20}), then only 0.024% of the potentially addressable space is occupied by this buffer.

A similar heuristic exploiting caller-callee relationships has been proposed by Rosenblum [172], but they used it to calculate likelihood values in their method based on Conditional Random Fields. Di Federico et al. [170] use caller-callee identification and return identification as well but define the concept in an architecture-independent way.

The second observation builds as well on the findings presented by Rosenblum [172] regarding idiom detection and instruction prefix trees by Bao et al. [175]. Both works generally showed that the distribution of instruction sequences found at function starts is weighted towards a limited number of highly popular constructs. Because the machine-learning based accuracy claims of Bao et al. [175] were disputed by Andriesse et al. [31], we decide to only use a very small set of instruction sequences. For SMDA, we rely on four known common prologues known for stack frame creation [295] and hotpatching [296] that we use additionally in the initial scanning procedure as shown in Table 6.2.

In case an input file is processed as regularly unmapped executable and not as a memory dump, we further use information provided by LIEF [292] to infer FEP candidates from the executable's entry point and potentially exported functions. Furthermore, candidates outside of regions marked as executable in the section table are discarded.

Given the combined list of FEP candidates resulting from both heuristics, we assign them an initial score based upon the number of references pointing to them and whether or not they start with one of the prologues defined in Table 6.2. This allows us sort the candidates by score and then perform a first round of function disassembly on them, as described in the next section.

Function Disassembly

The disassembly procedure for individual functions is based on recursive descent [297] and follows best practices and improvements presented in other works [166, 178, 170, 31]. Our disassembly model is equal to the one used in these works, which is also used by IDA Pro: every instruction may only belong to a single function and instructions may not overlap. Because recursive descent is technically a Depth-First Search (DFS), we use a stack as data structure to manage unprocessed basic block starts and initialize the stack with the respective FEP candidate.

Starting with a block, the decoding of the individual subsequent instructions is deterministic with regard to their instruction size. We therefore perform a linear sweep [166] unless we encounter instructions that potentially change the control flow. In this case, we end the block and handle the instructions specifically as described in the following. In all cases, we keep track of all visited offsets and use multiple encounters to collect code references between instructions.

Call instructions: Call instructions do not end the processing of the current basic block but are used to update the FEP candidate queue. We analyze the instruction and identify its destination. If it is a regular call using an explicit relative or absolute offset, it was by definition already handled in the FEP discovery phase. If otherwise it is a call using a register as operand, we execute a local dataflow analysis and perform backtracking to resolve its call target.

Unconditional jump instructions: These instructions have one or more outgoing edges in the CFG and end a basic block. The default case are jumps with a single target offset, which is simply added to the stack of unprocessed blocks if within the overall input buffer. The special case maps to the high-level construct of a switch statement, in assembly typically represented by the complex construct jump table. Here, the jump may dynamically calculate its target or use an address stored in a register. In this case, we use the heuristic described by Andriesse et al. [31] to resolve all potential jump targets and add them to the stack. Similarly, should the unconditional jump point to another FEP candidate or was already established as function start, we assume the complex construct tailcall and perform its resolution as described in previous works [170, 31].

Conditional jump instructions: These instructions have more than one outgoing edge in the CFG and end a basic block. In accordance with DFS, we first add the immediately following instruction to the stack, then the jump target.

Loop instructions: Similar to conditional jumps, these instructions have two outgoing edges and similarly end a basic block. We again first add their sequentially following instruction to the stack, then the loop jump target.

Terminating instructions: We treat return instructions and interrupts (e.g. `int3`) as symbols that end functions. A special heuristic is applied that checks if a `push` is used before the return. While rarely encountered in benign code or generally in the wild, it is still a well-known obfuscation construct [298] used to confuse disassemblers. Should we identify a `push-ret` construct, we try to resolve its target and treat it like a default jump in case it points into our buffer. Terminating instructions naturally also end basic blocks.

Once the stack is empty, all reachable basic blocks have been visited. The analysis of the function is concluded and the next FEP candidate can be processed. When the queue of FEP candidates is empty, we start the second phase of SMDA: Gap Analysis.

Gap Analysis

After all regular FEP candidates have been processed, the second phase is used to analyze the area not occupied by functions recovered in the first phase. The literature refers to this procedure as gap analysis and it is a concept that has been successfully applied in several works on disassembly [178, 179, 172, 31]. Functions may exist in further locations for multiple reasons, e.g. simply because references to them could not be resolved or they are actually unreferenced and were not addressed by the compiler in a dead code removal phase. Nevertheless, it is important to also locate these functions because they still carry context and may be of relevance during further analysis. We use gap analysis specifically to fulfil the requirement of completeness.

For SMDA, we only consider the space between the functions and their bodies found in the first phase because code is typically located in continuous areas and we assume further functions most likely to be found in these gaps. Should the input file be processed as unmapped, we again use LIEF [292] to limit gap search to regions marked as executable in the section table.

We basically reuse the method proposed in [179, 31], meaning that we perform a linear scan within the gaps. Whenever we encounter effective NOPs, i.e. instructions that have no effect on the program state except for increasing the program counter, we simply proceed the scan and ignore them.

The effective NOPs found depend on the compiler used. While both the Intel Software Developer's Manual [32] and AMD64 Architecture Programmer's Guide [299] define a list of preferred NOPs, additional variations do exist and are generated by compilers. In order to determine these non-standard NOPs, we used capstone directly to isolatedly disassemble all instructions found in the ground truth and collected those being disassembled as instructions containing the `nop` mnemonic. Furthermore, we included the `CC` (`int3`) byte in our skip list. While not an effective NOP, it is frequently used as padding byte by compilers to achieve 16- or 4-byte function alignment to speed up instruction

Name	Functions	TPR	FP	FN
7z	272	.992	2	2
client7z	644	.998	249	1
pageant	1,104	.989	59	12
putty	2,425	.989	66	27
sfxsetup	472	.998	16	1
x64-client7z	836	.994	0	5
x64-sfxsetup	460	.991	0	4
Combined	6,213	.991	392	52

Table 6.3.: Quality of manual labeling. *Functions* Number of functions as given in the ByteWeight data set. *TPR* is the percentage of fully correctly, manually labeled functions. *FP* are the false positives and *FN* the false negatives.

prefetch [300]. Ultimately, we ended up with a list of 41 byte sequences to be treated as effective NOPs during Gap Analysis (accessible in the SMDA GitHub repository [290]).

If we otherwise encounter an instruction different from an effective NOP, we perform another check. Because gap functions typically start only with a number of typical instructions not depending on previous context as well [175], we compare the first byte of the instruction against a list of commonly encountered start bytes. We statistically evaluated all known functions starts from the ground truth (cf. Section 6.2.2) and used only those 37 byte values that make up 99% of the cumulative distribution of all function starts. If the first instruction matches, we perform a regular Function Disassembly attempt as described in Section 6.2.1. Should this attempt be successful, we continue the gap analysis behind the newly detected function border. Otherwise, we proceed to the next gap, as we cannot reliably identify another potential FEP candidate in the current gap. This procedure is continued until all gaps have been analyzed.

Once this second phase is concluded, we have found all functions we want to extract from the given buffer. Because this recovery method of FEP-based function identification and gap analysis is relatively aggressive, it is potentially prone to false positives. However, the number of false positives is within an adequate range as shown by the following evaluation and additionally, since we value completeness over accuracy, this is acceptable for our purposes.

6.2.2. Evaluation

In this section, we evaluate SMDA with regard to its FEP detection heuristic and the completeness and accuracy of CFG recovery.

We first introduce the three data sets used. Next, we evaluate the FEP discovery heuristic based on references and prologues. Finally, we evaluate SMDA for its disassembly quality against three other tools, namely the industry standard IDA Pro, its new free competitor, Ghidra, and nucleus by Andriesse et al. [31].

Data Sets

With regard to data sets, we want to cover two aspects in this evaluation.

First, we want to achieve comparability with previous works. Thanks to the increasing emphasis on reproducibility, these data sets are easily available.

Second, we want to specifically address memory dumps and malicious software. Best to our knowledge, no related work addressed this aspect yet, which is why we created a data set of our own to allow a disassembly quality benchmark.

The data sets used in the following are in detail:

- ByteWeight as used by Bao et al. [175]
- The SPEC CPU2006, servers and glibc data set as used by Andriesse et al. [31]
- Malpedia57, a manually annotated sampling of memory dumps found in Malpedia

While the composition of ByteWeight was critized to be flawed for the evaluation of machine learning approaches because of significant code overlap in training and test data [31], we can still use it in our context as we do not rely on such a division. This data set we refer to as G_B consists in total of 68 32bit and 64bit Windows PE files, compiled with Microsoft Visual Studio in O1 and O2 optimization levels, and thus matches well what SMDA has been designed for. The data set provides both the compiled binaries and meta data files containing the ground truth about 102,532 function start and end locations but no information about instruction borders.

We additionally use the Bao data set to derive a secondary corpus ByteWeight* with the following variation: We produce memory dumps of all binaries in the same way as described in Section 4.3.2. Because the mapping for all files shifts code offsets from RVA `0x400` to `0x1000`, the ground truth is easily adapted. To additionally raise the difficulty for disassemblers, we overwrite the PE headers with null bytes, making recovery of structural information using PE header data impossible. The G_{B*} data set is otherwise identical in its content.

Andriesse et al. [30] did a thorough evaluation on the state of disassembly in 2016, using the SPEC 2006 benchmarking suite, as well as a collection of servers and glibc as a base. Commendably, they provided a full (Linux) build environment and ground truth files for the data sets used in [30, 31]. We obtained a license for SPEC 2006 in order to use it in our evaluation as well and maximize comparability. Indeed, the build environment produced Linux binaries that exactly matched the provided ground truth. We also tried to replicate the build procedure for Windows binaries, which was more complicated as no environment could be provided due to licensing issues of Microsoft Windows and Visual Studio. Not being able to match the exact Windows version and MSVC, we sadly were not able to achieve an exact reproduction of the binaries that would match the ground truth. For this reason, we had to forego the PE side of their corpus. The Linux corpus G_A consists of 304 SPEC binaries in optimization levels O0-O3, 20 server binaries and one 64bit glibc binary, altogether containing 803,602 functions with 55,698,481 instructions.

The third and final data set is Malpedia57, denoted as G_M. It is a data set that focuses on memory dumps of real-world malware samples. We used 57 randomly sampled memory dumps from Malpedia, covering 56 families. Because source code was not available for these malware samples, we had to produce ground truth in a different way. We

decided to use IDA Pro as a tool, load the memory dumps manually with no code detection heuristics enabled and then manually annotated all code found in these samples to define ground truth, producing 21,920 labels for functions with 1,335,293 instructions. Due to the tediousness of this procedure, we limited the effort spent for annotation to 80 hours, resulting in the mentioned 57 samples.

To show that our manual labeling is sufficiently accurate and matches the quality of the other ground truth, we also manually labeled a selection of binaries from the ByteWeight data set, which was close due to also being PE files. For methodology, we decided to only count functions as correct for which we labeled both their function start and all instructions inferred by function end in accordance with the ground truth. The results are shown in Table 6.3. As can be seen, with our manual annotation we achieved a high TPR of 99.16% and a F1 score of 0.965 overall. The majority of false negatives in our manual labeling were unreferenced nullsubs (i.e. single `C3` (`retn`) instructions), which provide negliable value to analysis. The high number of false positives for client7z actually reveals a mismatch of information generated by the compiler, as most of the manually identified functions are actually referenced by other code but seemingly did not show up in the ByteWeight ground truth (which was inferred from PDB files, according to the paper).

Accuracy of Function Entry Point Discovery

Before we evaluate SMDA's ability to produce accurate disassembly itself, we first examine the heuristics used for function entry point discovery as used in the first phase of operation. For this, we separately analyze the reference counting and prologue detection.

Reference Counting. Table 6.4 summarizes the results. No ground truth for the number of control flow references to function starts is available for any data set. Because of this, we resort to the evaluation of correctness additionally considering the number of references identified and function starts defined by SMDA expressed in true and false positives and precision. The results are based on all FEP discovery methods available to SMDA, prologue identification, the code reference heuristic, and gap search. A refcount above zero indicates that the FEP was located using the code reference heuristic, while a refcount of zero means the FEP was found through gap search.

The presence of a prologue as defined in Table 6.2 is generally decoupled from the occurrence of references. But as discussed in more detail in the following, they very frequently occur in parallel.

Depending on the data set, between 20.19% and 45.08% of the function starts identified by SMDA had no related code references and were only recovered through the Gap Analysis phase. Reading the results the opposite way, this means that between 54.92% and 79.81% of function starts can be identified through the FEP heuristic presented in Section 6.2.1. This supports the assumption stated earlier and shows that a decent number of function starts can be identified prior to any disassembly by simply scanning the buffer once and correlating the results. Looking at the reliability of the heuristic across all data sets, FEP candidates identified through more than one reference are

	Bao				Andriesse			
refcount	TPs	FPs	PPV	Occurrence	TPs	FPs	PPV	Occurrence
0	96,589	6,944	.933	45.08%	357,642	122,734	.744	40.67%
1	39,481	2,980	.930	18.43%	229,251	591	.997	26.07%
2	25,459	206	.992	11.88%	98,012	109	.999	11.14%
3	13,650	58	.996	6.37%	47,017	42	.999	5.35%
4	8,141	64	.992	3.80%	31,074	25	.999	3.53%
5-8	13,594	55	.996	6.34%	50,205	30	.999	5.71%
9-16	8,481	31	.996	3.96%	28,652	11	1.00	3.26%
17-32	4,260	16	.996	1.99%	16,625	2	1.00	1.89%
33-64	2,344	0	1.00	1.09%	9,964	1	1.00	1.13%
65-128	1,181	0	1.00	0.55%	5,543	2	1.00	0.63%
129-256	625	1	.998	0.29%	2,926	0	1.00	0.33%
257-512	298	0	1.00	0.14%	1,440	0	1.00	0.16%
513-1,024	125	0	1.00	0.06%	713	0	1.00	0.08%
1,025-2,048	27	0	1.00	0.01%	319	0	1.00	0.04%
	Malpedia57				All			
refcount	TPs	FPs	PPV	Occurrence	TPs	FPs	PPV	Occurrence
0	4,332	1,183	.786	20.19%	458,563	130,861	.778	41.12%
1	8,063	749	.915	37.58%	276,795	4,320	.984	24.82%
2	3,490	191	.948	16.27%	126,961	506	.996	11.38%
3	1,577	78	.953	7.35%	62,244	178	.997	5.58%
4	915	44	.954	4.27%	40,130	133	.997	3.60%
5-8	1,482	85	.946	6.91%	65,281	170	.997	5.85%
9-16	822	43	.950	3.83%	37,955	85	.998	3.40%
17-32	438	29	.938	2.04%	21,323	47	.998	1.91%
33-64	184	16	.920	0.86%	12,492	17	.999	1.12%
65-128	96	8	.923	0.45%	6,820	10	.999	0.61%
129-256	40	3	.930	0.19%	3,591	4	.999	0.32%
257-512	11	1	.917	0.05%	1,749	1	.999	0.16%
513-1,024	3	1	.750	0.01%	841	1	.999	0.08%
1,025-2,048	-	-	-	-	346	0	1.00	0.03%

Table 6.4.: Reliability of function entry point detection (prologue, code reference, and gap search combined) provided by SMDA for all three data sets and combined. TPs: true positives, FPs: false positives, PPV: Precision, Occurrence: fraction of identified function starts with the given number of references (refcount).

almost always correct as the precision of 0.996 indicates and even those with a single reference are correct in 98.4% of cases. We manually inspected a subset of the cases where more than one reference lead to a false positives and found that they were almost entirely connected to SMDA's tailcall detection and the decision taken where to define additional function starts in these cases.

With respect to the different data sets, the heuristic achieves the highest precision on the Andriesse data set which consists solely of Linux binaries. Here, even single references are indicators for actual function starts 99.7% of the time. On the other side, the Gap Analysis performed least precise on this data set, which is related to the extensive use and structural specifities of exception handler constructs in the code base, which lead to a relatively high number of false positives.

For the ByteWeight and Malpedia57 data sets, the precision for single reference function indications is at 93.0% and 91.5% respectively. The heuristic seems generally to perform worst on the malware data set, where also higher reference counts apparently lead to false positives.

The reason for this is tied to two specific families: `win.bolek`, `win.corebot`. In `win.bolek`, we find use of the Heaven's Gate technique [253, 254] for switching between 32bit and 64bit execution mode as noted earlier in Section 4.4.2 as well as a range of functions in 64bit Intel assembler. In the ground truth, the 64bit code has been excluded, but as the specific code areas in themselves also use code references, they are mistakenly picked up by the heuristic and lead to potential false positives. The most false positives originate however from `win.corebot`, where a full 64bit binary is contained in a data section called "x64". It is in fact a 64bit version of the CoreBot, which is deployed in case a 64bit version of Windows is detected. Naturally, this second PE file contains numerous relative code references, which could not be discarded during analysis because SMDA (similarly to all other disassemblers) does not expect overlayed, unmapped files in a data section. Expectedly, this also affects Gap Analysis. Both samples together cause 492 of 499 FPs with two or more references, which would otherwise put the Malpedia57 results in line with the other data sets.

Overall, the FEP detection heuristic based on reference counting has proven to be a very reliable instrument for locating function entry points. Considering that Malpedia57 consists entirely of memory dumps, this shows that the heuristic can even be applied without making assumptions about where to expect code or data, making it suitable for dealing with shellcode.

Prologue Detection. Using the four byte sequences defined as prologues in Section 6.2.1, we assess that in total 252,070 of 1,115,209 functions (22.60%) recognized by SMDA have one of these prologues, out of which 58,596 are not also identified by a reference and thus a gap function. The heuristic produces only 254 false positives (overall PPV: 0.999), all of which occur in the PE files and none in the ELF files of the Andriesse data set. Interestingly, all of the positively identified gap functions are in the Andriesse data set. As a result, this means that no function with one of these prologues in the Bao and Malpedia57 data sets do not also have a code reference.

We also used YARA to scan for all 4 of the prologue byte sequences in the data sets, which revealed another peculiarity with regard to which of these sequences appear in which data sets. Both variations of the sequence `mov edi, edi; push ebp; mov ebp, esp` (byte sequences `8B FF 55 8B EC` and `89 FF 55 8B EC`) occur only in the PE data sets. This is most likely a result of `mov edi, edi` being intended only as a placeholder to enable hot patching [296] as explicitly introduced by Microsoft [301].

The instruction sequence `push ebp; mov ebp, esp` on the other hand does have an ambiguous binary representation and can be produced by multiple byte sequences because the direction bit in the opcode field and swapping register modifiers yields two semantically equivalent sequences.

The variation `55 8B EC` seems to be preferred by the C/C++ compiler in Visual Studio, as it appears dominantly in the Bao and Malpedia57 data sets. A respective

YARA rule consisting only of these three bytes produces just 14 hits in the entirety of the Andriesse data set and all of them have been confirmed to not be instructions (but coincidentially data fragments).

The `55 89 E5` variation similarly does not at all appear in the Bao data set but in 5 samples of the Malpedia57 data set:

- in `win.dyre`, it appears seven times in code fragments distant from the main code. We assume this is a set of functions manually written in assembler as they contain a specific construct or rather trick related to a call instruction and string usage that is very unlikely to be generated by any compiler.
- in `win.locky` and `win.bedep`, it appears once each, again in functions that based on their register usage might originate from manually written assembly.
- in `win.corebot`, the byte sequence appears twice as part of Heaven's Gate, whose implementation is written in inline-assembly.
- in `win.pony`, it appears as part of the 3rd party library aPLib, whose source code is written in assembler.
- in `win.tinba` and `win.matsnu`, the sequence fully replaces occurrences of `55 8B EC`. The commonality of these families is that their source code is written in assembler, as both the linker field in the PE header and certain idioms in their code indicate.

Overall, the prologue detection greatly benefits the location of FEP candidates for ELF files and serves as a good supporting feature for PE files.

Evaluation of Disassembly Quality

Before we present and discuss the evaluation results for disassembly quality, we have to clarify one important aspect with regard to how a specific code entity is treated with respect to the ground truth. This code entity are stubs for referring to external functions, which exist for both ELF and PE files and may depend on compiler settings. Various disassemblers have different strategies for whether or not recognizing these stubs as code, which can significantly impact their results (by influencing the count of true and false positive function starts), especially with larger stub collections in small programs. Additionally, the disassemblers are used in batch mode, meaning their execution is automated and no further manual choice of settings is taken.

In our evaluation we only consider four disassemblers, the two strongest from Andriesse's evaluation (IDA Pro and nucleus), we add Ghidra (version: 9.1.2) as a new contender and SMDA (version: 1.2.5) as introduced in this dissertation.

Out of these four disassemblers (Ghidra, IDA Pro, nucleus, and SMDA), all except nucleus consider the jmp thunks potentially found in an ELF files' `PLT` section as functions and thus code. All four disassemblers consider jmp thunks encountered in PE files, when compiled with Visual Studio and incremental linking enabled [302] as functions and thus code.

				Ghidra 9.1.2		IDA Pro 6.7		IDA Pro 7.4		nucleus		smda 1.2.5	
GT	pack	compiler	OPT	TPR	PPV	TPR	PPV	TPR	PPV	TPR	PPV	TPR	PPV
G_A	glibc	gcc510-64	-	.001	.001	-	-	**.967**	**.990**	.921	.691	.944	.875
G_A	servers	gcc510-32	-	**.996**	**1.00**	.871	.917	.967	.998	.983	.979	.981	.973
G_A	servers	gcc510-64	-	**.992**	**1.00**	.856	.901	.959	1.00	.979	.983	.982	.962
G_A	servers	llvm370-32	-	.766	1.00	.840	.919	.867	1.00	**.985**	**.993**	.985	.958
G_A	servers	llvm370-64	-	**.992**	**1.00**	.826	.897	.942	1.00	.970	.991	.962	.959
G_A	SPEC (C)	gcc510-32	O0	**.994**	**1.00**	.800	.999	**.994**	**1.00**	.991	1.00	.928	.971
G_A	SPEC (C)	gcc510-32	O1	**.988**	**1.00**	.770	.999	.931	.995	.982	.987	.992	.992
G_A	SPEC (C)	gcc510-32	O2	**.987**	**1.00**	.711	.997	.901	.997	.913	.941	.862	.981
G_A	SPEC (C)	gcc510-32	O3	**.987**	**1.00**	.663	.998	.891	.995	.927	.945	.854	.982
G_A	SPEC (C)	gcc510-64	O0	.991	1.00	.800	.999	**.994**	**1.00**	.991	.999	.981	.913
G_A	SPEC (C)	gcc510-64	O1	**.987**	**1.00**	.764	.992	.919	1.00	.982	.979	.991	.991
G_A	SPEC (C)	gcc510-64	O2	**.984**	**.998**	.716	.991	.826	1.00	.926	.969	.872	.989
G_A	SPEC (C)	gcc510-64	O3	**.986**	**.998**	.665	.984	.799	1.00	.936	.971	.869	.983
G_A	SPEC (C)	llvm370-32	O0	**.985**	**1.00**	.803	.998	.820	1.00	.991	1.00	.981	.899
G_A	SPEC (C)	llvm370-32	O1	.697	1.00	.784	.998	.802	1.00	.967	.982	**.967**	**.995**
G_A	SPEC (C)	llvm370-32	O2	.624	1.00	.689	.998	.724	1.00	.973	.983	**.969**	**.994**
G_A	SPEC (C)	llvm370-32	O3	.616	1.00	.677	.998	.716	1.00	.974	.985	**.968**	**.994**
G_A	SPEC (C)	llvm370-64	O0	.991	1.00	.802	.978	**.994**	**1.00**	.992	1.00	.977	.923
G_A	SPEC (C)	llvm370-64	O1	**.983**	**.997**	.773	.997	.897	1.00	.952	.975	.862	.995
G_A	SPEC (C)	llvm370-64	O2	**.987**	**.997**	.683	.996	.844	1.00	.963	.975	.848	.992
G_A	SPEC (C)	llvm370-64	O3	**.987**	**.997**	.669	.996	.843	1.00	.963	.976	.846	.990
G_A	SPEC (C++)	gcc510-32	O0	**.997**	**1.00**	.928	.981	.996	1.00	.987	1.00	.992	.961
G_A	SPEC (C++)	gcc510-32	O1	**.995**	**1.00**	.847	.975	.944	.996	.974	.995	.988	.969
G_A	SPEC (C++)	gcc510-32	O2	**.995**	**1.00**	.806	.965	.917	.973	.929	.915	.803	.919
G_A	SPEC (C++)	gcc510-32	O3	**.995**	**1.00**	.757	.962	.906	.970	.941	.912	.794	.874
G_A	SPEC (C++)	gcc510-64	O0	**.997**	**1.00**	.927	.980	.996	1.00	.987	.998	.987	.911
G_A	SPEC (C++)	gcc510-64	O1	**.995**	**1.00**	.783	.969	.923	1.00	.972	.988	.986	.961
G_A	SPEC (C++)	gcc510-64	O2	**.995**	**.999**	.670	.956	.874	1.00	.923	.916	.773	.926
G_A	SPEC (C++)	gcc510-64	O3	**.995**	**.999**	.622	.951	.846	1.00	.943	.915	.759	.867
G_A	SPEC (C++)	llvm370-32	O0	**.997**	**1.00**	.895	.978	.994	1.00	.989	.906	.956	.862
G_A	SPEC (C++)	llvm370-32	O1	**.990**	**1.00**	.872	.979	.948	1.00	.941	.925	.952	.938
G_A	SPEC (C++)	llvm370-32	O2	**.985**	**1.00**	.762	.967	.890	1.00	.968	.883	.862	.908
G_A	SPEC (C++)	llvm370-32	O3	**.982**	**1.00**	.748	.967	.883	1.00	.969	.883	.841	.908
G_A	SPEC (C++)	llvm370-64	O0	**.998**	**1.00**	.893	.962	.996	1.00	.988	.905	.956	.857
G_A	SPEC (C++)	llvm370-64	O1	**.996**	**1.00**	.825	.977	.913	1.00	.882	.914	.846	.942
G_A	SPEC (C++)	llvm370-64	O2	**.994**	**.999**	.748	.964	.859	1.00	.939	.873	.775	.916
G_A	SPEC (C++)	llvm370-64	O3	**.994**	**.999**	.738	.963	.859	1.00	.942	.874	.760	.919
G_B	ByteWeight	msvc10-32	O1	.804	.952	-	-	.835	.996	.975	.923	**.992**	**.935**
G_B	ByteWeight	msvc10-32	O2	.809	.950	-	-	.833	.996	.975	.894	**.992**	**.927**
G_B	ByteWeight	msvc10-64	O1	.675	.999	-	-	.813	.999	.949	.969	**.975**	**.983**
G_B	ByteWeight	msvc10-64	O2	.703	.999	-	-	.811	.999	.948	.938	**.972**	**.981**
G_{B*}	ByteWeight*	msvc10-32	-	.775	.953	-	-	.743	.928	.745	.621	**.967**	**.910**
G_{B*}	ByteWeight*	msvc10-64	-	.653	.999	-	-	.543	.999	.645	.413	**.932**	**.985**
G_M	Malpedia57	-	-	.819	.940	-	-	.847	.964	.914	.627	**.976**	**.935**

Table 6.5.: Results (recall and precision) of function entry point discovery for the disassemblers, best F1 score highlighted in green. Original IDA Pro 6.7 results taken from Andriesse et al. [31] (thus no results for ByteWeight and Malpedia57) and reproduced with IDA 7.4.

In our opinion, it does make sense to consider these stubs generally as functions and code because typically they are frequently referenced by other code and consist of code themselves, albeit it being a single `jmp` instruction. We therefore include all of these stubs in the ground truth. On the other hand, external functions which are just represented by data in the form of an offset are not considered functions as they also do not consist of code. However, both Ghidra and IDA Pro will include them when iterating over the list of functions, carrying a flag about their so-called external status. Using this flag, we discard these function entries for these two disassemblers.

Now to continue with the evaluation, we align our evaluation design with the works presented by Andriesse et al. [31] as much as possible. For this, we similarly group by compiler and optimization level and use the geometric mean across results within each group to penalize outliers. For comparability, we list the results they obtained using IDA Pro 6.7 in 2017 and re-run the evaluation with a more recent version, IDA Pro 7.4.

We limit our evaluation on function entry points and instruction recovery, as the results for function starts and boundary detection can be generally considered alike as seen in the evaluation done by Andriesse et al. [31] and we consider function starts more relevant in the given context as we focus on heuristics for FEP discovery. We do not provide a full listing of results of instruction recovery as all disassemblers usually achieve an F1 score of 0.950 or above, as previously observed by Andriesse et al. [31]. Instead, we discuss them briefly along the other aspects.

The overall results are summarized in Table 6.5. We will discuss the results by data set.

Andriesse data set (G_A). An immediate observation is Ghidra's strength for the majority of Linux binaries. We believe this is a consequence of the extensive amount of analysis modules available in Ghidra, which are capable to recognize numerous code constructs including exception handlers and reconstruction of virtual tables etc., which especially should explain the great results for C++ and all optimization levels. One observation is however, that Ghidra's TPR dropped heavily for a number of parameter groups. This is most visible for the `glibc` group, where Ghidra achieves only a 0.11 TPR. We excluded that the error lies in our evaluation scripts by manually using Ghidra on the given binary and observed that Ghidra indeed does produce only very few identified functions. Inspecting the other groups, we noticed similar cases in which Ghidra would fail, not giving an error and producing limited output. Notice that IDA Pro similarly struggles with the SPEC (C), `llvm370-32` binaries, which may suggest that this configuration simply does not match the heuristics used by Ghira and IDA Pro as well as other binaries.

Comparing the results of IDA Pro 6.7 and 7.4, we observe a consistent, significant improvement, competing in many cases with Ghidra. IDA Pro generally produces better results for optimization level `O0` and falls off at level `O2-O3`. This is also in line with what was reported by Andriesse et al. [31].

Nucleus still outperforms IDA Pro 7.4 in many cases and note again that nucleus suffers slightly from the previously discussed interpretation of ground truth with regard to the `PLT` section, as it would otherwise have achieved better results.

SMDA despite not being originally developed for handling ELF files, still achieves solid results for O1/O2, even outperforming Ghidra in a few cases. Similarly to the other disassemblers, SMDA suffers from higher optimization settings. This is mostly related to failure in recognition of exception handling and non-returning calls, which especially interferes with the implemented Gap analysis. For improvements on Linux binaries, an adaption of nucleus' methodology for connecting basic blocks as an alternative method for Gap analysis appears a logical step to improve results in future work.

With respect to instruction recovery, nucleus achieves the overall best results with an F1 score between 0.985 and 0.997 across all groups, which is probably a consequence of its underlying linear sweep in the first phase. Ghidra reaches F1 scores between 0.976 and 0.998 with the mentioned outliers for `llvm370-32`: 0.820 for servers and 0.885 for SPEC (C). IDA Pro 7.4 achieves an F1 score between 0.973 and 0.993 with similar outliers of 0.921 for SPEC (C) and 0.944 for the servers group. SMDA has no heuristics tailored for C++ code and recovers instructions only with a F1 score of 0.950 and 0.960 on the respective groups but also achieves a F1 score between 0.975 and 0.990 for the remainder.

Bao data set (G_B). On the benign PE data set, SMDA achieves both the highest recall and F1 in all cases, specifically because it cleanly recognizes the aforementioned chains of jmp thunks to their full extent as present in some of the binaries. The reduced precision of SMDA for 32bit code originates in mistakenly recognized tiny functions in gaps such as nullsubs (functions consisting entirely of a `ret` instruction), which are not part of the ground truth.

Nucleus produces results in similar quality to SMDA but suffers from similar misclassifications.

Ghidra and IDA Pro perform significantly worse on the PE binaries than ELF binaries. The major reason are missed functions in the jmp thunks, which has impact similar to the missed `PLT` section for nucleus on the ELF files.

Bao data set, dumped (G_{B*}). When confronted with the same binaries of G_B, but after being memory dumped and without PE header, the disassemblers have to work in their respective raw binary mode as no structural information is available. For Ghidra, this means only a smaller performance loss of 3-5% in TPR, while IDA Pro suffers more, losing 10% in TPR for 32bit and 25% for 64bit code.

When Nucleus is not able to restrict its search to a code section, the whole buffer is analyzed. As a result, this leads to a significantly lowered precision with many false positives outside of the actual code.

On memory dumps, SMDA can show its full strength as it uses the same methodology regardless of how the code is organized in the binary (unmapped/dumped) and as a result only loses slightly in accuracy due to the inavailability of information on executable section borders otherwise provided by LIEF. Opposite to Nucleus, SMDA will only search code in gaps between already established functions, avoiding the false positives outside these boundaries at the cost of a few missed functions at the borders.

Malpedia57 data set (G_M). The same trend as for the dumped ByteWeight data set G_{B*} shows also for the Malpedia57 data set. Ghidra and IDA Pro achieve a TPR of 0.819 and 0.847 respectively, which again is probably related to less applicable heuristics compared to properly parsable PE files.

Nucleus covers 91.4% of the expected functions but again has a low precision of 0.627 because the full buffers are scanned.

SMDA once more is not affected by the changed data representation and recovers solid 97.6% of the expected functions with an overall F1 score of 0.948.

With respect to recovered instructions, Ghidra finds 88.84%, IDA Pro finds 89.53%, and SMDA finds 98.78% of the instructions. Nucleus again due to its linear sweep finds the most with 99.74% but has only 0.480 precision as a lot of data is mistakenly recognized as code as well.

Summary

In summary, the evaluation has shown that the approach of SMDA with its presented heuristics for FEP discovery is definitely capable of recovering code and Control Flow Graph information, especially for the scenario of concern in this thesis, which are memory dumps for Windows malware, without requiring further assumptions about the structural layout of buffers presented to it. This means in conclusion that RQ_6 has been successfully answered.

6.3. MCRIT: MinHash-based Code Relationship Identification

In the previous section, we demonstrated that accurate disassembly of memory dumps is possible and sufficiently achieved by our method SMDA. In order to answer research questions RQ_7 on third-party library usage and RQ_8 on code sharing among Windows malware families, we first need to be able to estimate code similarity.

Code similarity is a popular research topic in recent years. Haq and Caballero list more than 50 works since 2010 on the topic in their survey [33]. However, only 17 of them have been evaluated on 10 malware samples or more and generally for none of these, their proof of concept code is available either as open source or binary. Thus, we cannot simply use one of the existing proposed methods out of the box but have to find one ourselves. We therefore study the theory described in these works and gather best practices that we can use in our own approach.

For the granularity of our approach, we consider functions as unit to operate on, similar to most other approaches [33]. Comparison on the level of full binaries would not give us insight into which components are responsible for overlaps. Basic blocks or elements even below would likely not carry enough context for what we aim to achieve in this study.

We again start with a definition of requirements for our solution. Obviously, the approach should provide a reliable estimate of similarity among functions, allowing inexact matching to accomodate for changes in code. Because we ultimately want to use the method on the Malpedia corpus which consists of 95% families in 32bit code only (cf. Section 4.4.2), we do not pursue a cross-bitness or even cross-architecture solution but instead optimize for matching code in the same bitness and architecture.

Next, the method should allow for scalability into millions of functions, while maintaining a fast response time, e.g. a matter of seconds for the lookup of a full sample.

Finally, the representation for functions in the index should be efficient, ideally significantly smaller than the code indexed itself.

In this section, we now introduce MCRIT, the MinHash-based Code Relationship Identification Toolkit, our approach that allows efficient one-to-many code similarity analysis based on exact and MinHash-based locality-sensitive hashing. We first explain the methodology behind MCRIT and discuss both code hashing methods in detail. Next, we conduct an evaluation to optimize the configuration of the system and show that it fulfils the requirements.

6.3.1. Methodology

MCRIT uses two methods to enable efficient one-to-many code similarity analysis: exact and locality-sensitive hashing. Hashing is a well-researched method for similarity search [303].

We use immediate hashing of function contents to identify exact code duplicates across samples. To increase robustness, we apply transformations that remove absolute addresses from the disassembly, making it more robust against relocation. This method is called position independent code (PIC) hashing and has been described in detail by Cohen and Havrilla [187]. We revisit their approach in the next section and describe how we use it in the context of MCRIT, referring to it as PicHash.

We additionally use a technique to also perform fuzzy matching. This is necessary to detect variants of functions that may exist due to slight changes to the source code when the compiler or optimization level is changed. We choose a method based on locality-sensitive hashing (LSH), in which the hashing used preserves the similarity of hashed objects, making it suitable for nearest neighbor search. LSH-based methods are known for their great scalability and can provide efficient lookups, as techniques exist that allow to lower the algorithmic complexity for full pair-wise comparisons from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ and allowing single lookups in $\mathcal{O}(\log n)$ when accepting a certain degree of error. These methods have been for example successfully applied to genetic sequence [304] and association [305] analysis, as well as text [208] and image [306] similarity detection. With regard to code similarity, min-wise independent permutations locality sensitive hashing (short: MinHash [208]) and SimHash [215] have been used successfully. While these approaches use different similarity measures they have been compared in various studies. Henzinger [307] showed in an experiment using 1.6 billion webpages that SimHash seems to have higher precision, while MinHash has higher recall. Shrivastava [308] additionally showed experimentally that MinHash outperforms SimHash when used on binary vectors. For these reasons, we favor MinHash over SimHash. Best to our knowledge Jin et al. [207] were the first to use MinHash for code similarity and it has been used multiple times since then [211, 193, 195, 205, 196]. Notably, Dullien [214] has shown that using SimHash leads to solutions providing decent results as well.

In order to use MCRIT for code similarity measurements, the data of interest first has to be indexed using both the PicHash and MinHash method. It makes sense to allow annotation of arbitrary meta data to indexed functions, allowing association of functions with information such as file hashes, malware families and versions, or free and open source software (FOSS) library names.

Intended use of the system are lookup queries that can consist of a single or a set of functions and a threshold for MinHash similarity. The result of a query are all functions with their matching score, including their corresponding meta data, which enables e.g. aggregation by file hash for one-to-many program comparisions.

A reference implementation of MCRIT has been written in Python and is publicly available [309].

PicHash

As motivated previously, we use a direct function hashing method, for finding literal code duplicates. Cohen and Havrilla [187] applied this method to the CERT Artifact Catalog, a database of disassembled functions. In their experiments, they were able to reduce an initial 1.5 billion functions to just 39 million unique functions in their PIC representation. We interpret this as another indicator that tremendous redundancy exists in collections of binary files, as argued by us earlier in Section 4.3.3 as well.

Given the disassembly of a function as input, we perform the same transformations as suggested by them. To achieve position-independent code hashing, we need to perform wildcarding of absolute and relative code and memory references. In detail, we replace values that fulfil the following conditions with zero bytes as proposed in [187]:

- All operands that can be interpreted as memory references.
- All immediate operands that could be interpreted as an address A within the current buffer B, i.e. $BaseAddress <= A < BaseAddress+|B|$ (cf. Section 6.2.1).

We also replace relative call and jump offsets when they are interprocedural, i.e. point outside of the function's scope. This additionally hardens the PIC hashes against changes caused by padding bytes or code reordering. We leave all other opcodes, operands, and immediates untouched.

Just as proposed in [187], we now order all instructions in the given function by their address. We then concatenate their byte representation and use SHA256 for deriving a cryptographic hash. To preserve space, we only take the first 8 bytes as final representation of our PicHash.

Note that this method can be applied to functions of arbitrarily small size but it does not capture semantics of the functions in some cases, as explained in [187]. A prominent case are code stubs such as import thunks or wrapper functions. For example, import thunks are simple functions that consist of a single `jmp dword ptr [offset]` instruction, where the offset is typically a WinAPI function's start address. Because the offset would be masked by the PIC hashing method, all import stubs are represented by the same hash. Similar cases exist for wrapper functions that simply pass on arguments to a function but carry out additional error handling. In cases of functions with identical function signatures, their wrappers may be byte-identical. However, Cohen and Havrilla [187] note that they found most PIC hashes to represent functions with unique behavior.

We will study the effects of small function ambiguity with regard to function labels as provided by our ground truth in detail in the evaluation (cf. Section 6.3.2).

MinHash

The second method for indexing and measuring similarity of code is based on MinHash. In this section, we give an overview of the theory of the approach and explain the specific techniques from recent findings that we use to improve the method.

In the original work, Broder [208] defines resemblance and containment between text documents as a value between 0 and 1, expressing how similar they are. Resemblance can be computed on a fixed-sized sketch, i.e. a (smaller) representation derived from the original documents. Furthermore, he assumes that documents can be interpreted as sets of tokens d, for example shingles of consecutive words (i.e. n-grams). This allows Broder to define the calculation of resemblance as a set intersection problem, which is equivalent to the Jaccard similarity [280]:

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

The core idea of MinHash is that the relative size of overlap between sets can be approximated through a method of random sampling that can be done independently on each document. This is proven by Broder through showing that the minimum values of random permutations of the original sets are indeed unbiased estimates of the original resemblance. For implementation, he argues that using a projection in form of a hash function is a viable choice that will produce a negliable amount of collisions in practice. Using k independent and uniform hash functions h_k, the (so-called k-hash) MinHash signature m_k for a document D consisting of tokens d is defined [310] as:

$$m_k(D) = \min_{d \in D}(h_k(d))$$

Broder furthermore states that for practical applications, the MinHash signature size can be used to trade computation and speed for accuracy, with a value of $k = 100$ or $k = 200$ providing sufficiently low estimation errors which are calculated as $\epsilon = \frac{1}{\sqrt{k}}$ [311].

For our use case, we want to use MinHashing for the following problem. Given a set of functions f found in one program, we want to find similar functions in other programs. We present the considered features and their representation in detail in the next section and omit their discussion at this point.

Regardless of the features, in order to efficiently use MinHash and benefit from its properties for this use case, we need to be able to use it for nearest neighbor search. Among others, Rajaraman et al. [311] describe the theory behind the application of MinHash as a locality-sensitive hashing scheme usable for nearest neighbor search. We effectively want to find neighbors for each element in f and aggregate them by their origin, which lets us estimate program similarity.

Rajaraman et al. propose the use of so-called bands, i.e. a division of the whole MinHash signature of length k into d divisions of size r each, e.g. consecutive signature entries. Now, by hashing those r entries again and sorting them into buckets managed

per band, we can derive a d-fold lookup table, grouping signatures that already match in r signature entries each.

When we now want to find neighbors of a query signature, we can simply apply the banding technique to the query signature as well and then perform lookups in all buckets for entries that share the buckets. Joining the results from all bands, we end up with a selection of candidates, that we need to perform the actual signature matching against. According to Rajamaran et al. the probability $p_m(s, d, r)$ that the MinHashes for a pair with given Jaccard similarity s will at least match in one of d bands of size r is calculated as:

$$p_m(s, d, r) = 1 - (1 - s^r)^d$$

The resulting curve for this function is S-shaped and resembles a logistic function, regardless of the constants d and r. This means that the matching probability increases significantly with the actual similarity. In an example Rajamaran et al. show that for $d = 20$ and $r = 5$ (giving a signature length of 100), only one in 3,000 pairs of similarity $s = 0.8$ will be a false negative. According to them, this scheme allows for sub-linear and thus very time-efficient lookups of all functions in f against the whole index.

Further optimizations targeting efficient computation of MinHash have been researched, for which we want to highlight two approaches. On the one hand, the use of k individual hash functions may be computationally expensive. An alternative is to use just one hash function h and synthesize further independent hash functions h_k by transforming the output of h using a single XOR operation with a random value in the output size of h for each derived hash. As this causes a non-destructive permutation of bits, the result is effectively a new hash function [311]. On the other hand, Li and König [312] showed that storing only the one or two least significant bits per entry (so-called b-bit hashing) opposed to the full 64 bits per entry in a MinHash signature can drastically improve computation and reduce space consumption. Obviously, this modification increases variance massively. However, they also showed that the signature length only has to be increased by a factor of 3, when considering a threshold of 0.5 for resemblance to achieve the same accuracy, yielding a 20x improvement in processing speed.

For our approach, we will combine the presented ideas and use k-hash b-bit MinHash with XOR-synthesized hash functions and banding to enable efficient index lookups. We use M_k^b to identify the parameters used, e.g. M_{64}^8 for MinHashing with a signature length of 64 and a 8 bit representation per entry. This allows to choose a number of parameters for which we will evaluate meaningful combinations in Section 6.3.2 in order to identify a combination with the most promising tradeoffs. In addition, we use the concept of segmentation for MinHash signatures in order to allow to split a signature into parts that only contain hashes for certain features.

We will next address what is actually represented by the MinHashes and present our considered features.

Feature Extraction and Representation

We now discuss how functions and code can be represented by derived features that are usable for MinHashing. In the related work (cf. Section 3.3.2) we extensively discussed the taxonomy of code similarity methods introduced by Farhadi et al. [188] and gave a broad overview of techniques used in other works.

Text-based approaches are applied on full binary scale, therefore they do not fit our requirement of function-level matching and are left out of scope.

Approaches that are token- or metrics-based are well researched and have been shown to perform well. As examples of token-based approaches, Karim [190] as well as Walenstein et al. [191] used n-grams and n-permutations (short: n-perms) as representation for comparisons. An n-perm are all possible orderings of a given n-gram, which is equivalent to simply using sorted n-grams, with the idea to increase robustness against instruction sequence reordering. N-grams are also close to the original MinHash idea, as they are one of the most natural shingles.

Metrics-based approaches as introduced by Bruschi et al. [197] numerically summarize certain features of functions, e.g. number of instruction, basic blocks, or specific instruction types. Eschweiler et al. [199] did not use MinHash but relied on kNN clustering for nearest neighbor search, using a selection of metrics as a primary method in discovRE. They specifically showed that numerical features can be highly effective for finding similar functions, even across optimization levels and architectures.

The most prominent approach in category 4 is the method presented by Dullien et al. [200], which is the foundation for the BinDiff tool. To enable a similar way of structural subgraph comparisons in conjunction with set similarity, Cesare et al. [203] introduced flowgraph strings.

Behavorial-based approaches have been used primarily for cross-architecture matching scenarios, which are out of scope for this work. As this class typically involves either emulation or symbolic execution, these approaches are expected to be computationally much more expensive compared to the features presented before. Important works have been published by Jin et al. [207], Egele et al. [210], and Pewny et al. [211], which use input-output pairs and blanket execution for basic blocks of functions, in order to abstract code to its semantics.

Hybrid approaches are simply considered a combination of the previous categories.

A new class of approaches from recent years not covered in the taxonomy are based on the natural language processing technique *word2vec* by Mikolov et al. [216]. One of the best perfoming examples using embeddings is SAFE by Massarelli et al. [219]. While outperforming many other works, their approach uses a vector length of 100 with 64bit floating point values as representation, resulting in 800 bytes of storage required per function. This exceeds the actual code of the functions to be represented by one or more orders of magnitude in most cases (cf. Table 6.8), which violates one of our requirements.

In summary, we are aiming for a single platform and can expect rather consistent compiler usage as shown in Section 4.4.2. As we strive for matching efficiency in our

Instruction	Normalized Instruction
`push ebp`	`S REG`
`mov eax, 0x14`	`M REG, CONST`
`xor ecx, edx`	`A REG, REG`
`jmp 0x4022a0`	`C CONST`
`lea eax, [rcx - 2]`	`M REG, PTR`

Table 6.6.: Examples for instruction normalization as used for n-perm tokenization. *Instruction* describes the originally decoded instruction. *Normalized Instruction* is the corresponding instruction with masked mnemonic and operands.

work, we will use a combination of token- and metrics-based techniques, as they both provide good capture potential of a function's characteristics and they are directly usable with MinHash.

Token-based features

To enable the use of token-based features, the instruction stream of functions has to be divided into subsequences. Before actually creating these subsequences, we first normalize all instructions, by replacing the mnemonic and operands with tokens. The idea behind this normalization is to derive more robust tokens, a technique that has been used by several other approaches as summarized in the survey by Haq et al. [33].

Concretely, we replace the actual mnemonic with symbols for one of the 8 following semantic categories:

- A: arithmetic/logic - e.g. `add`, `and`, `inc`, `shl`, ...
- C: control flow - e.g. `call`, `cmp`, `je`, `loop`, ...
- F: floating point - e.g. `fadd`, `fcom`, `fld`, `fmul`, ...
- M: memory - e.g. `lea`, `lods`, `mov`, `xchg`, ...
- P: privileged - e.g. `hlt`, `int3`, `lgdt`, `syscall`, ...
- S: stack - e.g. `enter`, `pop`, `push`, `pushad`, ...
- X: extended - e.g. `addpd`, `cvtpi2ps`, `vfmadd132pd`, `vxorpd`, ...
- Y: cryptographic - e.g. `aesdec`, `crc32`, `sha256msg1`, `xcryptebc`, ...

At the time of writing, this scheme has a semantic classification for 741 instructions, which were encountered during processing of code. The full list is available via the SMDA GitHub repository [290].

Similarly, the operands are replaced with tokens of the classes, similar to what was used by Adkins et al. [313]:

- CONST: for immediates
- PTR: for all pointer-based memory references
- REG: for registers, with subgroups for regular, segment (SREG), extended (XREG), and floating point registers (FREG).

Examples for resulting transformations of this normalization scheme are shown in Table 6.6 and Figure 6.2.

Once the instructions are normalized, we iterate over all basic blocks and derive all n-perms of length n within them, for example $n = 3$ or $n = 4$. Should a block be shorter than n instructions, we produce a single shorter n-perm of the respective length.

The idea of producing n-perms per block is that it provides token-based information with respect to the control flow graph structure. This gives weight to smaller components that would otherwise disappear if the processing instead would span across the basic block boundaries and their respective (conditional) jump instructions.

After all instructions have been normalized and tokenized into n-perms, they already have been generalized to a degree that will suffice the requirements of inexact matching. Taking the string representation of the n-perms, they can directly be processed by the following steps of the MinHashing procedure.

Metrics-based features

For metrics-based features, we consider a total of 29 potential numerical values as candidates. These values can be categorized as follows.

First, there are very basic values such as the overall number of instructions (`num_ins`) and the number of basic blocks (`num_blocks`) as well as the number of control flow graph edges (`num_edges`) between those blocks. The cyclomatic complexity can be calculated from these values [314] as `cyclomatic_complexity` = 2 + `num_edges` - `num_blocks`. Characteristic may also be outstandingly large blocks, which are represented by the feature `max_block_size`. Eschweiler discussed also capturing a representation for the amount of local variables, which we incorporate as (`stack_size`).

Further features are based on relationships in the CFG, such as the number of strongly connected components (`num_sccs`) and related to this, the number of loops ((`num_loops`). Another measure for complexity is known from source code metrics but can be also derived for disassembled functions: the nesting depth ((`nesting_depth`).

Features that target the interprocedural CFG are for example the number of calls (`num_calls`), returns (`num_rets`) but also incoming data and code references to the function (`num_inrefs`) and all outgoing references from the function (`num_outrefs`).

Eschweiler also used counts specific to semantic categories of instructions, which proved to be very reliable. As we already defined such a scheme for token-based features, it makes sense to incorporate them for metrics as well. For the 8 classes we defined, we use both the absolute counts as well as their relative share with regard to the total number of instructions in the function of interest, denoted as `num_ins_<CLASS>` and `num_ins_<CLASS>_rel` respectively, totalling 16 more feature candidates.

As shown by Eschweiler et al. [199], not all of the values also discussed here are suited for estimating code similarity. We conduct a two-way correlation analysis of these features in Section 6.3.2 to determine the most expressive and complementing features.

One important aspect after having derived the metrics-based features for a given function is that they are exact values. Now, if they were directly used as input to the further processing steps of MinHash, the transformation through hashing would lead to exact comparisons, which is not in line with the required inexact matching capability for the system.

We therefore propose the following scheme to enable inexact matching. We first note that all values derived from the above features can be represented as discrete values. The only features that are not direct count values and thus by definition already discrete are the relative share of instructions classes, which can be simply rounded to their full percentages, effectively mapping them to the range of 0 to 100.

To achieve fuzziness, we want to reduce a large set of values to a smaller set, e.g. by grouping similar values together, a procedure known as quantization. One quantization technique is to define value ranges to be represented by a single value, a so-called bucket.

To allow proportional fuzziness, we use bucket sizes that grow with the size of the value they capture. As a baseline, we use powers of two as a scale and choose the bucket values as follows. Apart from the value 0 which has its own bucket of 0, for any exponent $i \in \mathbb{N}_0$, we set a width of $W_i = 2^i$ to create buckets in the interval $R_i = [2^{2i}, 2^{2(i+1)})$. As a result, a value j falling into R_i can be quantized using $Q_i(j) = 2^i \cdot \lfloor \frac{j}{2^i} \rfloor$.

As an example, consider $i = 1$, and thus $W_1 = 2$ and $R_1 = [4, 16)$. Given $j = 11$, and with $j \in R_1$, the bucket value for j is $Q_1(11) = 10$.

An issue with bucketing arises with values close to the borders chosen for buckets, in that values close left and right of a bucket border fall into separate buckets despite their actual proximity. To overcome this issue, we can make use of the fact that MinHash approximates set similarity. By representing values not by a single but multiple bucket values, e.g. incorporating the bucket values to the left and right of the original bucket, comparisons even in the unlucky border case will now result in a non-zero similarity. We call the resulting tuple a LogBucket Triplet.

Picking up the previous example, $j = 11$ would now be represented by the set $S_j = \{8, 10, 12\}$. Additionally considering a neighbored value $k = 12$, which would normally fall into a separate bucket $Q_1(12) = 12$, it would now instead be represented by $S_k = \{10, 12, 14\}$. Indirectly using the Jaccard approximation enabled through MinHash, we get $J(S_j, S_k) = \frac{|S_j \cap S_k|}{|S_j \cup S_k|} = \frac{2}{4} = 0.5$. Note that even values falling into buckets an additional position further away (e.g. 14 or 6) achieve a similarity of 0.2 but all other values result in no similarity. Only 0 is a special case for which we use a fixed triplet $\{-1, 0, 1\}$

This property of inexact matching directly translates into MinHash. Assuming a function with 11 instructions, we can apply Q and determine the set of buckets as before ($\{8, 10, 12\}$). Now we can transform this set into strings by simply prefixing the bucket values with the feature name, producing shingles `num_ins:8`, `num_ins:10`, `num_ins:12` which are compatible with the following steps of the MinHash procedure.

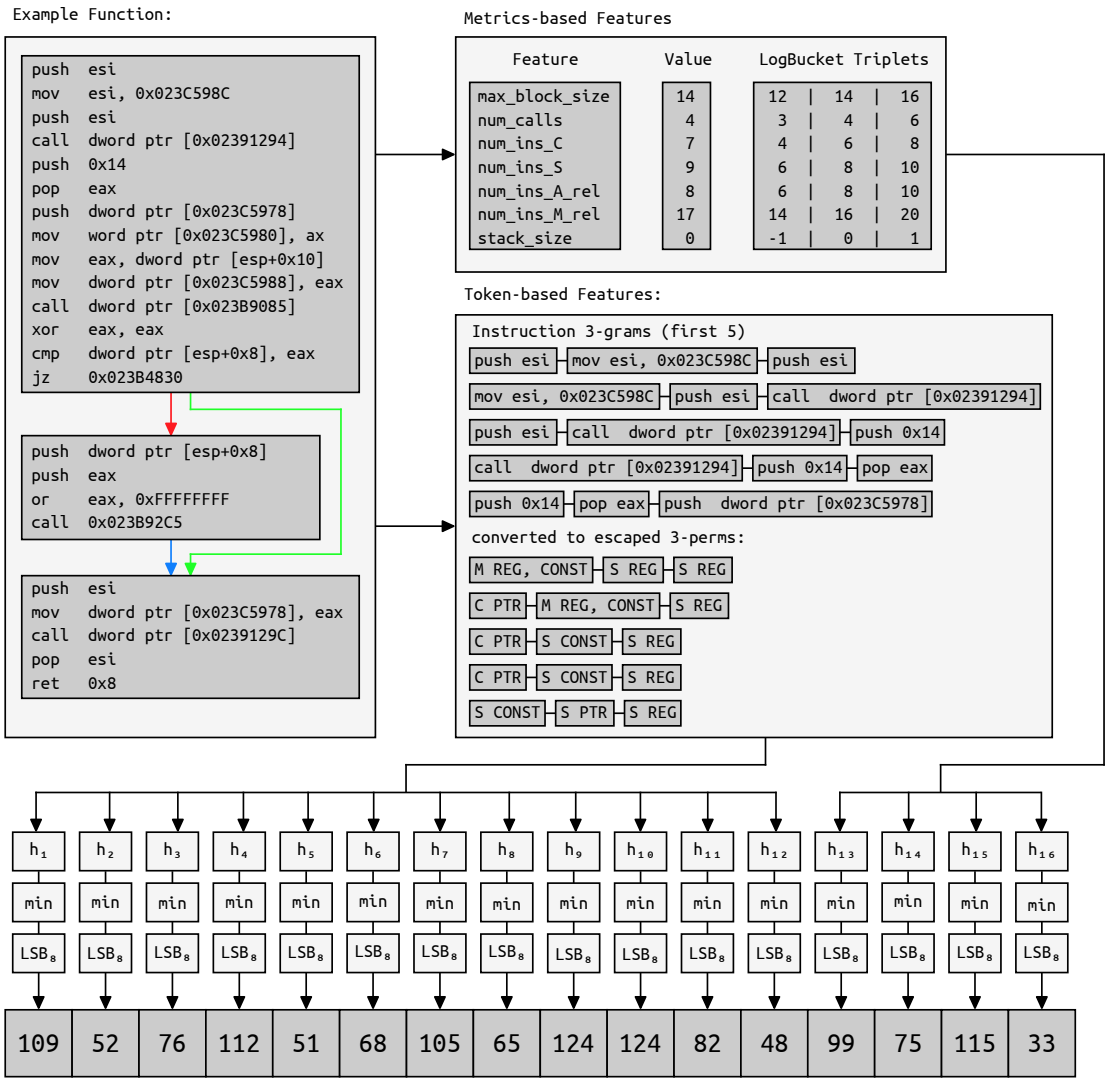


Figure 6.2.: A complete example for calculation of a M_{16}^{8} MinHash signature, segmented into 12 token-based and 4 metrics-based entries.

Example Generation for a MinHash Signature

Figure 6.2 gives an example for the full procedure of generating a MinHash signature from a given disassembled function. The system is parameterized M_{16}^{8}, meaning that the signature has a length of 16 and entries use 8 bits. The example input function consists of 23 instructions in 3 basic blocks.

We can now determine metrics-based features as introduced in Section 6.3.1, for example for the 7 features shown in diagram. Using the concept of LogBucket Triplets, we are able to quantize the single discrete value into three normalized values and together with the feature label arrive at $7 \cdot 3 = 21$ strings that are processed by the hash functions.

Similarly, token-based features as introduced in Section 6.3.1 can be determined. In the example, 3-grams are extracted and then converted into 3-perms using the escaping for semantic instruction categories and operands. These are forwarded as input to the hash functions as well.

For each output of a hash function $h_{1..16}$, min calculates the minimum value and LSB_8 reduces the (by default 32bit) output to the specified number of Least Significant Bits, in this case 8. The resulting MinHash signature then has 16 entries with values in the range from 0 to 255.

To determine the similarity of signatures means to calculate their resemblance (cf. Section 6.3.1). In this case it would be a field-wise comparison, dividing the number of equal fields by the signature length 16. Two signatures having the same value in 12 positions would have a resemblance of $\frac{12}{16} = 0.75$.

6.3.2. Evaluation

In order to analyze if the proposed method fulfils the defined requirements, we conduct an evaluation of several aspects. We first describe the data set used for this evaluation, which consists of 16 very commonly used open source libraries in multiple versions each.

Next, we focus on the exact matching method of PIC hashing to get a better understanding of how the data set is shaped and what can be already achieved by this method alone. This allows us to also investigate the effects of expected label ambiguity for small functions. After this, we determine which feature configuration to use for MinHashing by performing a correlation analysis. We then continue to evaluate further system parameters such as the MinHash signature length and composition with regard to token- and metrics-based features. One goal in this context is to see how well short signatures can still perform.

We conclude with an in-depth discussion of the most promising parameter configuration.

Data Set

To evaluate the proposed method, we need a data set that provides us with (syntactic) variations of semantically close or identical functions in order to show that the exact and inexact matching produces results as expected.

An established way according to the survey by Haq et al. [33] is to create this kind of ground truth by using several release versions of the same software projects, as we can expect mostly incremental changes between versions but that they otherwise maintain their functionality. Similar to other works [204, 213] before, we resort to using a selection of commonly found third-party libraries.

In order to be able to determine pairs of similar functions within and across these libraries on machine code level, one typically needs availability of function level symbolic information (i.e. function names). The presence of this information generally depends on

Library	MinVersion	MaxVersion	Versions	Packs	Files	Functions
libarchive	3.3.2	3.3.3	2	4	912	14,018
libbz2	1.0.6	1.0.6	1	3	30	380
libenca	1.18	1.19	2	5	240	1,326
libgcrypt	1.7.1	1.8.2	11	24	5,220	70,515
libgmp	6.1.1	6.2.0	3	8	8,172	16,061
libgnutls	3.4.14	3.6.11	13	26	12,659	168,613
libgpg-error	1.23	1.37	15	30	1,278	26,537
libiconv	1.14	1.16	3	8	32	5,351
liblzma	5.2.2	5.2.5	4	12	1,656	8,686
libnettle	3.2	3.5.1	6	18	5,270	15,861
libogg	1.3.2	1.3.4	3	10	40	1,528
libspeex	1.2.0	1.2.0-4	2	7	266	2,508
libssh	0.7.3	0.9.3	14	27	3,262	56,888
libxml2	2.9.4	2.9.10	7	12	1,048	72,537
libzlib	1.2.8	1.2.11	4	9	342	4,355
openssl	1.0.2h	1.1.0i	11	24	30,356	355,758
total	-	-	101	227	70,783	820,922

Table 6.7.: Groundtruth based on development libraries provided by Shift Media Project [315]. Each library is present in multiple versions and compiled by multiple versions of Visual Studio (*Packs*). *Files* lists the count of individual Object files produced by this, while *Functions* shows the count of functions with labels.

compilation settings. For our evaluation, we decided to use the extensive collection of pre-configured, linker-ready development libraries provided by the Shift Media Project [315]. Apart from including the source code and reference Visual Studio projects, this data already contains full symbolic information, which makes it equivalently good to compiling it ourselves.

All libraries have been natively compiled with one or more versions of Visual Studio, which was previously identified to be the most common compiler used for Windows malware (cf. Section 4.4.2).

A full listing of the libraries used is shown in Table 6.7. In total we use 16 different well-known libraries providing commonly used functional aspects such as encryption (`libcrypt`, `openssl`), compression (`libarchive`, `libzlib`), or parsing (`libenca`, `libxml2`).

Each library (except `libbz2`) is present in multiple release versions and compiled with up to four versions of Visual Studio (ranging from 2013 to 2019), producing a total of 227 unique combinations. All of the combinations are available for 32 and 64bit each and the resulting precompiled library files decompose to 70,783 Object files containing 820,922 individual functions with labels. It has to be noted that some libraries listed are contained within other libraries, so these numbers are obtained after application of basic filtering on the Object file level to avoid direct cross-library code duplicates.

For disassembly, we used IDA Pro 7.4, as the software already provides capability for interpretation of all information found in the Object files, including demangling of function names. Using the symbolic information and bridging expected gaps in naming convention (32bit functions may be prefixed with one or more underscores indicating the calling convention used), a total of 19,041,057 function pairs within the same bitness can be determined, with 1,679,918 of them being across libraries.

Instr.	Bytes	Functions	PicHash	Matches	Pairs	TPs	TPR	FP_F	FP_C	PPV	PPV_C
1	222,293 (0.125%)	44,994 (5.481%)	9	412,246,223	868,714	868,104	.999	147,012,694	264,365,425	.002	.003
2	556,886 (0.438%)	71,314 (14.168%)	1,894	84,438,718	2,209,293	1,415,404	.641	63,528,139	19,495,175	.017	.068
3	259,942 (0.584%)	25,431 (17.266%)	947	3,468,964	382,352	345,802	.904	2,518,565	604,597	.100	.364
4	231,928 (0.714%)	16,369 (19.260%)	1,031	638,504	194,074	160,893	.829	363,846	113,765	.252	.586
5	420,513 (0.950%)	23,023 (22.064%)	1,646	1,380,661	334,969	282,358	.843	923,678	174,625	.205	.618
6	483,044 (1.222%)	24,599 (25.061%)	1,658	815,458	382,632	298,963	.781	312,855	203,640	.367	.595
7	507,556 (1.507%)	21,586 (27.690%)	1,484	1,022,543	254,434	209,846	.825	573,444	239,253	.205	.467
8	341,432 (1.699%)	12,617 (29.227%)	1,105	240,804	172,157	117,808	.684	71,329	51,667	.489	.695
9	426,553 (1.938%)	14,158 (30.952%)	1,128	266,394	201,552	160,222	.795	92,168	14,004	.601	.920
10	431,697 (2.181%)	12,596 (32.486%)	1,339	351,046	363,065	101,952	.281	247,638	1,456	.290	.986
11	579,244 (2.506%)	15,834 (34.415%)	1,651	877,019	2,478,111	770,474	.311	76,515	30,030	.879	.962
12	512,712 (2.794%)	13,269 (36.031%)	1,152	199,814	149,008	114,085	.766	74,012	11,717	.571	.907
13	585,000 (3.123%)	13,796 (37.712%)	1,147	398,965	189,989	148,686	.783	246,548	3,731	.373	.976
14	590,324 (3.454%)	12,643 (39.252%)	1,001	1,040,795	628,852	591,638	.941	448,719	438	.568	.999
15	607,678 (3.796%)	12,117 (40.728%)	1,332	1,228,013	1,256,650	1,193,629	.950	34,264	120	.972	1.00
16	453,393 (4.050%)	8,628 (41.779%)	1,153	150,293	115,078	91,481	.795	58,496	316	.609	.997
17	468,905 (4.314%)	8,708 (42.840%)	1,105	263,714	240,243	217,673	.906	42,821	3,220	.825	.985
18	417,781 (4.548%)	7,089 (43.703%)	1,031	54,868	70,547	44,256	.627	10,502	110	.807	.998
19	526,228 (4.844%)	8,329 (44.718%)	1,132	75,181	83,911	57,060	.680	18,037	84	.759	.999
20	458,387 (5.102%)	6,896 (45.558%)	1,226	49,073	63,594	41,291	.649	7,782	0	.841	1.00
21	620,644 (5.450%)	9,113 (46.668%)	1,690	67,554	83,047	56,099	.676	11,455	0	.830	1.00
22	691,227 (5.839%)	9,326 (47.804%)	1,524	149,360	172,971	136,926	.792	9,288	3,146	.917	.978
23	590,968 (6.171%)	7,785 (48.753%)	1,223	59,206	74,760	50,013	.669	8,281	912	.845	.982
24	540,829 (6.474%)	6,972 (49.602%)	1,151	44,571	61,396	41,318	.673	2,341	912	.927	.978
25	567,361 (6.793%)	7,026 (50.458%)	1,187	46,181	65,074	42,500	.653	3,681	0	.920	1.00
26	539,172 (7.096%)	6,470 (51.246%)	1,130	42,876	60,959	37,853	.621	4,801	222	.883	.994
27	624,060 (7.446%)	7,076 (52.108%)	1,167	51,670	69,647	44,098	.633	6,660	912	.853	.980
28	501,635 (7.728%)	5,605 (52.791%)	1,050	33,292	50,441	31,034	.615	2,258	0	.932	1.00
29	594,115 (8.062%)	6,358 (53.565%)	1,033	188,018	201,118	180,808	.899	6,776	434	.962	.998
30	537,795 (8.364%)	5,576 (54.244%)	1,073	28,719	46,442	27,386	.590	1,333	0	.954	1.00
31	626,459 (8.716%)	6,437 (55.028%)	1,026	90,953	115,450	86,127	.746	4,826	0	.947	1.00
32	615,577 (9.062%)	6,096 (55.771%)	1,136	44,123	65,434	41,369	.632	2,754	0	.938	1.00
33-64	21,164,716 (20.951%)	139,228 (72.731%)	30,367	800,722	1,300,298	720,741	.554	76,741	3,240	.900	.996
65-128	36,464,466 (41.435%)	120,136 (87.365%)	33,078	455,509	937,711	440,026	.469	13,947	1,536	.966	.997
129-256	43,047,747 (65.618%)	70,959 (96.009%)	23,663	192,370	408,963	185,126	.453	6,906	338	.962	.998
257-512	30,354,879 (82.670%)	24,870 (99.039%)	9,278	70,517	127,565	67,759	.531	2,094	664	.961	.990
513-1,024	13,620,896 (90.321%)	5,522 (99.711%)	2,051	17,504	28,104	16,998	.605	506	0	.971	1.00
1,025-2,048	8,732,091 (95.226%)	1,793 (99.930%)	593	6,582	9,189	6,486	.706	96	0	.985	1.00
2,049-4,096	3,293,233 (97.076%)	339 (99.971%)	93	1,425	1,807	1,425	.789	0	0	1.00	1.00
4,097-8,192	5,204,269 (100.000%)	239 (100.000%)	25	2,699	3,032	2,699	.890	0	0	1.00	1.00
0+	178,013,635 (100.000%)	820,922 (100.000%)	137,709	923,324,845	14,522,633	9,448,416	.651	216,826,796	285,325,689	.018	.032
5+	176,742,586 (99.286%)	662,814 (80.740%)	133,828	10,808,492	10,868,200	6,658,213	.613	3,403,552	746,727	.616	.899
10+	174,563,488 (98.062%)	566,831 (69.048%)	126,807	7,082,632	9,522,456	5,589,016	.587	1,430,078	63,538	.789	.989
15+	171,864,511 (96.546%)	498,693 (60.748%)	120,517	4,214,993	5,713,431	3,862,181	.676	336,646	16,166	.916	.996
20+	169,390,526 (95.156%)	453,822 (55.282%)	114,764	2,442,924	3,947,002	2,258,082	.572	172,526	12,316	.924	.995

Table 6.8.: PicHash matching results, organized by number of instructions per function. Bytes: number of bytes for functions of this size (with cumulative percentage), Functions: functions of that size (with cumulative percentage), PicHash: unique PicHashes for the given size, Matches: Matches produced by these PicHashes, Pairs: expected matches based on ground truth, TPs: true positives with respect to expected matches, TPR: true positives rate with respect to same-size pairs, FP_F: false positives in function pairs where both are of the same library, FP_C: false positives in pairs with functions from different libraries, PPV: overall precision, PPV_C: precision when only considering FPs with pairs of different libraries.

Accuracy Evaluation of the PicHash Component

We now want to apply the exact matching technique, PicHash (cf. Section 6.3.1), on the data set. The results are summarized in Table 6.8.

First off, exact matching implies that only functions with the same number of instructions can be matched. As shown in Table 6.8, this applies to 14,522,633 function pairs, or 76.27% of all function pairs in the ground truth. PicHash is able to identify 9,448,416 of these and thus potentially achieves an overall TPR of .651, with respect to the same sized function pairs.

Studying the results in more detail, we can make the following observations.

Looking at the distribution of function sizes, we see a disproportionate number of very small functions. The cumulative fraction of functions with 5 instructions or less is 22.06% and that of functions with 10 instructions or less 32.49%, while the median function has 25 instructions. At the same time, when considering the number of bytes of all instructions in these small functions, they account for a way smaller part, e.g. functions of 10 instructions or less contribute only 2.18% of all bytes found in the code of the whole data set.

Next, looking at the number of PicHashes, we see that there are 137,709 unique hashes for 820,922 total functions. Expectedly, the relative number of function variants (functions per PicHash) increases with function size.

An extreme case shows at functions consisting of a single instruction, for which there are only 9 unique PicHashes. As an additional detail, all of these are control transfer instructions, i.e. returns, jumps, and calls. Given that there are 44,994 functions that are grouped by these 9 hashes, and the number of matches is quadratic per group, the result is a huge number of potential matches, totalling 412,246,223. As there are only 868,714 expected pairs derived from function names, this means that most of them are by definition false positives, despite having identical instructions.

While a drastic example, similar effects can still be observed for the other small functions, leading to a low Precision. The number of false positives generally decreases with increasing function size, especially the FPs that contain functions of different libraries. Nevertheless, there are still a significant number of unexpected FPs within the same libraries, even for rather large function sizes. We performed a cursory inspection on a random sample of the function pairs associated with these FPs. A primary reason for these unexpected matches is the renaming of internal functions in the version lineage of a library. As a second source, we identified wrapper functions that only differed in the functions called internally or parameters being passed by reference. As this concrete information is lost when performing the normalization/wildcarding of PicHashing, they reduce to the same byte sequence and thus PicHash. But given these circumstances, we consider these FPs within the same library as much less grave than those across libraries, especially with respect to the upcoming analysis of presence of third party libraries in malicious code as conducted in Section 6.4.3.

A general implication of these observations is that when doing matching on function level and not considering the ICFG these functions are embedded in, there is no way to discern these syntactical duplicates. However, introducing a minimum required size for functions still seems reasonable and can certainly mitigate these effects.

The bottom five rows in Table 6.8 summarize results for potential function size thresholds, ranging from 0 to 20. For the following evaluation of MinHash, we choose a function size threshold of 10 instructions, which we consider a good compromise taking the different aspects mentioned into concern. This threshold will exclude 30.95% of the functions but only 2.18% of the code, reducing the number of function to 566,831 and the overall expected matching pairs to 12,055,913. As a result, using the same function size threshold for PicHash, the method achieves an overall TPR of .587 for same-sized function pairs (or .464 when considering all pairs). For us most importantly, the Precision when only considering cross library FPs (PPV_C) reaches a very good .989 starting at this

threshold. As reasoned before, we consider these FPs seriously more grave than those found within the same library because of the implications for library detection.

feature-name	space	min	25%	50%	75%	max	avg	sd	ρ_{32bit}	ρ_{64bit}	ρ_{same}	ρ_{diff}
cyclomatic_complexity	232	0	2	4	9	561	8.223	13.767	0.961	0.953	0.958	0.923
max_block_size	554	2	9	12	17	6,669	22.173	100.989	0.946	0.966	0.963	0.805
nesting_depth	65	0	2	4	7	101	4.871	4.777	0.973	0.966	0.970	0.960
num_blocks	339	1	4	9	18	817	15.417	22.605	0.974	0.939	0.954	0.933
num_calls	148	0	1	3	7	380	6.308	10.260	0.980	0.980	0.980	0.952
num_edges	480	0	4	11	25	1,376	21.640	36.096	0.969	0.949	0.955	0.929
num_inrefs	114	0	0	0	1	224	1.018	3.907	0.396	0.861	0.605	0.303
num_ins_A	516	0	3	6	12	3,929	16.244	83.531	0.985	0.963	0.981	0.688
num_ins_C	495	0	7	15	32	1,343	26.784	38.942	0.980	0.971	0.976	0.964
num_ins_F	14	0	0	0	0	30	0.003	0.161	0.972	-	0.972	-0.000
num_ins_M	691	0	6	14	33	2,811	29.725	60.236	0.980	0.970	0.981	0.384
num_ins_P	25	0	0	0	0	40	0.012	0.373	0.606	0.693	0.655	0.471
num_ins_S	301	0	2	6	15	932	13.854	24.591	0.947	0.946	0.965	0.006
num_ins_X	261	0	0	0	0	5,897	2.569	66.481	0.893	0.957	0.934	0.637
num_ins_Y	39	0	0	0	0	280	0.161	4.015	0.967	0.989	0.982	0.929
num_ins_A_rel	83	0	9	13	17	86	14.465	9.469	0.914	0.944	0.954	0.106
num_ins_C_rel	97	0	26	33	40	100	33.017	12.931	0.909	0.964	0.952	0.424
num_ins_F_rel	13	0	0	0	0	27	0.004	0.236	0.972	-	0.972	-0.000
num_ins_M_rel	100	0	18	31	46	99	32.194	16.829	0.935	0.939	0.975	-0.648
num_ins_P_rel	13	0	0	0	0	13	0.007	0.207	0.629	0.716	0.677	0.418
num_ins_S_rel	80	0	3	11	30	86	17.022	16.306	0.907	0.963	0.976	-0.339
num_ins_X_rel	97	0	0	0	0	99	1.068	7.057	0.887	0.957	0.933	0.636
num_ins_Y_rel	45	0	0	0	0	84	0.141	2.501	0.980	0.990	0.987	0.917
num_instructions	1,293	10	23	48	100	6,743	89.718	182.991	0.950	0.961	0.973	0.771
num_loops	36	0	0	0	1	100	0.380	1.139	0.957	0.962	0.961	0.894
num_outrefs	88	0	0	0	0	128	0.692	2.843	0.886	0.886	0.886	0.810
num_returns	56	0	1	2	3	232	2.197	3.249	0.973	0.930	0.956	0.682
num_sccs	262	1	4	8	15	555	12.672	16.598	0.973	0.941	0.954	0.931
stack_size	389	0	0	8	48	4,092	44.652	143.470	0.952	0.912	0.971	-0.292

Table 6.9.: Results for the metrics-based feature evaluation. Qualified features highlighted yellow, selected features highlighted green. ρ lists the Spearman rank correlation for function pairs of 32bit and 64bit as well as same and difference bitness only.

Feature Evaluation for the MinHash Component

Before MinHash can be applied to measure the similarity of code, we need to select features to be used in the representation for functions. As mentioned in Section 6.3.2, we will only consider functions of size 10 instructions or more.

Metrics-based Features. We will first focus on the metrics-based features as introduced in Section 6.3.1. We denote the metrics-based feature as F_M.

Similar to the evaluation in [199], we require that on the one hand features cover a sufficiently expressive value range in which they notably vary and on the other hand values correlate notably among ground truth pairs of functions, indicating that they are robust across variants of the same function.

For the first requirement, we do a statistical aggregation and interpretation of the values across all functions in the data set. This will allow us to identify suitable candidates. We require a feature space of at least 100 distinct values (80 for relative features) and standard deviation for values of at least 5.

For the second requirement, we evaluate Spearman's Rank Correlation Coefficient [316] ρ. Because all values are distinct integers, we can use the representation:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

with

$$d_i = rg(X_i) - rg(Y_i)$$

being the difference in rank per value pair after converting the raw values to ranks.

We decided to use Spearman's method and not the Pearson Product-Moment Correlation [317] because it is considered more general (testing for monotonic instead of linear relationship) and more robust to outliers. We apply this analysis in two ways. First, we measure the correlation of the values for a given feature for all function pairs in the ground truth in order to identify the most robust feature candidates. We require a correlation of at least 0.95. Second, we measure the co-correlation of features for each function in order to identify potential redundancy in features. Here, we exclude features if they significantly correlate.

Table 6.9 and Figure 6.3 summarize the results. Most features cover a decent value space and also have a balanced value distribution. Additionally, all but 6 features have a high correlation of 0.95 and more for function pairs of the same bitness. Features not qualiyfing are primarily those addressing much less common instruction types F, P, X, Y which as a consequence lack expressiveness for the majority of functions. Other features that target less common constructs (loops, returns) cover a narrow value range and thus are also not as expressive as others, especially when considering the used binning method to increase fuzziness. Surprisingly, both incoming and outgoing references do not qualify. This may be due to how those references are specifically structured in Object files, resulting in a lower presence and expressiveness as previously reported [199].

In total, 16 of 29 features fulfil the requirements. In consideration of Figure 6.3, we immediately notice that subsets of these qualified features are heavily co-correlated. Features that describe the structural composition of a function's CFG (`num_blocks`, `num_edges`, `cyclomatic_complexity`, `num_sccs`) are strongly correlated with the overall number of instructions (`num_instructions`) but also control-flow associated instructions (`num_ins_C`). We pick `num_ins_C` as single representative for this group to reduce redundancy and also discard `num_ins_C_rel` because we only want one feature per instruction group.

For the other 3 popular instruction groups, we select `num_ins_S` as second explicitly counted feature but the relative values `num_ins_A` and `num_ins_M` because this combination has the least co-correlation.

With regard to the other qualified features, we use each of `num_calls`, `stack_size`, and `max_block_size` because neither of them has significant co-correlation with other previously selected features.

This leaves us with a total of 7 metrics-based features to be used (cp. Table 6.9).

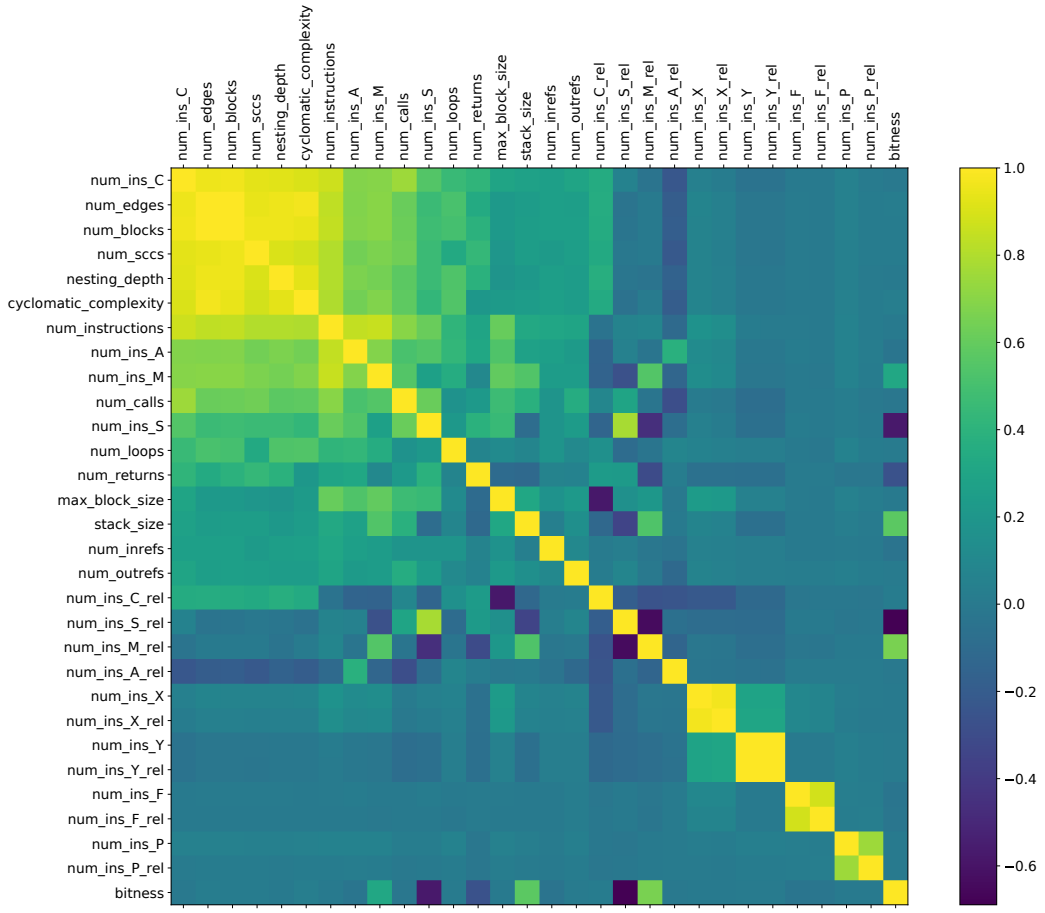


Figure 6.3.: Correlogram for the metric features. Bitness added for context and feature order rearranged to highlight feature groups with high co-correlation.

Token-based Features. We now focus on the token-based features: n-perms. We denote these features as $F_{T,n}$, with n being the length of the instruction sequence used. We will evaluate for values of $n \in \{3, 4, 5\}$.

To measure how well n-perms of different length perform, we calculate the achievable TPR at different similarity thresholds over the ground truth. For this, we transform all functions into their token-based feature representation and use the MinHash approach described in Section 6.3.1 with an instance of M^{32}_{1024} to derive the similarity score between true positive function pairs.

Table 6.10 summarizes the results, averaged over 3 replications with different seeds. The main takeaway is that shorter n-perms provide better matching potential as $F_{T,3}$ achieves the highest result at all thresholds. Even at a threshold of 100, the TPR of .635 already outperforms the results of PicHash (.464), but with at this point unknown penalty in FPs due to the introduced fuzziness of MinHash.

T	$F_{T,3}$	$F_{T,4}$	$F_{T,5}$	F_M	$F_{T,3} \cup F_M$
0	1.00	1.00	1.00	1.00	1.00
5	.993	.988	.984	.996	.999
10	.989	.982	.977	.991	.996
15	.984	.977	.925	.984	.992
20	.979	.972	.920	.978	.988
25	.976	.967	.913	.972	.985
30	.971	.961	.905	.966	.981
35	.963	.952	.898	.959	.978
40	.956	.897	.888	.952	.974
45	.949	.886	.875	.944	.969
50	.942	.876	.866	.889	.963
55	.929	.865	.854	.880	.955
60	.913	.854	.844	.869	.946
65	.899	.837	.825	.861	.939
70	.869	.817	.810	.851	.919
75	.820	.803	.796	.780	.883
80	.803	.785	.781	.736	.869
85	.784	.770	.765	.728	.859
90	.672	.661	.660	.717	.765
95	.650	.645	.645	.706	.749
100	.635	.634	.632	.706	.743

Table 6.10.: Potential TPR at given matching thresholds T for token-based and metrics-based features as well as their best combination.

Combination of Features. Table 6.10 additionally lists results for a similar evaluation of F_M, again using an instance of M^{32}_{1024}. The achievable TPR is very similar to $F_{T,3}$ for lower thresholds of up to 45 but remains higher for upper thresholds.

To analyze if the two feature types complement each other, we additionally list results for a combination where a true positive is recorded when either of the two feature types has a similarity score above the threshold ($F_{T,3} \cup F_M$). We can see that the combination indeed improves the result, which is strictly better at each threshold level. This is beneficial as it allows to choose higher thresholds while maintaining the TPR of individual features while likely lowering the overall FPR.

Parameter Evaluation of the MinHash Component

After evaluating the metrics- and token-based features and showing that they are generally capable of capturing similarity in functions, we now evaluate parameters of the MinHash method M^b_k itself. For signature length k, we consider values of 16, 32, 64, 128, 256 and for b bits used for representation of entries, we consider 1, 4, 8, and 32 bit. These value ranges should give decent coverage for candidates that can be considered as efficient in the sense of required space to store a function's fingerprint.

Given that we have two types of features, we also need to consider by which weight they should contribute, taking ratios of 0%, 25%, 50%, 75%, and 100% into account to model a distribution similar to quartiles. Based on the results shown in Table 6.10, we deliberately consider a required minimum similarity threshold of 60, which should give us a TPR of up to .946.

In total, we evaluate 100 configurations for which we each perform a matching of all functions matched against each other.

Given 566,831 functions with a minimum size of 10 instructions, this yields a total of 160,648,407,865 potential function pairs. However, as mentioned in Section 6.3.1, we are using MinHash banding to filter down to a selection of promising candidate pairs. We decided to use 20 bands of 4 entries, which gives us according to the formula a 6.23% chance to miss a candidate pair at expected threshold 60 but less than 0.5% chance to miss a candidate pair at threshold 70. As a result of banding, the actual number amount pairs evaluated per configuration is much smaller. In fact, the median number of evaluated candidate pairs is 1,456,700,812 (0.91%).

The evaluation of all parameter combinations took a total of 67 days and 21 hours using a Intel i7-6700 CPU (3.40GHz) with 64 GB RAM and a SSD hard drive. The median matching speed using 8 threads was 34,913 function similarity comparisons per second.

The overall results of this evaluation are shown in Table 6.11. We can make the following observations.

Generally, a higher signature length k will lead to higher precision as the estimation error decreases, as explained in Section 6.3.1. Note that this seems to have much less impact on recall, which is already high even for small signature lengths.

Similarly, a higher number of b (i.e. bits for entry representation) directly results in better precision and recall. We can see however that $b = 1$ does not produce any acceptable precision at all. The results of $b = 1$ at $k = 16$ are outliers, where the banding filtered too many candidates.

Also, $b = 4$ does not achieve a precision as good as $b = 8$ or $b = 32$ but catches up when k is increased. Interestingly, we can only assess a marginal difference between results achieved by an entry representation of $b = 8$ and $b = 32$. This generally supports the findings of Li and König [312] but in our case, storing more of the least significant bits seems to offer a better tradeoff.

With regard to the feature composition, n-perms perform better than metric features. In most cases, a combination performs better than the isolated features (cf. Section 6.3.2) but with a higher portion of n-perms promising better precision.

Overall, several parameter combinations offer similarly good results given their respective trade offs. For the following experiments, we choose a setup of M^8_{64} with a segmented signature composed of 75% n-perm and 25% metric features, which represents a good balance of all previously discussed aspects. We note that the 64 bytes of representation equal 20.78% of the average function size when considering functions with 10 or more instructions (cf. Table 6.8), which we still consider space-efficient. For this configuration, MinHash banding produced 835,624,841 matching candidates that were processed in 27,930 seconds (7 hours, 45 minutes, 29,918 comparisons per second), leading to 44,697,628 matches with a TPR of .920 and PPV of .277. We will now proceed to analyze the results of this configuration in greater detail.

b	$F_{T,3}$	k=16		k=32		k=64		k=128		k=256	
1	0	.723	.107	.810	.007	.864	.007	.872	.005	.917	.006
	25	.698	.216	.904	.016	.913	.013	.869	.012	.877	.012
	50	.772	.190	.838	.023	.913	.017	.919	.022	.914	.020
	75	.679	.151	.894	.015	.855	.013	.871	.018	.866	.016
	100	.773	.113	.842	.015	.903	.009	.908	.009	.906	.010
4	0	.873	.016	.777	.016	.874	.022	.877	.012	.876	.025
	25	.877	.033	.873	.059	.878	.080	.880	.090	.879	.106
	50	.885	.015	.877	.096	.922	.210	.922	.222	.882	.234
	75	.921	.053	.923	.182	.925	.190	.925	.248	.925	.263
	100	.909	.096	.916	.163	.915	.152	.922	.173	.922	.183
8	0	.870	.011	.868	.037	.868	.040	.870	.033	.868	.041
	25	.820	.051	.927	.060	.871	.138	.873	.141	.873	.152
	50	.925	.095	.875	.188	.872	.255	.874	.279	.874	.323
	75	.923	.136	.912	.293	.920	.277	.920	.323	.922	.339
	100	.910	.041	.864	.197	.911	.257	.919	.254	.918	.268
32	0	.923	.015	.918	.029	.868	.030	.868	.040	.868	.040
	25	.928	.024	.876	.066	.874	.106	.873	.158	.872	.158
	50	.869	.170	.871	.283	.923	.285	.876	.290	.922	.302
	75	.925	.091	.868	.254	.923	.328	.918	.323	.922	.334
	100	.908	.143	.911	.226	.916	.221	.917	.268	.918	.276

Table 6.11.: Parameter evaluation of the MinHash component. TPR and PPV at threshold 60 shown per configuration.

Accuracy Evaluation of the MinHash Component

After having identified a suitable configuration for our MinHash system with respect to the requirements, we have a closer look at its corresponding results. For this, we analyze the matching results when using different similarity thresholds ranging from 40 to 100. In addition to examining results for Recall and Precision, we also consider the mean Average Precision (mAP) [318] and Average Precision at k (AP@k). Both of these measures are used to describe the quality of the ranked matching results for queries, thus giving insight into how the system performs from a perspective of user experience. Manning et al. [318] define mAP as follows. Given a query $q_j \in Q$ (in our case a function), its expected TPs $\{d_1, ...d_{m_j}\}$, and R_{jk} as the set of ranked retrieval results from top to d_k, the mean Average Precision is

$$mAP(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{k=1}^{m_j} Precision(R_{jk})$$

Should a query not return any TP, the Precision is taken as zero. The value AP@k instead considers only results up to position k, focusing on the first results and thus the ones most relevant to a user.

The results for the reference configuration M_{64}^{8} are summarized in Table 6.12.

Considering Recall and Precision, we see that gains in Recall below the previously used matching threshold of 60 are expensive, as we experience a sharp drop in Precision. This also affects the Precision when only considering matches as FPs when they are in

different libraries (PPV_C), indicating that low thresholds should be used with care, at least when inspecting single functions. Otherwise, we see that the regular Precision PPV increases up to 0.616, while PPV_C increases up to .950, showing that almost exclusively functions from the same library are matched. This discrepancy is very similar to what was observed for PicHash (cf. Section 6.3.2), underlining that deriving ground truth matches solely from function names may be insufficient.

Examining the results for mean Average Precision, we see that the MinHash component achieves values between .849 and .866 throughout all thresholds. This indicates that the previously noted decline in overall Precision does not significantly impact the usability even at lower thresholds, as there remains a stable set of accurate query results being ranked in the most relevant positions. Our result also exceeds the results presented by Ding et al. [204], who achieved a mAP of .536 on a very similar but smaller (63,939 functions) data set of open source libraries. When ignoring false positives in the same library, the values increase much further, achieving a mAP_C of up to .978 which only notably decreases for thresholds below 50. This means that the system almost always returns only functions from the same group of libraries, especially in the first ranks, as AP_C@1 and AP_C@10 additionally underline.

Based on all of our findings and considering the combination of PicHash and MinHash, we can conclude that MCRIT is a system that can successfully determine function similarity, focusing primarily on structural code similarity. This makes it suitable for the task of identifying code sharing and third-party library usage in Windows Malware.

T	TPs	Matches	TPR	PPV	PPV_C	AP@1	AP@10	mAP	AP_C@1	AP_C@10	mAP_C
40	12,736,401	212,078,358	.946	.060	.101	.862	.883	.849	.978	.979	.816
45	12,643,943	145,004,144	.939	.087	.149	.862	.883	.851	.978	.979	.869
50	12,589,490	99,751,924	.935	.126	.219	.862	.883	.854	.978	.979	.910
55	12,483,836	61,875,611	.927	.202	.352	.862	.883	.858	.977	.979	.945
60	12,388,100	44,697,628	.920	.277	.487	.861	.883	.861	.977	.978	.961
65	12,257,864	34,835,370	.911	.352	.609	.861	.882	.863	.976	.978	.970
70	11,446,785	27,815,267	.850	.412	.704	.860	.882	.865	.975	.977	.975
75	11,214,691	24,099,357	.833	.465	.780	.859	.880	.866	.974	.975	.978
80	10,835,047	20,907,974	.805	.518	.837	.856	.878	.866	.971	.972	.977
85	10,596,654	18,880,603	.787	.561	.886	.853	.875	.865	.968	.969	.976
90	10,286,408	17,339,073	.764	.593	.914	.850	.872	.863	.964	.966	.973
95	8,761,446	14,656,269	.651	.598	.940	.846	.868	.860	.960	.962	.970
100	8,515,040	13,831,829	.633	.616	.950	.842	.864	.856	.956	.957	.965

Table 6.12.: Accuracy evaluation of the MinHash component (M^8_{64}) for matching thresholds T of 40 and above. Index C indicates that only FPs from different families are considered.

6.4. Third-party Library Usage and Code Sharing in Windows Malware

In this section, we now apply MCRIT in order to investigate third-party library usage and code sharing in Windows malware. We first introduce the data sets used for this purpose. We then verify that MCRIT does produce meaningful results when applied in

a practical setting. Next, we measure the presence of libraries in malware. Finally, we analyze and discuss the implications of this for similarity matching in malware.

6.4.1. Data Sets

In order to study library usage and code sharing in malware, we naturally need a selection of malware and library code.

For malware, we continue to use the same reference snapshot of Malpedia, as described in Section 4.3.4. However, similar to how we proceeded for the Windows API usage analysis in Section 5.3.1, we make a decision to exclude certain families with inappropriate characteristics. For consistency, we only use families with memory dumps available, reducing the number of families from 929 to 839. Again, we drop families that have been created with the .NET framework as they use bytecode instead of native x86/x64 Intel code as targeted by our disassembler and code similarity method. This affects a total of 113 families, which leaves 726 families for consideration. Finally, we decided to furthermore exclude families that have been written in Delphi and Go. Both of these programming languages have extensive collections of own library code that is typically statically linked into applications when used. Among the remaining, we identified 63 families for these languages. While having low representation among all families (less than 10%), we believe that they could still introduce a distorsion of the results as they are much less likely to use third party libraries due to their built-in library code. This leaves us with a final 663 malware families with 2.056 samples. These contain a total of 1,633,349 functions and 1,116,619 of them have 10 or more instructions.

For library reference code, we incorporate three sources. First, we reuse the 16 libraries from the Shift Media Project [315] data set that was used for the MCRIT evaluation and described in Section 6.3.2. Because we no longer require function labels, we can use even more code from this data set. A total of 449 library files provide 1,142,518 functions, 749,976 have 10 or more instructions.

Second, we use all the code from the `empty_msvc` project [319]. The data set is composed of the resulting programs when compiling "empty" projects (i.e. only a single `main()` routine that returns 0) in as many versions of Microsoft Visual Studio as possible, using both dynamic and static linking. The goal of this is to isolate the code that is introduced into these programs by the Visual C compiler and providing useful ground truth to detect it. It contains 118 configurations, which contain 112,481 functions out of which 44,699 have 10 or more instructions.

Finally, we collected a further 36 libraries for which we anticipate potential encounters in malware. Candidates for this selection have been determined by reviewing numerous analysis reports and related work by Alrabaee et al. [213] but are also based on the author's own experience. Almost all of these libraries were obtained as pre-compiled files, and many of them through the package manager vcpkg [320]. All 460 files associated with these libraries combined contain another 1,194,978 functions and 565,543 of these have 10 or more instructions.

In sum, the following experiments are conducted across 2,476,837 functions.

6.4.2. Accuracy Verification

In order to verify that MCRIT performs well in a more applied context, we examine its matching results for two libraries: MSVC and libzlib. We already detailed in Section 4.4.2 that most Windows malware is built using MSVC, which makes it a natural choice. The use of libzlib in malware is also well-documented, e.g. in [321] and [322].

As motivated before, source code for malware is typically not available and as a consequence, we have no information or ground truth for the presence of these libraries in malware. Therefore, we need to be able to identify them by other means. For this task, we created YARA rules to generically detect both libraries and ensured that they at least detect all representatives in our library collection while not causing any false positives. The rules are listed in Appendix B. Using these rules on the Malpedia data set, we assess 1,294 hits on samples for MSVC and 195 hits for libzlib.

Now, in order to detect the presence of the libraries in samples using MCRIT, we match all functions associated with the libraries on a per file basis against all malware samples and report the result for the best match. The best match in this sense is defined as the library that matched functions carrying the most bytes, each weighted with the function's matching score to incorporate this as a measure of confidence. To decide whether we consider a library present in a sample or not, we use the percentage of weighted bytes matched from the library as a second threshold. The results are shown in Table 6.13.

A first general observation is that when using MCRIT for compound detection of libraries, the system achieves a much better Precision than previously for individual functions. The primary reason for this is that the noise introduced by matches with low matching score has much less impact in this setting. This is because we now consider libraries matched against samples as a whole and larger accumulations of false positives between groups of functions are less likely as the results indicate, especially when increasing the matching threshold to 50 or above.

Similar to our previous analysis in Section 6.3.2, we see that for a matching threshold of 40, the achieved Precision falls significantly behind the results achieved for higher thresholds and the overall number of FPs also leads to very low Accuracy in the case of libzlib. For a threshold of 45, the number of FPs almost drops to about half for MSVC and even a third for libzlib while not significantly losing TPs. Going from threshold 45 to 50, the number of FPs is again significantly reduced but with a more notable impact on TPs. When increasing the matching threshold to 60, we see that FPs disappear almost entirely for libzlib, even at low required matching percentages. However, the TPR also falls behind compared to lower thresholds and we get no match with more than 20% of code for libzlib at this threshold.

Looking at the matching results for both libraries in more detail we can make several interesting observations. When linking libzlib into a program, a string pointing to the library's authorship, copyright but also version of the library is frequently included. We found this string for 176 out of the 195 cases where the YARA rule produced a hit. Looking at the version numbers, only 71 correspond to versions for which we have reference code in our data set (68x 1.2.8 and 3x 1.2.11). All other versions are below,

with the three most common being 1.2.7 (31x), 1.2.3 (27x) and surprisingly 1.1.3 (15x). For context, libzlib 1.1.3 was released in 1998, 1.2.3 was released in 2005, and even 1.2.7 was released already in 2012. Comparing this to Figure 4.3 from Section 4.3.4, which gives temporal information about the samples in the corpus, we note that these library dates are much older than the likely creation of the samples. This again points out that malware authors may tend to use aged and outdated software and versions, similar to what was observed to the distribution of MSVC versions found in the PE header's linker field (cf. Section 4.4.2).

This generally implies that when thinking of which code to include in a reference collection of libraries, one should definitely include coverage for even very old versions for optimal results. Then again, contrasting these findings with the library detection results, we are delighted to see that MCRIT is still capable of detecting a significant share of these much older libraries despite only having code of version 1.2.8 (relased in 2013) and above for reference.

We also had an in-depth look at the false positives. In most cases, especially for the lowest matching thresholds, the fuzziness of MinHash caused an aggregation of 2-3 function FPs, which was already sufficient to exceed the low expected byte percentage threshold and register as FP on library-level. These effects become much less frequent once the required code percentage is at 1% or 1.5% and almost disappears entirely when requiring a higher matching threshold.

However, we noted several cases where the occurrence of FPs had a different reason. In these, functions were recognized correctly as being part of libzlib but the sample was not previously identified by the YARA rule. By further investigation, we found that those were indeed single functions (e.g. the checksum algorithms Adler-32 or CRC-32) from libzlib that were however also the only traces from the library found in the malware. We think the most likely reason here is that the source code of these functions have been copied directly by authors into the source code of the malware. This explains why the YARA rule missed these samples, as neither the author or version strings nor the inflate table could be matched in such an isolated case.

This phenomenon raises a more general question how we have to expect libraries being included into programs. With options including full precompiled libraries, or as full source code package, or just manually chosen snippets of source code, this has significant implications about the detectability of these libraries as the above examples illustrated. Even with the first two cases, optimizations like Dead Code Elimination [323] or Function-Level Linking [324] could create situations where only a fraction of the library code will be present in the resulting compiled program, asking for a very fine-grained resolution of functions.

When looking closer at MSVC, we note that the variance in program sizes has notable impact on the FPs. The number of bytes in functions for libzlib range between 63,971 and 73,910 for all versions in the data set. This number fluctuates much more for MSVC, because the footprint of functions introduced by MSVC when linking dynamically is magnitudes smaller when compared to linking statically, especially for debug builds. We register between 681 (VS 2003 `/MD`) and 469,193 bytes (VS 2017 with `/MTd`) in this case, with a median of 16,323 and an average of 82,807 bytes. When only considering

dynamically linked binaries, this average drops to 5,439 bytes, and again, 1-2 mismatched functions contribute enough bytes to register a FP. Because of this low byte size, the functions to be mismatched can be very small themselves (10-15 instructions) and we noted before that matching these is prone to errors (cf. Section 6.3.2).

Examining the results in total, we consider a threshold pair of 50 for the matching threshold and 1.5% for matched library size as a good compromise for balancing TPs and FPs. This value combination achieves among the best overall Accuracy for both libzlib and MSVC, while slightly leaning towards Precision over Recall.

		MSVC					libzlib				
T	Percent	TPs	FPs	TPR	PPV	ACC	TPs	FPs	TPR	PPV	ACC
40	0.5	1,284	506	.992	.717	.738	185	688	.949	.212	.532
40	1	1,232	349	.952	.779	.791	182	305	.933	.374	.787
40	1.5	1,188	287	.918	.805	.800	179	244	.918	.423	.826
40	2	1,164	201	.900	.853	.832	174	218	.892	.444	.840
40	5	1,050	83	.811	.927	.834	125	81	.641	.607	.899
40	10	926	20	.716	.979	.803	64	2	.328	.970	.911
40	20	886	0	.685	1.00	.793	32	0	.164	1.00	.891
45	0.5	1,213	333	.937	.785	.790	179	205	.918	.466	.852
45	1	1,158	209	.895	.847	.825	168	76	.862	.689	.931
45	1.5	1,106	159	.855	.874	.824	163	34	.836	.827	.956
45	2	1,078	116	.833	.903	.831	152	17	.779	.899	.960
45	5	1,021	30	.789	.971	.846	101	2	.518	.981	.936
45	10	900	16	.696	.983	.792	47	0	.241	1.00	.901
45	20	884	0	.683	1.00	.792	26	0	.133	1.00	.887
50	0.5	1,147	256	.886	.818	.795	171	78	.877	.687	.932
50	1	1,088	118	.841	.902	.835	148	18	.759	.892	.956
50	1.5	1,056	69	.816	.939	.844	123	7	.631	.946	.947
50	2	1,048	25	.810	.977	.862	110	3	.564	.973	.941
50	5	995	11	.769	.989	.843	66	0	.338	1.00	.913
50	10	895	5	.692	.994	.795	35	0	.179	1.00	.893
50	20	883	0	.682	1.00	.791	3	0	.015	1.00	.871
60	0.5	1,026	61	.793	.944	.833	98	8	.503	.925	.930
60	1	1,008	25	.779	.976	.842	64	2	.328	.970	.911
60	1.5	998	14	.771	.986	.843	61	1	.313	.984	.909
60	2	996	5	.770	.995	.846	58	1	.297	.983	.907
60	5	979	5	.757	.995	.837	44	0	.226	1.00	.899
60	10	888	4	.686	.996	.792	15	0	.077	1.00	.879
60	20	866	0	.669	1.00	.783	0	0	.000	.000	.000

Table 6.13.: Detected presence of MSVC (out of 1294) and zlib (out of 195) in malware samples of the Malpedia corpus, using matching threshold T and the percentage of bytes from recognized functions in relation to the full library size as a second threshold.

6.4.3. Presence of Third-Party Libraries in Windows Malware

We will now proceed and measure the presence of all other libraries in the malware samples from Malpedia. Overall, we have matched 1,360,218 library functions from

1,117 library files against 1,116,619 functions from 2,056 malware samples. The results are shown in Table 6.14.

We first note that we generally register library matches for 533 (80.39%) families with 1,507 (73.30%) samples. In total 171,934 (15.4%) functions in malware have been recognized as libraries, with 20.28% of the samples' matchable code covered. This is well in line with the estimated 10-45% FOSS package usage reported by the only comparable study we know of, conducted by Alrabaee et al. [213].

Among the detected functions, 148,925 or 86.62% have been detected as one library only, with the three most matched libraries being MSVC (97,615), openssl (14,923) and wolfssl (12,555). For those functions being detected as multiple libraries, 12,543 or 7.30% have been detected with 2 library labels, 4,580 or 2.66% have been detected with 3 library labels, and 5,886 (3.42%) have been detected as 4 or more libraries. In these collisions, the most common pairings occur between libgnutls and openssl (1,613), libxml2 and openssl (1,027), and poco and tinyxml (783). As a further reason for collisions, we identified cases where libraries were simply included in other libraries, e.g. libzlib in libmariadb, or libbz2 and liblzma both being used by libarchive.

As expected, the most prominent library overall is MSVC with matches against 428 families, for which we also note the by far highest distribution of matched library content. The reason that we do not observe an even higher matching percentage throughout all percentiles is that the `empty_msvc` project only contains reference binaries compiled as executable (and not DLL) and also only as C and not C++ projects. This may create potential gaps in MSVCRT code coverage, as we observed many families being compiled as DLLs as reported in Section 4.4.2 and we also certainly have to expect C++ code.

Other frequently detected libraries are mostly centered around networking (e.g. poco), encryption (e.g. wolfssl), and general data processing (e.g. tinyxml, abseil). No occurrences were detected for three libraries focusing on sound processing (libsndfile, libspeex, opus), the regular expression library pcre2 and libcurl.

It has to be noted that these results should again be interpreted with two aspects in mind. First, a number of libraries contain files small enough that they may suffer from the same effect described previously for dynamically-linked MSVC (cf. Section 6.4.2), causing misdetections through just a few matched small functions.

Second, the reference library data has been collected in 2020, with many of the library versions being among the most recent available. We already noted before that these may be signficantly more modern than what is used in malware (cf. Section 6.4.2), which may cause a detection gap.

We therefore conclude that our estimated library use of 20.28% code is more likely a lower than an upper border. This conclusion is additionally supported by findings in the following section, in which we view function match clusters in the context of family labels.

6.4.4. Code Sharing in Windows Malware

We now extend the perspective to match code across all malware families, taking the previously detected library functions as additional information into concern.

			Matchable Bytes					Percent Matched				
Library	Files	Functions	min	50%	max	Families	Samples	min	25%	50%	75%	max
abseil	109	8,791	30	3,004	101,823	116	238	2.69	7.92	26.76	33.56	94.55
aplib	15	202	36	7,071	9,371	23	68	3.22	11.02	16.98	76.17	92.21
boost	36	43,387	59	82,498	920,522	83	183	2.27	3.51	5.88	11.80	24.71
c-ares	2	389	57,775	68,405	68,405	2	3	2.79	2.89	3.00	4.12	5.25
double-conversion	2	317	41,053	48,730	48,730	1	1	3.77	3.77	3.77	3.77	3.77
expat	2	570	83,841	102,038	102,038	15	21	2.67	2.83	3.17	3.91	29.44
freeglut	2	732	81,267	98,636	98,636	29	62	2.80	3.03	4.60	8.80	14.77
freetype	2	2,952	447,151	507,114	507,114	8	14	2.73	3.15	4.01	4.13	5.53
grpc	22	68,641	2,498	332,099	2,116,389	72	176	2.03	3.58	4.93	7.28	23.77
harfbuzz	2	12,429	1,028,413	1,129,158	1,129,158	7	26	2.90	3.73	3.82	3.88	7.70
jasper	2	1,449	207,451	247,071	247,071	20	55	2.74	2.92	3.71	3.86	6.36
libarchive	8	11,195	387,918	434,346	454,358	13	33	2.74	3.53	3.86	4.02	6.04
libbz2	8	425	39,581	45,977	54,872	6	10	4.03	4.69	9.02	46.60	50.44
libcurl	2	2,513	603,403	711,812	711,812	0	0	-	-	-	-	-
libenca	10	773	16,060	20,047	20,898	14	23	2.67	2.81	3.16	3.65	6.95
libflac	4	1,232	9,859	181,263	182,749	163	367	2.31	3.77	5.74	8.69	28.81
libgcrypt	48	53,098	398,092	439,985	490,298	87	187	3.02	3.95	5.58	9.87	24.92
libgmp	16	13,112	234,459	320,533	370,025	73	168	2.40	3.26	4.01	4.94	8.24
libgnutls	52	136,434	624,361	748,151	1,034,480	23	60	3.08	5.12	5.72	12.04	16.74
libgpg-error	60	15,646	39,458	62,459	83,571	20	41	2.32	3.24	4.88	5.37	12.43
libiconv	16	5,150	70,781	83,546	87,469	29	40	3.07	3.53	4.40	6.88	14.29
libjpeg-turbo	4	2,702	314,459	363,379	390,180	5	9	2.79	3.17	3.17	6.92	9.80
liblzma	26	7,744	87,205	92,446	106,678	14	25	2.31	2.49	2.92	3.02	6.48
libmariadb	2	1,439	187,267	211,510	211,510	39	88	2.50	3.26	3.96	7.36	10.46
libnettle	72	24,014	116,797	119,944	139,928	87	157	2.47	3.88	6.41	8.60	35.11
libogg	22	1,089	7,060	7,941	8,561	20	54	2.40	4.00	4.00	4.35	7.01
libpng	2	843	115,248	143,768	143,768	17	30	3.26	4.66	5.26	5.33	7.30
libpq	12	1,508	3,996	36,329	100,890	111	261	2.66	6.15	8.61	8.61	29.08
libsndfile	2	2,054	357,782	398,272	398,272	0	0	-	-	-	-	-
libspeex	14	1,971	86,984	118,378	124,292	0	0	-	-	-	-	-
libssh	54	41,553	172,814	208,649	256,605	24	52	2.70	3.11	5.20	6.44	19.13
libvorbis	6	547	10,339	21,124	138,287	5	34	2.86	3.77	3.77	6.13	6.32
libwebp	8	4,384	9,006	209,553	486,856	25	59	2.77	3.90	5.22	6.10	7.56
libxml2	24	61,313	740,382	924,399	943,157	25	65	2.84	3.53	3.81	6.26	10.74
libzlib	20	3,817	46,568	67,885	73,161	50	123	2.84	3.49	11.67	23.91	37.24
lmdb	2	296	49,010	61,597	61,597	5	6	2.95	3.02	3.16	3.35	4.56
mcpp	2	326	64,366	75,177	75,177	1	1	2.94	2.94	2.94	2.94	2.94
mdnsresponder	2	133	17,543	20,257	20,257	13	17	2.75	3.02	3.53	3.98	6.61
MSVC	118	44,699	544	17,730	442,677	428	1,125	1.87	34.41	87.01	89.81	98.48
openssl	100	383,893	264,781	1,072,110	1,548,783	35	83	2.77	4.24	6.11	33.63	65.70
opus	2	962	268,842	334,281	334,281	0	0	-	-	-	-	-
pcre	10	1,809	1,130	225,312	286,070	44	92	2.53	3.25	4.96	7.45	16.70
pcre2	6	1,829	252,446	320,065	379,564	0	0	-	-	-	-	-
poco	32	50,278	5,741	153,381	839,983	228	574	2.88	16.92	16.92	16.92	49.20
protobuf	6	59,588	372,904	2,369,427	2,684,741	46	120	2.63	3.10	3.61	6.06	11.15
qt5	80	164,407	287	119,804	3,898,923	185	404	1.53	7.12	8.34	17.07	100.00
sqlite3	2	3,776	685,293	845,807	845,807	10	14	2.90	3.40	6.82	30.94	40.48
tiff	4	1,308	2,236	223,757	247,602	68	128	2.63	3.69	4.22	12.99	16.17
tinyxml	2	518	22,550	38,357	38,357	216	495	1.94	3.39	5.67	10.75	60.07
tinyxml2	7	2,020	27,228	29,296	34,098	81	145	1.98	3.37	4.53	12.75	57.43
upb	12	1,290	1,505	15,676	49,747	87	259	2.21	3.37	3.92	5.58	15.87
wolfssl	8	11,473	166,882	374,895	512,771	136	298	2.53	4.06	5.94	10.57	55.24
zeroc-ice	32	102,481	7,858	191,860	3,380,278	84	171	2.58	3.85	6.78	10.84	27.25
zstd	2	2,331	567,691	656,595	656,595	1	1	3.24	3.24	3.24	3.24	3.24
Combined	1,117	1,360,218	-	-	-	533	1,507	-	-	-	-	-

Table 6.14.: Detected presence for 53 third-party libraries. For each library, the number of corresponding LIB files and functions equal or above 10 instructions is listed. *Matchable Bytes* provides additional information about the range of sizes for the LIB files, *Percent Matched* gives a five number summary for how much content of the libraries was matched.

For this, we conducted the previous experiment in the other direction, matching 1,116,619 functions from 2,056 malware samples against themselves in addition to the 1,360,218 library functions from 1,117 library files.

Overall, we register 244,187,596 MinHash matches when using a similarity threshold of 50, out of which 55,336,402 (22.66%) are also PicHash matches. On a per sample level,

matching takes between 0.97 and 2,522.19 seconds, with a median of 25.49 seconds. The whole data set was processed in 35 hours and 27 minutes.

Figure 6.4 shows the distribution of function cluster sizes when measured in families represented in the cluster, annotated with whether or not the cluster has been classified as a collection of library functions. This view on the data can be used to provide an interpretation in terms of occurrence frequency with which similar code is encountered across malware families. It is inspired by similar concepts presented by Kornau-von Bock und Polach et al. [325] for large-scale code similarity analysis within Google.

Because clusters may contain a split of functions recognized as library or not, we decide their belonging based on the percentage of the functions that had a library label. Here, we deliberately chose 10% as a threshold and expand the respective label to the full cluster if the threshold is met. As a result, 217,266 functions have transitively received a label in addition to the 171,934 that were already labeled directly before (cf. Section 6.4.3). With increasing cluster size, the fraction of clusters tagged with a library of any kind generally increases, with MSVC dominating over other libraries for cluster sizes of 25 families and larger. MSVC is generally the most commonly found library, which is a similar result to what we previously observed (cf. Table 6.14).

We can see that there are several peaks in the overall distribution, best visible e.g. for cluster sizes of 43, 103-104, 263 etc. families. Most of them occur for MSVC tagged clusters and for some of them, their surrounding exhibits a shape reminding of a gaussian bell curve. Our analysis did not give a concluding answer for this phenomenon, but we found indications that these correspond to groups of common code shared across version and compilation options of MSVC. The bell curve in this context is a result of the fuzziness introduced through MinHash.

In total, 727,419 functions are neither matched directly nor transitively with the reference libraries in the data set. The stacked bars for cluster sizes 9 or smaller are cut off for better interpretability of the diagram, excluding the portion of functions not matched with any library. Note that the functions not shown in the diagram constitute a significant part of all functions and sum up to 558,334 or 50,00% of the functions in all malware samples, with a majority of them concentrated for 0-3 families matched (305,796, 104,497, 51,015, 29,045), which together add up to 43.91%. These also most likely represent the family-intrinsic parts of the malware families and are the area where we can expect to find the most interesting overlaps between families.

It is safe to assume that despite our efforts, the collection of reference library code will have missed libraries or versions of libraries that are commonly found in malware. As a result, we expect the real number of larger cluster sizes not associated with any library to be smaller than suggested by Figure 6.4 and again assume our results more likely to represent a lower bound. This is a particularly interesting observation, as it suggests that code found so commonly across families carries less value when investigating authorship links between families and that less frequently observed overlaps could in turn be amplified.

Next, we aggregate the individual function matches to measure similarity on a sample level. As a similarity measure, we summarize the size in bytes of matched functions

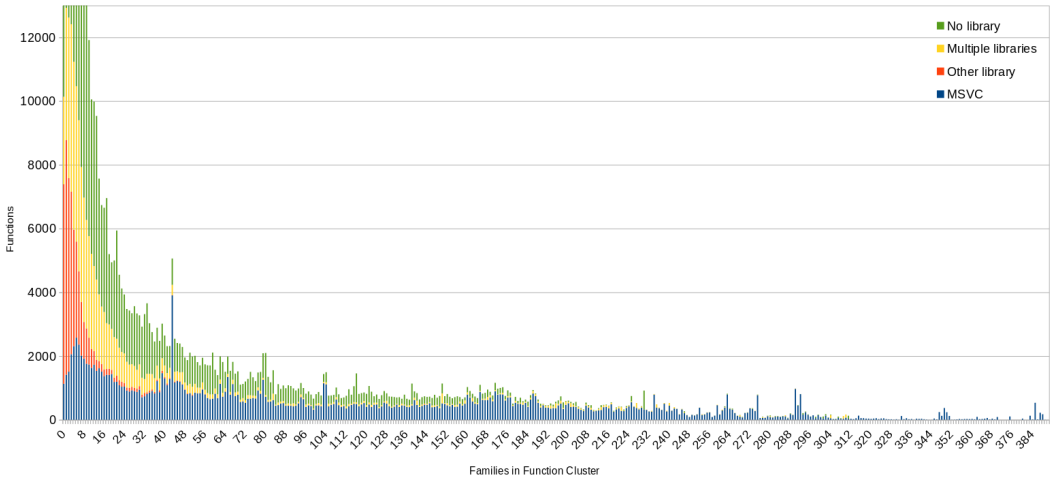


Figure 6.4.: Distribution of function cluster sizes when counted in families, annotated whether or not detected as a third-party library. Bars for cluster sizes 9 and smaller are cut off for better interpretability, excluding only functions tagged "no library".

multipled with the similarity score. This takes the actual amount of code matched into concern (opposite to e.g. counting functions only) and also the confidence with which matches have been assessed.

Additionally, we make use of the previously stated observations. On the one hand, we exclude all matches in clusters that have been identified as library code. This allows us to focus on the parts in the code that we assume to be family-intrinsic. This idea has been proposed before as well by Tahan et al. [194], who used it on n-grams of full-scale binaries instead of for the comparision of individual functions as in this work.

On the other hand, we use the number of families found per cluster to further adjust the scoring. For this, we scale the byte score by dividing with $\log_2$ of the number of families contained in the cluster, if this number is 4 or above. This ensures that clusters shared among few families are counted in full while bigger clusters are softly diminshed according to their size.

Figure 6.5 shows the results of an exemplary similarity clustering using a threshold of 10% between samples. Samples of the same family have been assigned the same random colors and a number of interesting cases have been annotated with numbers. The overall clustering results in 746 components with 6,237 edges. At first glance, we see that the links lead to mostly homogeneously colored clusters, indicating that code similarity and thus relationship between samples of same families is reliably recognized. At the same time, there appear to be only few cases where code between families is overlapping for 10 or more percent.

Note that the 746 components capture 663 families, meaning that a number of families have been divided into multiple clusters as well. Reasons for this are for example that some are divided by bitness, have changed too significantly in the versions captured in the data set, or because they contain multiple modules (loader and payload). We will

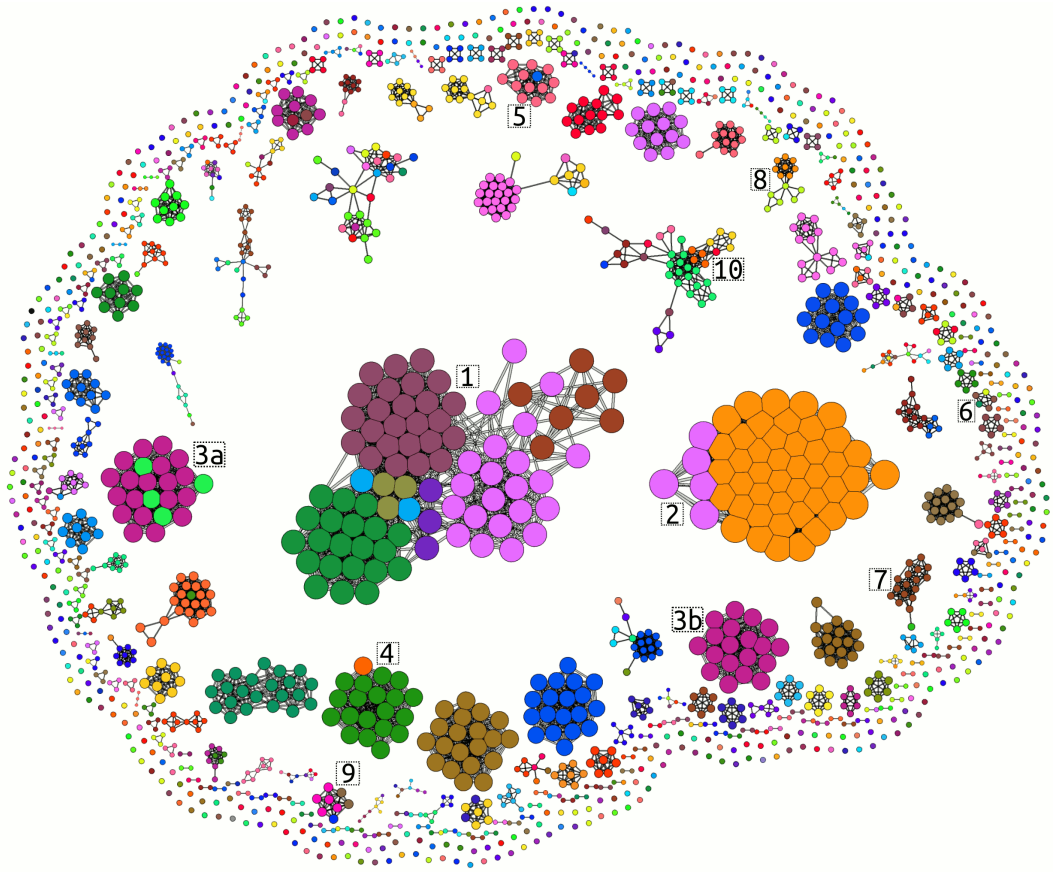


Figure 6.5.: Similarity clustering resulting from aggregating all function matches on sample level. Nodes represent samples (same color for same family) and edges indicate a similarity score (measured in bytes) of 10% or above.

now discuss a number of 10 selected, interesting cases to demonstrate the capabilities of the methodology implemented with MCRIT.

Cluster 1 is the biggest one and contains 83 samples. It contains 7 families all derived from the Zeus source code leak: `win.citadel` (plum), `win.kins` (green), `win.ice_ix` (cyan), `win.zeus_sphinx` (purple), `win.vmzeus` (pink), `win.floki_bot` (brown), and the source family itself, `win.zeus` (ochre). The highest node degree with 43 has `win.zeus` version 2.0.8.9, the leaked version.

Cluster 2 is another big cluster and consists of two Zeus-like families: `win.vmzeus` (pink) and `win.pandabanker` (orange). The samples of `win.vmzeus` are not linked to the ones from cluster 1 because a WinAPI usage obfuscation scheme has been added that alters the majority of functions structurally to a degree that they are no longer matched by MCRIT. In this case, it is this specific obfuscation scheme that links the evolved strain of `win.pandabanker` to `win.vmzeus`. This has also been observed by IBM X-Force [326].

Cluster 3a and cluster 3b are both `win.dridex` (berry), which cleanly divide into 32bit and 64bit respectively. The samples colored green in cluster 3a are also related, as they belong to the ransomware offspring of `win.dridex`: `win.friedex`.

Cluster 4 is the primary collection of samples for family `win.locky` (green). The orange sample is indeed associated with the code base, as it is `win.locky_decryptor`. In this case, the decryptor component is implemented reusing the code for enumerating files and handling cryptography from the ransomware. Note also that a second, smaller cluster of `win.locky` is found above the marking for cluster 6, which contains the later samples that employed heavy code obfuscation.

Cluster 5 consists primarily of samples from family `win.gpcode` (pink), which is manually written in Assembler. A consequence of this is that it contains many characteristic code sequences that look differently from what compilers would emit. The sample colored in blue belongs to family `win.crypto_fortress` (blue), which appears to have been an attempt at rebranding the ransomware but clearly has strong code overlap upon manual inspection.

Cluster 6 consists of 3 families: `win.sage_ransom`, `win.crylocker`, and `win.ransoc`, that are as well documented to be rebrandings of each other [327].

Cluster 7 shows the strong relationship of samples from `win.isfb` (brown) among each other but also links to a sample of its documented predecessor `win.snifula` (green) [328].

Cluster 8 captures the relationship between `win.pony` (orange) and `win.icedid` (lime). This was also observed and documented by Intezer [329].

Cluster 9 contains the evolutionary linage of `win.murofet` (brown), `win.gameover_p2p` (navy), and `win.gameover_dga` (magenta). While all three families are attributed to the original author of `win.zeus` [330], their code links to the zeus cluster (1) are distant and diminished by the $\log_2$ scaling described earlier. On the other hand, the newly introduced parts provide a strong link among the families.

Cluster 10 shows the effect of incomplete library recognition and filtering. Multiple of the families in this cluster have much older versions of OpenSSL than in the reference data set and/or the uncovered VCTools statically linked into them. This causes MCRIT to detect suspected overlap in these uncovered functions, e.g. OpenSSL causing matches between `win.xtunnel` (mint) and `win.zeus_openssl` (orange), as well as `win.xtunnel` and `win.rokrat` (berry). Neither of these relationships makes any sense from the known background in which these malware families are situated, so we can register them as false positives.

We assume that there are more false positive multi-family clusters that have been composed for similar reasons but we also think that this approach may uncover interesting leads of potentially unknown connections between malware families. Overall, we think that the results are plausible and that this experiment demonstrates MCRIT's powerful clustering abilities.

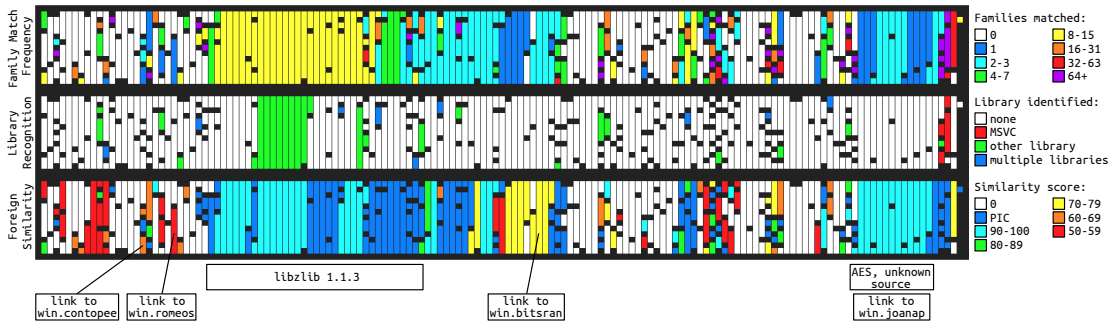


Figure 6.6.: MCRIT graph for the analyzed `win.wannacryptor` sample that lead to Lazarus being attributed for the attack. Additional annotation below the graph shows further links to Lazarus as identified by MCRIT. The sample contains 216 functions with 10 or more instructions.

6.4.5. Case Studies for the Application of MCRIT

In this section, we now outline practical use cases for the application of MCRIT. We have selected one case each with an APT and crimeware background.

In the first experiment, we will reproduce the analysis that was conducted in order to generate attribution hints for `win.wannacryptor` (or more commonly referred to as "WannaCry") towards the North Korean threat actor group Lazarus.

In the second experiment, we will examine how the banking malware `win.citadel` has evolved from the leaked `win.zeus` source code, outlining which components have been added or changed and which remained closer to the original.

Both case studies exemplify how MCRIT can aid and accelerate in-depth malware analysis workflows [38] by guiding an analyst's attention to code areas of interest. We again use the same data sets as before and a similarity threshold of 50.

Attribution of WannaCry

On May 12th 2017, news broke of a ransomware spreading rapidly by employing worm-like distribution, using an SMB exploit called EternalBlue [331], which was part of the NSA arsenal previously leaked by TheShadowBrokers. On May 15th 2017, Neel Mehta from Google tweeted [332] two hashes for malware samples with corresponding offsets that pointed out a link in code between `win.wannacryptor` and malware family `win.contopee`. This other malware family was previously observed being used by the threat actor Lazarus [333], credited e.g. for attacks on the SWIFT banking system [333] and the wiper attack on Sony [334]. The code sharing link was then quickly verified by other researchers [335, 336]. In this experiment, we will reproduce the findings published originally in the tweet using MCRIT.

Figure 6.6 shows a visualization we refer to as MCRIT graph, a composite diagram for matching results of the `win.wannacryptor` sample that was referenced. All three rows in the diagram are overlays of each other and provide different aspects of matching

information. They generally visually represent all functions in the sample with 10 or more instructions. For each function the respective size is indicated by length of its bar, with one block being equal to 10 instructions and expanding first top down and potentially wrapping over into the next column. All functions are listed in the sorting of their start addresses, moving left to right.

The first row picks up the occurrence frequency interpretation introduced in Section 6.4.3 and shows to how many other families this function was matched. The second row indicates whether a function was matched against a library function from the data set and we differ between a function from MSVC, one library or if multiple libraries were matched. The third row shows the best similarity score of a function not belonging to a sample of the same family, i.e. foreign similarity. The rating PIC means in this context that the function had a MinHash similarity score of 100 and also matched using PicHash and thus can be considered a binary clone.

As shown in the diagram, MCRIT is indeed capable of establishing the same link to `win.contopee` and the function in question is matched with a score of 67.19. In addition, we annotated further code sharing links to families attributed to and exclusively used by Lazarus as found by MCRIT. MCRIT finds another function related to the construction of the same imitated TLS encapsulation in the `RomeoGolf` variant of family `win.romeos`. Further inspection of the sample reveals that the same code linked to `win.contopee` is also found there but was not recognized due to code being merged and additional use of string obfuscation. Another piece of code that makes use of the embedded libzlib 1.1.3 (also partially recognized by MCRIT) is linked to family `win.bitsran`, which uses the same version of libzlib but in a slightly different but still matchable way. Finally, at the end of the `win.wannacryptor` sample we find an implementation of AES for which we could not identify the source. However, the exact same functions are found to be used in family `win.joanap`, which is also linked to Lazarus.

An important takeaway of this example is the insight that all of these links are established through functions matched against single or very few other families (1-3), i.e. functions with low occurrence frequency. We believe that a selection of functions with these characteristics may serve well as a list of pivotable candidates when searching for code-based relationships between families.

Mapping out Citadel

In the second experiment, we will dive into the ecosystem of families derived from the source code leak of `win.zeus`, focusing on `win.citadel`. This family was very popular and actively developed during 2012-2013.

Figure 6.6 shows the MCRIT graph for a sample of `win.citadel`, version 1.3.4.5. In addition to the three rows, we have annotated the functional capabilities of code areas in the binary as manually obtained through reverse engineering and added their averaged foreign similarity score. Looking at the first row of the graph, we can see that the majority of functions is matched with 4-7 and then 8-15 families. This is in line with the observations made in Section 6.4.3 and underlines the popularity of `win.zeus` as a baseline for derivative projects. The green areas directly correspond to the 7 families

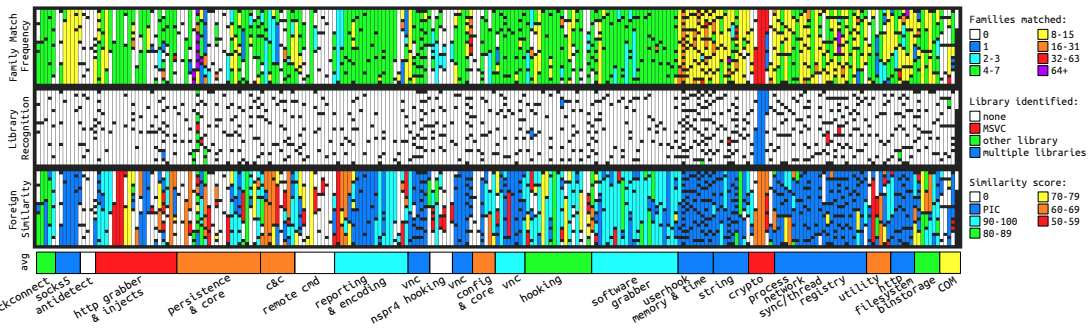


Figure 6.7.: MCRIT graph for an analyzed `win.citadel` sample, version 1.3.4.5. Additional annotation below the graph shows which code areas are responsible for which functional aspects in the malware, color indicates the averaged similarity score in the area. The sample contains 747 functions with 10 or more instructions.

mentioned for Cluster 1 (cf. Figure 6.5). The yellow area additionally links into other zeus-like families such as `win.murofet`, `win.pandabanker`, or even `win.bolek`.

With respect to libraries matched, we can see that the reference library set did not produce significant hits, apart from one large function in the cryptography area. This function is also matched into many families (32-63) and provides AES encryption and decryption. It has a comparatively low matching score because it was slightly modified by the author of `win.citadel`.

When looking at the foreign similarity score, we can easily identify which areas were significantly modified by the malware author. These also overlap with the descriptions made in the manual published for this bot [337]. First, the unmatched (white) areas are a series of anti-detection mechanisms that also identify security products potentially running on the victim's machine. The remote command capability was heavily extended and hooking for the network library `nspr4.dll` was added, which is e.g. used by FireFox.

For the areas with lower matching scores (red, orange), HTTP grabber and injects were constantly updated during the activity of `win.citadel` to keep up with changes introduced through browser updates in Internet Explorer, FireFox, and Chrome. Similarly, the author seemingly had a liking for integrating customized cryptography, which lead to a heavily modified RC4 (using a more complex and salted SBox generation) and a lesser modified implementation of AES with an additional XOR operation using a static 128 bit key. The configuration, persistence, and C&C code (including parts of the reporting) was also modified and diverts from standard `win.zeus`.

Lesser modifications were made on the backconnect functionality, hooking and software grabbing. Almost no changes were made to Socks5 proxy and VNC capability and many of the core aspects of system interaction such as process, memory, registry, network and file system management were almost not changed at all. As the MCRIT graph shows, especially the latter have been reused in a wider array of families because of the convenience provided by them.

This second case study illustrated how MCRIT can be used to investigate and understand the composition of a binary and how to potentially isolate family-intrinsic aspects

of certain functionality. This again may serve as a supporting guideline in the context of situational awareness, e.g. when an analyst considers where to focus analysis efforts on or even just to help identify potential code reuse across families in a more widened context than classical 1:1 binary diffing.

6.5. Summary

In this chapter we covered two topics: robust recovery of code from memory dumps and measuring structural code similarity.

Robust code recovery is a strict requirement for enabling the reliable execution of a wide array of in-depth analysis methods. We continued again with concentrating on memory dumps as input format, which we identified as a favorable representation for unpacked malware (cf. Section 4.3.1). Personal experience from practical settings suggested that disassemblers struggle with handling memory dumps but as this had not been previously addressed in academic research of disassembly methods, we set RQ_6 specifically to focus on the recovery of code and Control Flow Graph information for Intel x86/x64 from memory dumps without making assumptions about structural properties of the given file. As an answer to this question we presented SMDA, a methodology that incorporates various core ideas from previous work [178, 179, 172, 175, 170, 31] and showed their applicability in this specific case. SMDA is organized in two phases, first identifying potential function entry points by heuristically locating code references on function level and then second, filling in gaps between the functions recognized in the first phase.

We showed that the introduced method of FEP discovery alone is capable of finding up to 79.81% of function starts (cf. Section 6.2.1) and that having two or more code references pointing to the same location indicates almost always a function as the precision of 0.996 for this case underlines. For the accuracy evaluation, we reused the data sets from Andriesse et al. [31] and Bao et al. [175] to enable comparability. Benchmarking recent versions of industry disassemblers IDA (7.4) and Ghidra (9.1.2), we found that their results on the Linux binaries significantly improved since 2017, and were pleasently surprised that Ghidra produced very accurate results. We evaluated the disassemblers on the Windows binaries from the Bao data set both in their given and memory-dumped form. This direct comparability allowed us to formally prove the experienced decline in performance for the other disassemblers. SMDA was the only disassembler to retain a high accuracy on memory dumps with a TPR of 0.958, which was possible without processing any structural information provided by file headers or similar sources.

In the second part of the chapter, we directed our attention to measuring code similarity in order to answer two more research questions: RQ_7, asking about the frequency of third-party library use in Windows malware and RQ_8 asking about actual intrinsic code overlap in Windows malware.

To answer these questions, we presented MCRIT as a system combining two methods for exact and locality-sensitive hashing of functions to enable efficient one-to-many code similarity analysis. We used the exact position-independent code (PIC) hashing method

defined by Cohen and Havrilla [187], primarily to study how function size affects the results when using function names as labels for assigning pairs of supposedly similar functions. Using a data set of 16 well-known libraries in various versions containing 820,922 functions, we found out that small functions of less than 10 instructions will frequently have exact code overlap but differ in their labels, which will cause challenges to the interpretability of precision. We concluded that excluding these small functions is a viable approach as they only contribute 2% of the code measured in bytes while contributing 31% of the functions.

We then performed an evaluation and selection for metrics- and token-based features to be used as representation for code in a MinHash-based similarity method. Further evaluating parameterizations for this MinHash method, we showed that b-bit hashing as suggested by Li and König [312] can be effectively applied, which allowed us to select a small MinHash signature length of 64 bytes (about 20% of the average function size) and still maintain good matching results, especially when considering the mean average precision, i.e. quality of results when considered from a usability perspective.

To answer the research questions, we applied MCRIT to a data set composed of 53 libraries and 663 families from Malpedia, consisting of 2,476,837 functions in total. With regard to RQ_7, our experiments suggest that 15.4% of the functions or 20.28% of the code in the malware samples are associated with library code from our reference data set. This is in line with the findings by Alrabaee et al. [213] who found between 10-45% FOSS package usage. However, we expect our result to be a lower border because the characteristics of the function matching clusters suggest that there remains a notable amount of functions that would have been detected as library if reference code had been available.

Investigating intrinsic code overlap as questioned by RQ_8, we found that only few families show linkage above 10% and in many cases this is a result of one family's source code being publicly available e.g. through a leak. Using MCRIT to study the case for code-based authorship attribution in `win.wannacryptor` however suggests that solid links can be much smaller, which opens field for interesting and challenging future work.

7. Summary and Outlook

In this dissertation, we concentrated on improvements to the efficiency and quality of Windows malware analysis. We identified ground truth and situational awareness as major challenges for even better analysis and workflows.

To address ground truth, we first reasoned about aspects that qualify ground truth for malware research and created a reference data set called Malpedia. Focusing on situational awareness, we continued with an examination of malware interactions with the Windows API, proposing a robust method for the extraction and comparison of API usage profiles. Finally, we improved disassembly quality on memory dumps and used code similarity analysis to explore third party library usage and code sharing across malware families.

All of these facets constitute enhancements to the tasks of classification, characterization, and contextualization, which are of high relevance for the in-depth analysis of malware. Especially supporting classification with relationship indicators and additional cross-family context during an investigation has the potential to save significant amounts of time and thus accelerate and facilitate an analyst's workflow. Apart from this, a key novelty of our work is the scale in terms of manually verified family coverage at which these analyses have been conducted, thanks to the comprehensiveness of Malpedia.

7.1. Summary of Contributions

With respect to the individual chapters, we summarize our contributions as follows.

In Chapter 4, we assessed the state of ground truth available for research using static analysis on Windows malware. Having identified an insufficiency in existing data sets, we defined a total of eleven aspects grouped into three major requirements: Representativeness (REQ_R), Accessibility (REQ_A), and Practicality (REQ_P). These requirements were then corroborated by a comparison to Rossow's Prudent Practices [50]. As a result, a data set fulfilling these requirements is expected to prove useful for research from an academic point of view as well as being able to support applied research and investigations in the context of practical malware analysis. Next, we created Malpedia as a reference malware corpus and validated that it indeed adhered to all requirements. With 1,136 manually verified malware families (January 2019) and its focus on unpacked representations in the form of preferably memory dumps, it is the most comprehensive openly available data set of its kind. Using Malpedia, we then performed a structural analysis across unpacked samples for 839 Windows malware families. This particularly revealed that meta data in the form of PE headers is widely available to analysis and also appears plausible in most cases.

In Chapter 5, we studied malware interaction with the Windows API in depth. As a prerequisite to analysis, we proposed ApiScout as a method for the robust extraction of Windows API usage information from memory dumps. ApiScout is a generalization of Eureka's approach [105] and uses two phases: first a system-wide inventorization of available WinAPI functions and their offsets, and then scanning and filtering of references to these in memory dumps. This decouples its application from the dynamic analysis environment, allowing for a more flexible and repeatable analysis. An accuracy evaluation revealed that comparable approaches (Scylla's IAT Search and Volatility's ImpScan) make too strong assumptions about how API references are structured and located, which may lead to incomplete results for them, while ApiScout achieves near perfect results. Next, we applied ApiScout on all memory dumps from Malpedia, showing that dynamic API imports are frequently encountered while custom obfuscation that actually impairs analysis is rare. A frequency analysis of individual WinAPI usage showed that most API functions are found in only few families, suggesting that usage profiles are distinctive on a family level. Furthermore, categorizing API functions into semantic classes gave insight into potential capabilities enabled by them, establishing behavorial context. We then introduced ApiVectors as a data representation that allows efficient storage and comparison of API usage profiles. An evaluation of this approach pointed out that API usage profiles allow effective malware classification.

In Chapter 6, we shifted our attention to the analysis on the code level. In a first part we proposed SMDA, an approach specifically tailored for the recovery of x86/x64 disassembly from memory dumps. The method distinguishes itself by not making assumptions about structural properties of a given input file and at its core is a combination of well-proven previous work [178, 179, 172, 175, 170, 31]. In the evaluation, we compared SMDA against the industry standards IDA and Ghidra. We noticed significant improvements of their performance on Linux binaries compared to previous evaluations [31] but were also able to verify a personal observation made over the years during practical analysis, namely that those disassemblers experience a notable drop in accuracy when presented with memory dumps. We showed that SMDA was not affected by this, as it maintained a TPR of above 95% and in the course also established that the method used for function Entry point identification is highly accurate. In a second part, we focused on code similarity. We first revisited the hashing method for position-independent code as defined by Cohen and Havrilla [187]. We used it to show that small functions with less than 10 instructions are prone to collisions with regard to their function labels, which implies challenges to the creation of ground truth and the interpretability of precision in evaluations. Next, we proposed a MinHash-based fuzzy code similarity method called MCRIT for which we conducted an extensive feature evaluation. Using MCRIT on Malpedia and a set of 53 popular third-party libraries, we identified that 15-20% of code found in malware likely originates from libraries and is not intrinsic to the families. When excluding such library code from consideration, we notice that malware families barely exceed 10% similarity, except for known cases of source code leaks or documented family relationships due to same origin. This implies that an approach for hunting similarity based on occurrence frequency of code across malware families and samples can provide valuable hints for identifying potential relationships.

In summary, we are convinced that this dissertation has effectively demonstrated the need for and benefit of having a representative data set available for in-depth malware research. The contributions made throughout this thesis draw a differentiated and detailed picture of the overall malware landscape from a binary code analysis point of view, indicating that there is still a lot more to explore and improve upon.

7.2. Conclusions

We draw the following primary conclusions from our research:

- It is feasible to curate a data set that enables comprehensive, comparative research on malware analysis methods and malware itself. While malware packing leads to an explosion in unique observed files, a high data redundancy can be assessed with regard to actually unique payloads. Packers also appear as a distinct barrier and ecosystem of their own, leaving payloads widely unaltered with respect to their meta data and content. Automated memory dumping is an effective approach for obtaining an approximate representation of unpacked payloads, helping to reduce the problem scale.
- Many malware authors seem to intentionally walk past new development tools if keeping e.g. an older version of Visual Studio they are used to promises higher compatibility with victim systems and stability in their results. Many authors also appear careless about data fragments that shed light on their working environments, expressed in information such as debug data, build system paths, and genuine compilation timestamps.
- It is possible to reliably extract Windows API usage information from memory dumps for more than 95% of the studied malware families, despite dynamic imports of API references being way more prevalent than previously documented in the literature. An interpretation of the WinAPI usage profile can serve as a good first estimation of the potential capabilities of a malware sample. The degree to which these WinAPI usage profiles are characteristic to families also implies that they capture personal choice and style of malware authors on how to use the Windows API to implement capabilities.
- The presence of statically-linked code from programming frameworks such as Delphi or Go has notable impact on both WinAPI usage profiles and code similarity. This underlines the relevance of methods that are able to isolate such generic from project intrinsic code. At the same time, the variety of code in third-party libraries and how these libraries are concretely used and linked remains to pose a significant challenge to analysis procedures.
- Our observation that no excessive code sharing across malware families exists in the absence of published or leaked code is promising. It suggests that families generally contain significant unique regions of code that enable effective classification, while actual overlaps invite to investigate whether previously unknown relationships between malware families may exist.

7.3. Practical Impact

At the time of writing (January 2022), Malpedia has become an established information resource valued by both academic and industry researchers and practicioners. Its closed community has grown to more than 1.700 members and more than 12.000 commits have been added to the underlying Git repository. The corpus now covers 2.288 malware families represented by 6.365 samples. The web service making the data set accessible on average handles about 250.000 requests by more than 4.000 unique visitors daily. Malpedia is frequently mentioned as a useful resource by experts [338, 339, 340] and has been integrated with several other services and frameworks, including MISP [98], OpenCTI [341], TheHive [342], VirusTotal [238], UnpacMe [343], URLhaus [344], and malwoverview [345].

The provided open-source implementations of ApiScout [269] and SMDA [290] have been integrated into popular malware analysis tools, including the malware processing framework AssemblyLine [346] and the behavorial capability analysis tool capa [157]. MCRIT [309] has been used to support law enforcement investigations. This underlines that the theoretical foundations for the research presented in this dissertation are well suited for and have significant relevance for practical applications.

7.4. Future Work

We identify the following avenues for future research and effort:

Maintainance and extension of Malpedia: In order to remain perspectively useful and relevant, it should be ensured that Malpedia is actively maintained and its family coverage is kept up to date. This is in line with REQ_R and REQ_P as defined in Section 4.2.1, which demand temporal coverage and topicality for an effectively useful data set. Keeping a community involved with this task appears advisable and has been aspired since Malpedia's inception. In the future, it may become necessary to address potential shifts in the malware landscape e.g. modularized malware being divided over several deployment stages or if there is movement towards higher level languages including scripting languages like Powershell and Javascript. If required, this should be reflected in adjustments to the hierarchical data organization.

Extended analysis of WinAPI usage: Chapter 5 has successfully demonstrated how the analysis of WinAPI information can be used to discriminate between malware families. This could be expanded to the consideration of benign software, allowing to further study if and how WinAPI usage spectrums of benign and malicious software differ (similar to Zwanger and Freiling [146]). In the same context, the code similarity analysis methods presented in Section 6.3 and 6.4 could be used to isolate and exclude those WinAPI functions introduced by library code and frameworks in order to achieve a more pure representation of WinAPI functions used in the actual malicious code. Given the significance of WinAPI usage in order to implement certain capabilities, one could investigate whether WinAPI usage similarity may even serve as an indicator for code similarity between programs. Additionally, future effort should be invested to investigate

how WinAPI information can be leveraged to accelerate malware analysis workflows, e.g. by guiding an analyst's attention towards selected groups of semantically classified WinAPI functions tied to specific malicious capabilities.

Extended investigation of code sharing in malware: In Section 6.4.3, we showed that third-party library usage is common in malware but genuine source code overlaps appear to be rare. Apart from the two highlighted case studies, we believe it would be fruitful to further study the characteristics of such code overlaps. This could lead to a better understanding how these potentially very interesting links between families express themselves concretely in the code and how code similarity could be better approached from a practical point of view. Best to our knowledge, the use of one-to-many code similarity analysis is not widely adapted yet, which also offers chances to examine it from a usability point of view to tame complexity and quantity of data to consider during malware investigations.

Transfer of the proposed approaches to other platforms: For this dissertation, we have exclusively focused on Windows as operating system and Intel x86/x64 as architecture. As indicated in Section 4.3.4, malware is being actively developed for several other architectures and operating systems. One future endeavor could be to generalize the presented approaches to other platforms. For example, examining how characteristic Android API and permission sets are for mobile malware families (similar to Aafer et al. [347]), examining if the function entry point identification heuristic can be adapted to ARM or MIPS, or if the code similarity analysis can be expanded to cross-platform matching [199].

Bibliography

[1] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage, "Spamalytics: An Empirical Analysis of Spam Marketing Conversion," *Communications of the ACM*, vol. 52, 2009.

[2] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna, "The Underground Economy of Spam: A Botmaster's Perspective of Coordinating Large-Scale Spam Campaigns," in *Proceedings of the 4th USENIX Conference on Large-Scale Exploits and Emergent Threats (LEET)*, 2011.

[3] R. Cohen and D. Walkowski, "Banking Trojans: A Reference Guide to the Malware Family Tree," 2019. Blog post: `https://www.f5.com/labs/articles/education/banking-trojans-a-reference-guide-to-the-malware-family-tree` [online; accessed April 2022].

[4] European Cybercrime Centre, "WannaCry Ransomware," 2017. Blog post: `https://www.europol.europa.eu/wannacry-ransomware` [online; accessed April 2022].

[5] Cyber Security Policy, "Securing cyber resilience in health and care: Progress update october 2018," tech. rep., Department of Health and Social Care, Oct. 2018.

[6] K. Geers, D. Kindlund, N. Moran, and R. Rachwald, "WORLD WAR C: Understanding Nation-State Motives Behind Today's Advanced Cyber Attacks," tech. rep., FireEye, 2014.

[7] J. A. Guerrero-Saad, C. Raiu, D. Moore, and T. Rid, "Penquin's Moonlit Maze - The Dawn of Nation-State Digital Espionage," tech. rep., Kaspersky Labs, 2018.

[8] R. Langner, "To Kill a Centrifuge - A Technical Analysis of What Stuxnet's Creators Tried to Achieve," tech. rep., The Langner Group, 2013.

[9] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

[10] J. Calvet, F. L. Lévesque, J. M. Fernandez, E. Traourouder, F. Menet, and J.-Y. Marion, "WaveAtlas: Surfing Through the Landscape of Current Malware Packers," in *Proceedings of the 2015 VirusBulletin Conference (VB)*, 2015.

[11] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, 2011.

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

[13] D. Plohmann, S. Enders, and E. Padilla, "ApiScout: Robust Windows API Usage Recovery for Malware Characterization and Similarity Analysis," *The Journal on Cybercrime & Digital Investigations*, vol. 4, 2018.

[14] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer-Verlag Berlin Heidelberg, 1999.

[15] D. Binkley, "Source Code Analysis: A Road Map," in *Future of Software Engineering (FOSE)*, 2007.

[16] G. Balakrishnan and T. Reps, "WYSINWYX: What You See is Not What You EXecute," *ACM Transactions on Programming Languages and Systems*, vol. 32, 2010.

[17] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools, 2nd Edition*. Pearson, 2007.

[18] R. M. Stallman and GCC Developer Community, *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Scotts Valley, CA: CreateSpace, 2009.

[19] C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," in *Proceedings of the 17th International Workshop on Languages and Compilers for High Performance Computing (LCPC), Mini Workshop on Compiler Research Infrastructures*, 2004.

[20] Microsoft, "Visual Studio," 2021. Homepage: `https://visualstudio.microsoft.com/` [online; accessed April 2022].

[21] various, "PE format," 2021. MSDN Article: `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format` [online; accessed April 2022].

[22] M. Pietrek, "Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format," 2002. MSDN Article: `https://docs.microsoft.com/en-us/previous-versions/bb985992(v=msdn.10)` [online; accessed April 2022].

[23] D. Pistelli, "Microsoft's Rich Signature (undocumented)," 2008. Blog post: `http://ntcore.com/files/richsign.htm` [online; accessed April 2022].

[24] lifewire, "Article: Things They Didn't Tell You About MS LINK and the PE Header," 2004. Blog post: `http://bytepointer.com/articles/rich_header_lifewire_vxmags_29A-8.009.htm` [online; accessed April 2022].

[25] 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)*, 2017.

[26] M. Poslušný and P. Kálnai, "Rich Headers: Leveraging this mysterious artifact of the PE format," in *Proceedings of the 2020 VirusBulletin Conference (VB)*, 2020.

[27] A. Albertini, "Proof of Concepts," 2011. Github Repository: `https://github.com/corkami/pocs` [online; accessed April 2022].

[28] Microsoft, "MSDN Library," 1992. Homepage: `https://msdn.microsoft.com/library` [online; accessed April 2022].

[29] J. Kinder, *Static Analysis of x86 Executables*. PhD thesis, Technische Universität Darmstadt, 2010.

[30] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proceedings of the 25th USENIX Security Symposium (USENIX)*, 2016.

[31] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-Agnostic Function Detection in Binaries," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[32] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2007.

[33] I. U. Haq and J. Caballero, "A Survey of Binary Code Similarity," 2019. arXiv:1909.11424 [cs.CR].

[34] D. Plohmann, E. Gerhards-Padilla, and F. Leder, "Botnets: Detection, measurement, disinfection & defence," *European Network and Information Security Agency (ENISA)*, vol. 1, 2011.

[35] AV-TEST Institut, "Gesamtmenge von Malware und PUA unter Windows," 2021. Microsoft C++ Team Blog: `https://portal.av-atlas.org/malware/statistics` [online; accessed April 2022].

[36] M. Midler, K. O'Meara, and A. Parisi, "Current Ransomware Threats," tech. rep., SEI, 2020.

[37] Iman Ghosh, "This is the crippling cost of cybercrime on corporations." Blog article for World Economics Forum: `https://www.weforum.org/agenda/2019/11/cost-cybercrime-cybersecurity/` [online; accessed April 2022].

[38] D. Plohmann, S. Eschweiler, and E. Gerhards-Padilla, "Patterns of a Cooperative Malware Analysis Workflow," in *Proceedings of the 5th International Conference on Cyber Conflict (CyCon)*, 2013.

[39] E. J. Chikofsky and J. H. Cross II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, 1990.

[40] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 2, 1936.

[41] T. Robinson, *Building Virtual Machine Labs: A Hands-On Guide*. CreateSpace Independent Publishing Platform, 2017.

[42] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2013.

[43] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security and Privacy*, vol. 5, 2007.

[44] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley, 2014.

[45] I. Guilfanov, "IDA Pro," May 1990. Company Website: `https://hex-rays.com/ida-pro/` [online; accessed April 2022].

[46] National Security Agency, "The Ghidra Software Reverse Engineering suite," 2019. Project Website: `https://ghidra-sre.org/` [online; accessed April 2022].

[47] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' processes," in *Proceedings of the 29th USENIX Security Symposium (USENIX)*, 2020.

[48] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[49] A. Taboada-Crispi, H. Sahli, M. Orozco Monteagudo, D. Hernandez Pacheco, and A. Falcon, "Anomaly detection in medical image analysis," *Handbook of Research on Advanced Techniques in Diagnostic Imaging and Biomedical Applications*, 2009.

[50] 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)*, 2012.

[51] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Are Your Training Datasets Yet Relevant?," in *Proceedings of the 7th International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2015.

[52] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara, "Experimental Study with Real-World Data for Android App Security Analysis Using Machine Learning," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, 2015.

[53] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time," in *Proceedings of the 28th USENIX Security Symposium (USENIX)*, 2019.

[54] B. Miller, A. Kantchelian, M. Tschantz, S. Afroz, R. Bachwani, R. Faizullabhoy, L. Huang, V. Shankar, T. Wu, G. Yiu, A. Joseph, and J. Tygar, "Reviewer Integration and Performance Measurement for Malware Detection," in *Proceedings of the 13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2016.

[55] E. van der Kouwe, D. Andriesse, H. Bos, C. Giuffrida, and G. Heiser, "Benchmarking crimes: An emerging threat in systems security," 2018. arXiv:1801.02381 [cs.CR].

[56] S. Abt and H. Baier, "Are We Missing Labels? A Study of the Availability of Ground-Truth in Network Security Research," in *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[57] J. Upchurch and X. Zhou, "Variant: A Malware Similarity Testing Framework," in *Proceedings of the 10th International Conference on Malicious and Unwanted Software (MALWARE)*, 2015.

[58] A. Nappa, M. Z. Rafique, and J. Caballero, "The malicia dataset: Identification and analysis of drive-by download operations," *International Journal of Information Security*, vol. 14, 2015.

[59] Y. Lin, C. Lee, Y. Wu, P. Ho, F. Wang, and Y. Tsai, "Active versus passive malware collection," *Computer*, vol. 47, 2014.

[60] F. Ceschin, F. Pinage, M. Castilho, D. Menotti, L. S. Oliveira, and A. Gregio, "The Need for Speed: An Analysis of Brazilian Malware Classifiers," *IEEE Security and Privacy*, vol. 16, 2018.

[61] B. Grill, A. Bacs, C. Platzer, and H. Bos, ""Nice Boots!" - A Large-Scale Analysis of Bootkits and New Ways to Stop Them," in *Proceedings of the 12th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)* (M. Almgren, V. Gulisano, and F. Maggi, eds.), 2015.

[62] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," 2018. arXiv:1802.10135 [cs.CR].

[63] F. O. Catak and A. F. Yazi, "A benchmark api call dataset for windows pe malware classification," 2019. arXiv:1905.01999 [cs.CR].

[64] T. Barabosch, *Formalization and Detection of Host-Based Code Injection Attacks in the Context of Malware*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2018.

[65] H. S. Anderson and P. Roth, "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models," 2018. arXiv:1804.04637 [cs.CR].

[66] R. Harang and E. M. Rudd, "SOREL-20M: A Large Scale Benchmark Dataset for Malicious PE Detection," 2020. arXiv:2012.07634 [cs.CR].

[67] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, 2012.

[68] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," 2014.

[69] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep Ground Truth Analysis of Current Android Malware," in *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2017.

[70] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, 2016.

[71] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth Attacks: An Extended Insight into the Obfuscation Effects on Android Malware," *Computers and Security*, vol. 51, 2015.

[72] N. Kiss, J.-F. Lalande, M. Leslous, and V. Viet Triem Tong, "Kharon dataset: Android malware under a microscope," in *Learning from Authoritative Security Experiment Results (LASER)*, 2016.

[73] A. H. Lashkari, A. F. A. Kadir, H. Gonzalez, K. F. Mbah, and A. A. Ghorbani, "Towards a network-based framework for android malware detection and characterization," in *Proceedings of the 15th Annual Conference on Privacy, Security and Trust (PST)*, 2017.

[74] F. Alswaina and K. Elleithy, "Android Malware Family Classification and Analysis: Current Status and Future Directions," *Electronics*, vol. 9, 2020.

[75] A. Calleja, J. Tapiador, and J. Caballero, "A Look into 30 Years of Malware Development from a Software Metrics Perspective," in *Proceedings of the 9th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2016.

[76] L. A. Goldberg, P. W. Goldberg, C. A. Phillips, and G. B. Sorkin, "Constructing Computer Virus Phylogenies," *Journal of Algorithms*, vol. 26, 1998.

[77] G. B. Sorkin, "Grouping related computer viruses into families," in *Proceedings of the IBM Security ITS*, 1994.

[78] T. Dumitras and I. Neamtiu, "Experimental Challenges in Cyber Security: A Story of Provenance and Lineage for Malware," in *Proceedings of the 4th Workshop on Cyber Security Experimentation and Test (CSET)*, 2011.

[79] M. Lindorfer, A. Di Federico, F. Maggi, P. M. Comparetti, and S. Zanero, "Lines of Malicious Code: Insights into the Malicious Software Industry," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[80] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proceedings of the 22nd USENIX Security Symposium (USENIX)*, 2013.

[81] I. U. Haq, S. Chica, J. Caballero, and S. Jha, "Malware lineage in the wild," *Computers & Security*, vol. 78, 2018.

[82] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[83] F. Maggi, A. Bellini, G. Salvaneschi, and S. Zanero, "Finding Non-trivial Malware Naming Inconsistencies," in *Proceedings of the 7th International Conference on Information Systems Security (ICISS)*, 2011.

[84] A. Mohaisen and O. Alrawi, "AV-Meter: An Evaluation of Antivirus Scans and Labels," in *Proceedings of the 11th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2014.

[85] R. Perdisci and M. U, "VAMO: Towards a Fully Automated Malware Clustering Validity Analysis," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012.

[86] A. Kantchelian, M. C. Tschantz, S. Afroz, B. Miller, V. Shankar, R. Bachwani, A. D. Joseph, and J. D. Tygar, "Better Malware Ground Truth: Techniques for Weighting Anti-Virus Vendor Labels," in *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2015.

[87] A. R. A. Grégio, V. M. Afonso, D. S. F. Filho, P. L. d. Geus, and M. Jino, "Toward a taxonomy of malware behaviors," *Computer*, vol. 58, 2015.

[88] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVclass: A Tool for Massive Malware Labeling," in *Proceedings of the 9th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2016.

[89] S. Sebastián and J. Caballero, "AVclass2: Massive Malware Tag Extraction from AV Labels," in *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, 2020.

[90] M. Hurier, K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "On the Lack of Consensus in Anti-Virus Decisions: Metrics and Insights on Building Ground Truths of Android Malware," in *Proceedings of the 13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2016.

[91] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. Le Traon, J. Klein, and L. Cavallaro, "Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, 2017.

[92] X. Ugarte-Pedrero, M. Graziano, and D. Balzarotti, "A close look at a daily dataset of malware samples," *ACM Transactions on Privacy and Security*, vol. 22, 2019.

[93] ParzivalWolfram, LeTesla, and alu_pahrata, "The malware wiki," 2009. Website: `https://malwiki.org/` [online; accessed April 2022].

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

[95] K. Bandla and S. Castro, "APTnotes," 2016. Github Repository: `https://github.com/aptnotes/data` [online; accessed April 2022].

[96] F. Roth and Contributors, "APT Groups and Operations," 2015. Google Spreadsheet: `https://apt.threattracking.com` [online; accessed April 2022].

[97] B. Strom, A. Applebaum, D. Miller, K. Nickels, A. Pennington, and C. Thomas, "MITRE ATT&CK: Design and Philosophy," tech. rep., The MITRE Corporatation, 2018.

[98] 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 3rd ACM on Workshop on Information Sharing and Collaborative Security (WISCS)*, 2016.

[99] Council on Foreign Relations, "Cyber Operations Tracker," 2005. Website: `https://www.cfr.org/cyber-operations` [online; accessed April 2022].

[100] ThaiCERT, "Threat Group Cards," 2019. Website: `https://apt.thaicert.or.th/cgi-bin/aptgroups.cgi` [online; accessed April 2022].

[101] G. Laurenza and R. Lazzeretti, "daptaset: A comprehensive mapping of apt-related data," in *Proceedings of the 25th European Symposium on Research in Computer Security (ESORICS)*, 2020.

[102] J. Gray, D. Sgandurra, and L. Cavallaro, "Identifying authorship style in malicious binaries: Techniques, challenges, and datasets," 2021. arXiv:2101.06124 [cs.CR].

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

[104] K. O'Meara, "API Hashing Tool, Imagine That," 2019. Blog Post: `https://insights.sei.cmu.edu/cert/2019/03/api-hashing-tool-imagine-that.html` [online; accessed April 2022].

[105] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A Framework for Enabling Static Malware Analysis," in *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS)*, 2008.

[106] J. Raber and B. Krumheuer, "QuietRIATT: Rebuilding the Import Address Table Using Hooked DLL Calls," in *Proceedings of BlackHat DC*, 2009.

[107] G. Hunt and D. Tarditi, "Detours," 2002. Project Homepage by Microsoft Research: `https://www.microsoft.com/en-us/research/project/detours/` [online; accessed April 2022].

[108] Qi Xi, Tianyang Zhou, Qingxian Wang, and Yongjun Zeng, "An api deobfuscation method combining dynamic and static techniques," in *Proceedings of the 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC)*, 2013.

[109] S. Choi, "API Deobfuscator: Identifying Runtime-obfuscated API calls via Memory Access Analysis," 2015. Presentation given at BlackHat Asia: `https://www.blackhat.com/docs/asia-15/materials/asia-15-Choi-API-Deobfuscator-Indentifying-Runtime-Obfuscated-API-Calls-Via-Memory-Access-Analysis.pdf` [online; accessed April 2022].

[110] D. Korczynski, "RePEconstruct: reconstructing binaries with self-modifying code and import address table destruction," in *Proceedings of the 11th International Conference on Malicious and Unwanted Software (MALWARE)*, 2016.

[111] Y. Kawakoya, M. Iwamura, and J. Miyoshi, "Taint-assisted IAT Reconstruction against Position Obfuscation," *Journal of Information Processing*, vol. 26, 2018.

[112] Y. Kawakoya, E. Shioji, Y. Otsuki, M. Iwamura, and J. Miyoshi, "Stealth Loader: Trace-free Program Loading for Analysis Evasion," *Journal of Information Processing*, vol. 26, 2018.

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

[114] NtQuery, "Scylla," 2011. Github Repository: `https://github.com/NtQuery/Scylla` [online; accessed April 2022].

[115] M. Ligh, "Volatility Command ImpScan," 2012. Github Repository: `https://github.com/volatilityfoundation/volatility/wiki/Command-Reference` [online; accessed April 2022].

[116] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant, "Semantics-aware malware detection," in *Proceedings of the 26th IEEE Symposium on Security and Privacy (S&P)*, 2005.

[117] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE)*, 2007.

[118] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[119] C. Chen, C. X. Lin, M. Fredrikson, M. Christodorescu, X. Yan, and J. Han, "Mining graph patterns efficiently via randomized summaries," *Proceedings of the VLDB Endowment*, vol. 2, 2009.

[120] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors," in *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[121] Y. Hu, L. Chen, M. Xu, N. Zheng, and Y. Guo, "Unknown malicious executables detection based on run-time behavior," in *Proceedings of the 5th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2008.

[122] Lei Liu and Kun Shao, "An approach of malicious executables detection on black gray based on adaboost algorithm," in *Proceedings of the 2nd International Conference on Anti-counterfeiting, Security and Identification (ASID)*, 2008.

[123] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "A View on Current Malware Behaviors," in *Proceedings of the 2nd USENIX Conference on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[124] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Proceedings of the 5th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2008.

[125] P. Trinius, C. Willems, T. Holz, and K. Rieck, "A Malware Instruction Set for Behavior-Based Analysis," in *Proceedings of the 2010 Sicherheit Conference*, 2010.

[126] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host," in *Proceedings of the 18th USENIX Security Symposium (USENIX)*, 2009.

[127] M. Apel, C. Bockermann, and M. Meier, "Measuring similarity of malware behavior," in *Proceedings of the 34th Annual IEEE Conference on Local Computer Networks (LCN)*, 2009.

[128] Julia Yu-Chin Cheng, Tzung-Shian Tsai, and Chu-Sing Yang, "An information retrieval approach for malware classification based on windows api calls," in *Proceedings of the 12th International Conference on Machine Learning and Cybernetics (ICMLC)*, 2013.

[129] C. Guarnieri, A. Tanasi, J. Bremer, M. Schloesser, K. Houtman, R. van Zutphen, and B. de Graaff, "Cuckoo sandbox," 2010. Website of Cuckoo Sandbox: `https://cuckoosandbox.org/` [online; accessed April 2022].

[130] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on API call sequence analysis," *International Journal of Distributed Sensor Networks*, vol. 11, 2015.

[131] S. Gupta, H. Sharma, and S. Kaur, "Malware Characterization Using Windows API Call Sequences," in *Proceedings of the 6th International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE)*, 2016.

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

[133] B. Anderson, C. Storlie, and T. Lane, "Improving malware classification: bridging the static/-dynamic gap," in *Proceedings of the 5th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2012.

[134] P. Shijo and B. Salim, "Integrated Static and Dynamic Analysis for Malware Detection," in *Proceedings of the 2nd International Conference on Information and Communication Technologies (ICICT)*, 2014.

[135] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *Proceedings of the 22nd IEEE Symposium on Security and Privacy (S&P)*, 2001.

[136] Y. Lu, S. Din, C. Zheng, and B. Gao, "Using multi-feature and classifier ensembles to improve malwaredetection," *Journal of Chung Cheng Institute of Technology*, vol. 39, 2010.

[137] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze, "Malware Detection Based on Mining API Calls," in *Proceedings of the 25th ACM Symposium on Applied Computing (SAC)*, 2010.

[138] V. S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar, "Signature Generation and Detection of Malware Families," in *Proceedings of the 13th Australasian Conference on Information Security and Privacy (ACISP)*, 2008.

[139] E. Baranov, F. Biondi, O. Decourbe, T. Given-Wilson, A. Legay, C. Puodzius, J. Quilbeuf, and S. Sebastio, "Efficient Extraction of Malware Signatures Through System Calls and Symbolic Execution: An Experience Report." preprint, 2018.

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

[141] M. Alazab, S. Venkatraman, P. Watters, and M. Alazab, "Zero-Day Malware Detection Based on Supervised Learning Algorithms of API Call Signatures," in *Proceedings of the 9th Australasian Data Mining Conference (AusDM)*, 2011.

[142] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, "PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime," in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2009.

[143] P. Beaucamps, I. Gnaedig, and J.-Y. Marion, "Behavior abstraction in malware analysis," in *Proceedings of the 10th International Conference on Runtime Verification (RV)*, 2010.

[144] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, 2005.

[145] P. Beaucamps, I. Gnaedig, and J.-Y. Marion, "Abstraction-based malware analysis using rewriting and model checking," in *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS)*, 2012.

[146] 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)*, 2013.

[147] B. Caillat, B. Gilbert, R. A. Kemmerer, C. Kruegel, and G. Vigna, "Prison: Tracking Process Interactions to Contain Malware," in *Proceedings of the 7th International Symposium on Cyberspace Safety and Security (CSS)*, 2015.

[148] D. Kirat and G. Vigna, "MalGene: Automatic Extraction of Malware Analysis Evasion Signature," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

[149] A. Mohaisen, O. Alrawi, and M. Mohaisen, "Amal: High-fidelity, behavior-based automated malware analysis and classification," *Computers and Security*, 2015.

[150] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. Matsumoto, "Design and Evaluation of Dynamic Software Birthmarks Based on API Calls," tech. rep., Nara Institute of Science and Technology, 2007.

[151] S. Choi, H. Park, H.-I. Lim, and T. Han, "A static birthmark of binary executables based on api call structure," in *Proceedings of the 12th Annual Asian Computing Science Conference (ASIAN)*, 2007.

[152] Q. Guan, Y. Tang, and X. Liu, "A Malware Homologous Analysis Method Based on Sequence of System Function," in *Proceedings of the 4th International Conference on Advanced Science and Technology (AST)*, 2012.

[153] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying Dormant Functionality in Malware Programs," in *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[154] L. Guevara and D. Plohmann, "Semantic Exploration of Binaries," 2014. Presentation given at Botconf: `https://www.botconf.eu/wp-content/uploads/2014/12/2014-1.3-Semantic-Exploration-of-Binaries.pdf` [online; accessed April 2022].

[155] K. Oosthoek and C. Doerr, "SoK: ATT&CK Techniques and Trends in Windows Malware," in *Proceedings of the 15th International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2019.

[156] S. Bühlmann, "JoeBox Malware Analysis," 2011. Website of JoeSecurity: `https://www.joesecurity.org/` [online; accessed April 2022].

[157] W. Ballenthin, M. Raabe, and the FLARE Team, "capa: Automatically Identify Malware Capabilities," 2020. Blog post for FireEye: `https://www.fireeye.com/blog/threat-research/2020/07/capa-automatically-identify-malware-capabilities.html` [online; accessed April 2022].

[158] O. Alrawi, M. Ike, M. Pruett, R. P. Kasturi, S. Barua, T. Hirani, B. Hill, and B. Saltaformaggio, "Forecasting Malware Capabilities From Cyber Attack Memory Images," in *Proceedings of the 30th USENIX Security Symposium (USENIX)*, 2021.

[159] various, "Supply Chain Analysis: From Quartermaster to Sunshop," tech. rep., FireEye, 2013.

[160] S. Tomonaga, "Classifying Malware using Import API and Fuzzy Hashing – impfuzzy," 2016. Blog post for JPCERT/CC: `https://blogs.jpcert.or.jp/en/2016/05/classifying-mal-a988.html` [online; accessed April 2022].

[161] Free Software Foundation, "Manpage of OBJDUMP," 1991. Website: `https://linux.die.net/man/1/objdump` [online; accessed April 2022].

[162] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary Translation," *Communications of the ACM*, vol. 36, 1993.

[163] C. Cifuentes and M. V. Emmerik, "UQBT: Adaptable Binary Translation at Low Cost," *Computer*, vol. 33, 2000.

[164] C. Cifuentes and M. Van Emmerik, "Recovery of jump table case statements from binary code," *Science of Computer Programming*, vol. 40, 2001.

[165] H. Theiling, "Extracting safe and precise control flow from binaries," in *Proceedings 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 2000.

[166] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of Executable Code Revisited," in *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE)*, 2002.

[167] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, 2003.

[168] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen, "On the Static Analysis of Indirect Control Transfers in Binaries," in *Proceedings of the 6th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2000.

[169] X. Meng and B. P. Miller, "Binary Code is Not Easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[170] A. Di Federico, M. Payer, and G. Agosta, "Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries," in *Proceedings of the 26th International Conference on Compiler Construction (CC)*, 2017.

[171] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly but Were Afraid to Ask," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, 2021.

[172] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt, "Learning to Analyze Binary Computer Code," in *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI)*, 2008.

[173] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *The International Journal of High Performance Computing Applications*, vol. 14, 2000.

[174] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating Code from Data in X86 Binaries," in *Proceedings of the 2011 European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, 2011.

[175] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23rd USENIX Security Symposium (USENIX)*, 2014.

[176] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing Functions in Binaries with Neural Networks," in *Proceedings of the 24th USENIX Security Symposium (USENIX)*, 2015.

[177] K. Pei, J. Guan, D. W. King, J. Yang, and S. Jana, "XDA: Accurate, Robust Disassembly with Transfer Learning," in *Proceedings of the 28th Annual Network & Distributed System Security Conference (NDSS)*, 2021.

[178] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," in *Proceedings of the 13th USENIX Security Symposium (USENIX)*, 2004.

[179] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *ACM SIGARCH Computer Architecture News*, vol. 33, 2005.

[180] S. Wang, P. Wang, and D. Wu, "Reassembleable Disassembling," in *Proceedings of the 24th USENIX Security Symposium (USENIX)*, 2015.

[181] J. Caballero, N. Johnson, S. McCamant, and D. Song, "Binary Code Extraction and Interface Identification for Security Applications," in *Proceedings of the 17th Annual Network & Distributed System Security Conference (NDSS)*, 2010.

[182] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proceedings of the 26th USENIX Security Symposium (USENIX)*, 2017.

[183] B. S. Baker, U. Manber, and R. Muth, "Compressing differences of executable code," in *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software (WCSSS)*, 1999.

[184] Z. Wang, K. Pierce, and S. McFarling, "BMAT – A Binary Matching Tool for Stale Profile Propagation," *The Journal of Instruction-level Parallelism*, vol. 2, 2000.

[185] E. Carrera and G. Erdélyi, "Digital genome mapping-advanced binary malware analysis," in *Proceedings of the 2004 VirusBulletin Conference (VB)*, 2004.

[186] A. Schulman, "Finding binary clones with opstrings & function digests: Part III," *Dr. Dobb's Journal*, vol. 30, 2005.

[187] C. Cohen and J. Havrilla, "Function Hashing for Malicious Code Analysis," tech. rep., SEI, CMU, 2009.

[188] M. R. Farhadi, B. C. M. Fung, P. Charland, and M. Debbabi, "BinClone: Detecting Code Clones in Malware," in *Proceedings of the 8th IEEE International Conference on Software Security and Reliability (SERE)*, 2014.

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

[190] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology*, vol. 1, 2005.

[191] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia, "Exploiting Similarity Between Variants to Defeat Malware "Vilo" Method for Comparing and Searching Binary Programs," in *Proceedings of BlackHat DC*, 2007.

[192] A. Saedbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting Code Clones in Binary Executables," in *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[193] J. Upchurch and X. Zhou, "Malware provenance: code reuse detection in malicious software at scale," in *Proceedings of the 11th International Conference on Malicious and Unwanted Software (MALWARE)*, 2016.

[194] G. Tahan, L. Rokach, and Y. Shahar, "Mal-ID: Automatic Malware Detection Using Common Segment Analysis and Meta-Features," *The Journal of Machine Learning Research*, 2012.

[195] M. Hassen and P. K. Chan, "Scalable Function Call Graph-Based Malware Classification," in *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[196] E. Raff and C. Nicholas, "Hash-Grams: Faster N-Gram Features for Classification and Malware Detection," in *Proceedings of the 18th ACM Symposium on Document Engineering (DocEng)*, 2018.

[197] D. Bruschi, L. Martignoni, and M. Monga, "Code normalization for self-mutating malware," *IEEE Security and Privacy*, vol. 5, 2007.

[198] J. Miller, "Conceptual design, implementation and validation of signatures based on data of aree analysis for the comparison of functional similarity of executable files available in binary form," diploma thesis, University of Rostock, Germany, 2008.

[199] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in *Proceedings of the 23rd Annual Network & Distributed System Security Conference (NDSS)*, 2016.

[200] H. Flake, "Structural Comparison of Executable Objects," in *Proceedings of the 1st Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2004.

[201] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," in *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2005.

[202] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic Worm Detection Using Structural Information of Executables," in *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[203] S. Cesare, Y. Xiang, and W. Zhou, "Malwise – An Effective and Efficient Classification System for Packed and Polymorphic Malware," *IEEE Transactions on Computers*, vol. 62, 2013.

[204] S. H. Ding, B. C. Fung, and P. Charland, "Kam1n0: MapReduce-Based Assembly Clone Search for Reverse Engineering," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

[205] H. Huang, A. M. Youssef, and M. Debbabi, "BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection," in *Proceedings of the 11th Asia Conference on Computer and Communications Security (AsiaCCS)*, 2017.

[206] F. Leder, B. Steinbock, and P. Martini, "Classification and detection of metamorphic malware using value set analysis," in *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE)*, 2009.

[207] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary Function Clustering Using Semantic Hashes," in *Proceedings of the 11th International Conference on Machine Learning and Applications (ICMLA)*, 2012.

[208] A. Broder, "On the Resemblance and Containment of Documents," in *Proceedings of the Compression and Complexity of Sequences (SEQUENCES)*, 1997.

[209] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast Location of Similar Code Fragments Using Semantic 'Juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, 2013.

[210] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components," in *Proceedings of the 23rd USENIX Security Symposium (USENIX)*, 2014.

[211] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-Architecture Bug Search in Binary Executables," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[212] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proceedings of the 10th International Conference on Mining Software Repositories (MSR)*, 2013.

[213] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "FOSSIL: A Resilient and Efficient System for Identifying FOSS Functions in Malware Binaries," *ACM Transactions on Privacy and Security*, vol. 21, 2018.

[214] T. Dullien, "Motivation and Overview for FunctionSimSearch," 2018. Blog post: `https://github.com/googleprojectzero/functionsimsearch/blob/master/doc/01-motivation-and-overview.md` [online; accessed April 2022].

[215] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, 2002.

[216] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," 2013. arXiv:1301.3781 [cs.CL].

[217] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable Graph-Based Bug Search for Firmware Images," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, 2016.

[218] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2017.

[219] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: Self-Attentive Function Embeddings for Binary Similarity," in *Proceedings of the 16th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2019.

[220] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[221] A. R. Bryant, *Understanding How Reverse Engineers Make Sense of Programs from Assembly Language Representations*. PhD thesis, Air Force Institute of Technology, 2012.

[222] D. Pucsek, J. Baldwin, L. MacLeod, C. Berg, Y. Coady, and M. Salois, "ICE: Binary Analysis That You Can See," in *Proceedings of the 13th IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, 2013.

[223] J. Baldwin, *Program Comprehension Support for Assembly Language: Assessing the Needs of Specialized Groups*. PhD thesis, University of Victoria, 2014.

[224] J. Baldwin, C. A. Teh, E. Baniassad, D. Van Rooy, and Y. Coady, "Requirements for tools for comprehending highly specialized assembly language code and how to elicit these requirements," *Requirements Engineering*, vol. 21, 2014.

[225] C. Q. Nguyen and J. E. Goldman, "Malware Analysis Reverse Engineering (MARE) Methodology and Malware Defense (M.D.) Timeline," in *Proceedings of the 6th Annual Conference on Information Security Curriculum Development (InfoSecCD)*, 2010.

[226] J. Bermejo Higuera, C. Abad Aramburu, J.-R. Bermejo Higuera, M. A. Sicilia Urban, and J. A. Sicilia Montalvo, "Systematic Approach to Malware Analysis (SAMA)," *Applied Sciences*, vol. 10, 2020.

[227] E. Kim, S.-J. Park, D.-K. Chae, S. Choi, and S.-W. Kim, "A Human-in-the-Loop Approach to Malware Author Classification," in *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM)*, 2020.

[228] L. Obrst, P. Chase, and R. Markeloff, "Developing an ontology of the cyber security domain," in *Proceedings of the 7th Conference on Semantic Technology for Intelligence, Defense, and Security (STIDS)*, 2012.

[229] M. Praszmo, "Dissecting smokeloader," 2018. Blog post for CERT.PL: `https://cert.pl/en/posts/2018/07/dissecting-smoke-loader/` [online; accessed April 2022].

[230] A. Marzano, D. Alexander, O. Fonseca, E. Fazzion, C. Hoepers, K. Steding-Jessen, M. H. P. C. Chaves, I. Cunha, D. Guedes, and W. Meira, "The Evolution of Bashlite and Mirai IoT Botnets," in *Proceedings of the 23rd IEEE Symposium on Computers and Communications (ISCC)*, 2018.

[231] O. Bergman, N. Gradovitch, J. Bar-Ilan, and R. Beyth-Marom, "Folder versus tag preference in personal information management," *Journal of the American Society for Information Science and Technology*, vol. 64, 2013.

[232] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format." RFC 7159, 2014.

[233] W3C, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," 2008. Website: `https://www.w3.org/TR/xml/`.

[234] StackOverflow, "Developer survey results 2018."

[235] A. F. Skulason and V. Bontchev, "A new virus naming convention," in *CARO meeting*, 1991.

[236] V. Bontchev, "Current status of the caro malware naming scheme," *Proceedings of the 2005 Virus-Bulletin Conference (VB)*, 2005.

[237] P. J. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace." RFC 4122, 2005.

[238] VirusTotal, "VirusTotal Malware Intelligence Services." Website: `https://www.virustotal.com/intelligence/` [online; accessed April 2022].

[239] M. Stevens, "Fast Collision Attack on MD5," *IACR Cryptology ePrint Archive*, vol. 2006, 2006.

[240] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1," in *Proceedings of the 37th Annual International Cryptology Conference (CRYPTO)*, 2017.

[241] M. Brengel and C. Rossow, "MemScrimper: Time- and Space-Efficient Storage of Malware Sandbox Memory Dumps," in *Proceedings of the 15th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2018.

[242] J. Segura, "StatCounter Global Stats: Desktop Windows Version Market Share Worldwide," 2021. Website: `http://gs.statcounter.com/` [online; accessed April 2022].

[243] StackOverflow, "Developer Survey Results 2016," 2016. Blog post: `https://insights.stackoverflow.com/survey/2016` [online; accessed April 2022].

[244] D. Plohmann, "Knowledge Fragment: Hardening Win7 x64 on VirtualBox for Malware Analysis," 2017. Blog post for ByteAtlas: `http://byte-atlas.blogspot.de/2017/02/hardening-vbox-win7x64.html` [online; accessed April 2022].

[245] T. Jenke, D. Plohmann, and E. Padilla, "RoAMer: The Robust Automated Malware Unpacker," in *Proceedings of 14th International Conference on Malicious and Unwanted Software (MALWARE)*, 2019.

[246] W. J. Liu, "Process Hacker," 2019. SourceForge Entry: `https://processhacker.sourceforge.io/` [online; accessed April 2022].

[247] Shadowserver, "The Shadowserver Foundation," 2019. Wiki: `https://www.shadowserver.org/wiki/` [online; accessed April 2022].

[248] S. Moore and E. Keen, "Gartner Forecasts Worldwide Information Security Spending to Exceed $124 Billion in 2019," 2018. Press Release: `https://www.gartner.com/en/newsroom/press-releases/2018-08-15-gartner-forecasts-worldwide-information-security-spending-to-exceed-124-billion-in-2019` [online; accessed April 2022].

[249] A. Albertini, "PE," 2013. Github Repository: `https://github.com/corkami/docs/blob/master/PE/PE.md` [online; accessed April 2022].

[250] G. Szappanos, "PlugX - The Next Generation," tech. rep., Sophos Labs, 2014.

[251] E. Carrera, "pefile," 2007. Github Repository: `https://github.com/erocarrera/pefile` [online; accessed April 2022].

[252] Visgean, "Mirror of the zeus 2.0.8.9. source code," 2013. Github Repository: `https://github.com/Visgean/Zeus/blob/c55a9fa8c8564ec196604a59111708fa8415f020/make/tools.inc.php#L695` [online; accessed April 2022].

[253] roy g biv / defjam, "Heaven's Gate: 64-bit code in 32-bit file," *Valhalla eZines*, vol. 1, 2011.

[254] ReWolf, "WOW64Ext," 2019. Github Repository: `https://github.com/rwfpl/rewolf-wow64ext` [online; accessed April 2022].

[255] S. Eschweiler, "YANT: Yet Another Nymaim Talk," 2017. Presentation given at Botconf 2017: `https://www.botconf.eu/wp-content/uploads/2017/12/2017-Eschweiler-YANT-Yet-Another-Nymaim-Talk.pdf` [online; accessed April 2022].

[256] MITRE, "DLL Side-Loading," 2019. MITRE ATT& CK Wiki: `https://attack.mitre.org/techniques/T1073/` [online; accessed April 2022].

[257] A. Blaszczyk, "The not so boring land of Borland executables, part 1," 2014. Hexacorn Blog: `http://www.hexacorn.com/blog/2014/12/05/the-not-so-boring-land-of-borland-executables-part-1/` [online; accessed April 2022].

[258] Horsicq, "Detect-It-Easy," 2014. GitHub Repository: `https://github.com/horsicq/Detect-It-Easy/` [online; accessed April 2022].

[259] M. Rullgard and C. Zoulas, "Magic Number Recognition Library," 2018. Linux Manpage: `http://man7.org/linux/man-pages/man3/libmagic.3.html` [online; accessed April 2022].

[260] R. Chen, "Windows Confidential: Getting Out of DLL Hell," 2007. MSDN Article: `https://docs.microsoft.com/en-us/previous-versions/technet-magazine/cc162526(v=msdn.10)` [online; accessed April 2022].

[261] K. Kahane, "Dynamically linking with MSVCRT.DLL using Visual C++ 2005," 2007. Blog post: `https://kobyk.wordpress.com/2007/07/20/dynamically-linking-with-msvcrtdll-using-visual-c-2005/` [online; accessed April 2022].

[262] Visual CPP Team, "/DYNAMICBASE and /NXCOMPAT," 2009. Blog post: `https://blogs.msdn.microsoft.com/vcblog/2009/05/21/dynamicbase-and-nxcompat/` [online; accessed April 2022].

[263] various, "Dynamic-Link Library Entry-Point Function," 2021. MSDN Article: `https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-entry-point-function` [online; accessed April 2022].

[264] A. Shulmin, S. Yunakovsky, V. Berdnikov, and A. Dolgushev, "The Slingshot APT FAQ," 2018. Blog post: `https://securelist.com/apt-slingshot/84312/` [online; accessed April 2022].

[265] S. Miller, "Definitive Dossier of Devilish Debug Details – Part One: PDB Paths and Malware," September 2019. Blog post for FireEye: `https://www.mandiant.com/resources/definitive-dossier-of-devilish-debug-details-part-one-pdb-paths-malware` [online; accessed April 2022].

[266] V. Zwanger, E. Gerhards-Padilla, and M. Meier, "Codescanner: Detecting (Hidden) x86/x64 code in arbitrary files," in *Proceedings of the 9th International Conference on Malicious and Unwanted Software (MALWARE)*, 2014.

[267] GReAT, "OlympicDestroyer is here to trick the industry," 2018. Blog post for Kaspersky Labs: `https://securelist.com/olympicdestroyer-is-here-to-trick-the-industry/84295/` [online; accessed April 2022].

[268] J. Lambert, "Incident Response Triangle: Timeliness, Depth, Accuracy," 2019. Tweet: `https://twitter.com/JohnLaTwC/status/1088126545825157120` [online; accessed April 2022].

[269] D. Plohmann, "ApiScout," 2017. Github Repository: `https://github.com/danielplohmann/apiscout` [online; accessed April 2022].

[270] G. Dabah, "Powerful disassembler library for x86/amd64," 2010. Github Repository: `https://github.com/gdabah/distorm` [online; accessed April 2022].

[271] H. Saidi, P. Porras, and V. Yegneswaran, "Experiences in malware binary deobfuscation," in *Proceedings of the 2010 VirusBulletin Conference (VB)*, 2010.

[272] Trend Micro Forward-Looking Threat Research Team, "Dissecting PRILEX and CUTLET MAKER ATM Malware Families," 2017. Blog post: `https://www.trendmicro.com/ru_ru/research/17/l/dissecting-prilex-cutlet-maker-atm-malware-families.html` [online; accessed April 2022].

[273] various, "Conventions for Function Prototypes," 2018. MSDN Article: `https://msdn.microsoft.com/en-us/library/windows/desktop/dd317766(v=vs.85).aspx` [online; accessed April 2022].

[274] A. Fog, "Calling Conventions," tech. rep., Technical University of Denmark, 2004.

[275] R. Rivest, "The MD5 Message-Digest Algorithm." RFC 1321, 1992.

[276] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," in *Proceedings of the 2004 International Workshop on Fast Software Encryption (FSE)*, 2004.

[277] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings." RFC 4648, 2006.

[278] T. Tsukiyama, Y. Kondo, K. Kakuse, S. Saba, S. Ozaki, and K. Itoh, "Method and system for data compression and restoration," 1983. Patent: US4586027A.

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

[280] P. Jaccard, "The Distribution of the Flora in the Alpine Zone," *New Phytologist*, vol. 11, 1912.

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

[282] C. Raiu, "Looking at Big Threats Using Code Similarity. Part 1," 2020. Blog post for Kaspersky Labs: `https://securelist.com/big-threats-using-code-similarity-part-1/97239/` [online; accessed April 2022].

[283] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[284] M. Gorelik and R. Moshailov, "Fileless Malware: Attack Trend Exposed," 2017. Analysis Report for Morphisec: `https://www.morphisec.com/hubfs/wp-content/uploads/2017/11/Fileless-Malware_Attack-Trend-Exposed.pdf` [online; accessed April 2022].

[285] A. Doniec and M. Lechtik, "Funky Malware Formats," 2019. Presentation at Kaspersky SAS: `https://speakerdeck.com/hshrzd/funky-malware-formats` [online; accessed April 2022].

[286] Kaspersky GReAT, "Shadowpad: popular server management software hit in supply chain attack," 2017. Analysis Report for Kaspersky Labs: `https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2017/08/07172148/ShadowPad_technical_description_PDF.pdf` [online; accessed April 2022].

[287] I. Skochinsky, "Outline of Recursive Disassembly in IDA Pro," 2012. Reddit Comment: `https://www.reddit.com/r/ReverseEngineering/comments/rtzb0/disassembling_in_ida/c48tiuy/` [online; accessed April 2022].

[288] I. Skochinsky, "Simplified Overview of how IDA Pro performs Recursive Disassembly," 2013. Stack Exchange Answer: `http://reverseengineering.stackexchange.com/a/2349/1403` [online; accessed April 2022].

[289] pancake, "Analysis by Default," 2015. Blog Post on radare2 code analysis modes: `http://radare.today/posts/analysis-by-default/` [online; accessed April 2022].

[290] D. Plohmann, "SMDA," 2018. Github Repository: `https://github.com/danielplohmann/smda` [online; accessed April 2022].

[291] N. A. Quynh, "Capstone engine," 2013. Github Repository: `https://github.com/aquynh/capstone` [online; accessed April 2022].

[292] R. Thomas, "Lief - library to instrument executable formats," 2013. Github IO Project Page: `https://lief-project.github.io/` [online; accessed April 2022].

[293] R. Chen, "A few stray notes on Windows patching and hot patching," 2013. MSDN Article: `https://devblogs.microsoft.com/oldnewthing/20130102-00/?p=5663` [online; accessed April 2022].

[294] K. Frei, "X64 Unwind Information," 2006. MSDN Article: `https://docs.microsoft.com/de-de/archive/blogs/freik/x64-unwind-information` [online; accessed April 2022].

[295] various, "Considerations for Writing Prolog/Epilog Code," 2016. MSDN Article: `https://docs.microsoft.com/en-us/cpp/cpp/considerations-for-writing-prolog-epilog-code?view=msvc-160` [online; accessed April 2022].

[296] various, "/hotpatch (Create Hotpatchable Image)," 2018. MSDN Article: `https://docs.microsoft.com/en-us/cpp/build/reference/hotpatch-create-hotpatchable-image?view=msvc-160` [online; accessed April 2022].

[297] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2011.

[298] A. Lakhotia, E. U. Kumar, and M. Venable, "A Method for Detecting Obfuscated Calls in Malicious Binaries," *IEEE Transactions on Software Engineering*, vol. 31, 2005.

[299] AMD Technology, "AMD64 Technology AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions Publication No. Revision Date," 2012.

[300] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," *Copenhagen University College of Engineering*, 2012.

[301] R. Chen, "A few stray notes on Windows patching and hot patching," 2013. MSDN Article: `https://docs.microsoft.com/en-us/cpp/cpp/calling-conventions?view=msvc-160`.

[302] various, "/INCREMENTAL (Link Incrementally)," 2018. MSDN Article: `https://docs.microsoft.com/en-us/cpp/build/reference/incremental-link-incrementally?view=msvc-160` [online; accessed April 2022].

[303] J. Wang, H. T. Shen, J. Song, and J. Ji, "Hashing for similarity search: A survey," 2014. arXiv:1408.2927 [cs.DS].

[304] M. Narayanan and R. Karp, "Gapped Local Similarity Search with Provable Guarantees," in *Algorithms in Bioinformatics*, 2004.

[305] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy, "Mash: fast genome and metagenome distance estimation using minhash," *Genome Biology*, vol. 17, 2016.

[306] O. Chum, J. Philbin, and A. Zisserman, "Near duplicate image detection: min-hash and tf-idf weighting," in *Proceedings of the British Machine Vision Conference (BMVC)*, 2008.

[307] M. Henzinger, "Finding near-duplicate web pages: A large-scale evaluation of algorithms," in *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2006.

[308] A. Shrivastava and P. Li, "In Defense of MinHash Over SimHash," in *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2014.

[309] D. Plohmann, "MCRIT," 2021. Github Repository: `https://github.com/danielplohmann/mcrit`.

[310] O. Ertl, "SuperMinHash - A New Minwise Hashing Algorithm for Jaccard Similarity Estimation," 2017. arXiv:1706.05698 [cs.DS].

[311] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. Cambridge University Press, 2011.

[312] P. Li and C. König, "B-Bit Minwise Hashing," in *Proceedings of the 19th International Conference on World Wide Web (WWW)*, 2010.

[313] F. Adkins, L. Jones, M. Carlisle, and J. Upchurch, "Heuristic malware detection via basic block comparison," in *Proceedings of the 8th International Conference on Malicious and Unwanted Software (MALWARE)*, 2013.

[314] P. A. Laplante, *What Every Engineer Should Know about Software Engineering*. CRC Press, 2007.

[315] M. Oliver, "Shift Media Project," 2002. Website of the Shift Media Project: `https://shiftmediaproject.github.io/` [online; accessed April 2022].

[316] C. Spearman, "The proof and measurement of association between two things," *The American Journal of Psychology*, vol. 15, 1904.

[317] K. Pearson, "Note on Regression and Inheritance in the Case of Two Parents," *Proceedings of the Royal Society of London Series I*, vol. 58, 1895.

[318] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[319] D. Plohmann, "Empty MSVC," 2019. Github Repository: `https://github.com/danielplohmann/empty_msvc` [online; accessed April 2022].

[320] Microsoft, "Vcpkg: Overview," 2019. Github Repository: `https://github.com/microsoft/vcpkg` [online; accessed April 2022].

[321] GReAT, "Red October. Detailed Malware Description 5. Second Stage of Attack," 2013. Blog post for Kaspersky Labs: `https://securelist.com/red-october-detailed-malware-description-5-second-stage-of-attack/36879/` [online; accessed April 2022].

[322] GReAT, "Lazarus Under The Hood," 2017. Analysis Report by Kaspersky Labs: `https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/07180231/LazarusUnderTheHood_PDF_final_for_securelist.pdf` [online; accessed April 2022].

[323] various, "Optimizing C++ Code : Dead Code Elimination," 2013. Microsoft C++ Team Blog: `https://devblogs.microsoft.com/cppblog/optimizing-c-code-dead-code-elimination/` [online; accessed April 2022].

[324] various, "/Gy (Enable Function-Level Linking)," 2016. MSDN Article: `https://docs.microsoft.com/en-us/cpp/build/reference/gy-enable-function-level-linking?redirectedfrom=MSDN&view=msvc-160` [online; accessed April 2022].

[325] S. Zennou, S. K. Debray, T. Dullien, and A. Lakhotia, "Malware Analysis: From Large-Scale Data Triage to Targeted Attack Recognition (Dagstuhl Seminar 17281)," *Dagstuhl Reports*, vol. 7, 2017.

[326] L. Kessem, "Panda Banker," 2016. Blog post for IBM X-Force: `https://isc.sans.edu/forums/diary/Kraken+Technical+Details+UPDATED+x3/4256/` [online; accessed April 2022].

[327] B. Duncan, "Sage 2.0 Ransomware," 2017. Blog post for SANS ISC: `https://isc.sans.edu/forums/diary/Sage+20+Ransomware/21959/` [online; accessed April 2022].

[328] M. Kotowicz, "ISFB: Still Live and Kicking," *The Journal on Cybercrime & Digital Investigations*, vol. 2, 2016.

[329] J. Rosenberg, "IcedID Banking Trojan Shares Code with Pony 2.0 Trojan," 2017. Blog post: `https://www.intezer.com/blog/research/icedid-banking-trojan-shares-code-pony-2-0-trojan/` [online; accessed April 2022].

[330] C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. J. Dietrich, and H. Bos, "SoK: P2PWNED — Modeling and Evaluating the Resilience of Peer-to-Peer Botnets," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[331] A. McNeil, "How did the WannaCry ransomworm spread?," 2017. Blog post for MalwareBytes: `https://blog.malwarebytes.com/cybercrime/2017/05/how-did-wannacry-ransomworm-spread/` [online; accessed April 2022].

[332] N. Mehta, "Attribution hints for WannaCrypt," 2017. Tweet: `https://twitter.com/neelmehta/status/864164081116225536` [online; accessed April 2022].

[333] A. L. Johnson, "SWIFT attackers' malware linked to more financial attacks ," 2016. Blog post for Symantec: `https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=8ae1ff71-e440-4b79-9943-199d0adb43fc&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments` [online; accessed April 2022].

[334] Threat Research and Interdiction Group, "Operation Blockbuster: Unraveling the Long Thread of the Sony Attack," 2016. Analysis Report: `https://operationblockbuster.com/` [online; accessed April 2022].

[335] GReAT, "WannaCry and Lazarus Group – the missing link?," 2017. Blog post for Kaspersky: `https://securelist.com/wannacry-and-lazarus-group-the-missing-link/78431/` [online; accessed April 2022].

[336] Symantec Security Response, "What you need to know about the WannaCry Ransomware," 2017. Blog post for Symantec: `https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/wannacry-ransomware-attack` [online; accessed April 2022].

[337] Aquabox, "User Manual for Citadel Version 1.3.4.5," 2012. Archived in Github Repository: `https://github.com/malwares/Botnet/blob/master/Citadel%201.3.4.5/Citadel%201.3.4.5%20Botnet/Manual/Manual%20Citadel%20v%201.3.4.5.txt` [online; accessed April 2022].

[338] A. Luca and I. Răileanu, "Conference review: Botconf 2017," 2017. Blog post for VirusBulletin: `https://www.virusbulletin.com/blog/2017/12/conference-review-botconf-2017/` [online; accessed April 2022].

[339] T. Roccia, "The Hitchhiker guide to Incident Response and Threat Intelligence," September 2019. Presentation as part of the ENISA Summer School: `https://nis-summer-school.enisa.europa.eu/2019/presentations/Presentation_Thomas%20Roccia.pdf` [online; accessed April 2022].

[340] J. Schmidt, "Orientierung im Security-Babylon," 2020. Article for Heise Online: `https://www.heise.de/hintergrund/Orientierung-im-Security-Babylon-4892855.html` [online; accessed April 2022].

[341] Luatrix, "OpenCTI Project Overview," 2021. Website: `https://www.opencti.io/` [online; accessed April 2022].

[342] N. Adouani, T. Franco, and S. Kadhi, "TheHive Project Overview," 2021. Website: `https://thehive-project.org/` [online; accessed April 2022].

[343] S. Wilson and S. Frankoff, "UNPACME Project Overview," 2021. Website: `https://www.unpac.me/` [online; accessed April 2022].

[344] R. Hüssy, "URLhaus Project Overview," 2021. Website: `https://urlhaus.abuse.ch/` [online; accessed April 2022].

[345] A. Borges, "Malwoverview Project Overview," 2021. Github Repository: `https://github.com/alexandreborges/malwoverview` [online; accessed April 2022].

[346] Canadian Centre for Cyber Security, "AssemblyLine Project Overview," 2021. Website: `https://cyber.gc.ca/en/assemblyline` [online; accessed April 2022].

[347] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android," in *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2013.

Appendices

A. Windows Malware Families in Malpedia

In the following we provide a full listing of all 839 Windows malware families that have one or more unpacked sample in the form of a memory dump available. Together, they form the set of families used in the evaluations of Chapters 4, 5, and 6. For reference, they are part of Malpedia Git commit `1639cad`, created on January 3rd, 2019.

win.7ev3n	win.9002	win.abaddon_pos	win.abantes
win.abbath_banker	win.acridrain	win.acronym	win.adam_locker
win.adkoob	win.advisorsbot	win.adylkuzz	win.agent_btz
win.agent_tesla	win.alice_atm	win.alina_pos	win.allaple
win.alma_communicator	win.alma_locker	win.alpc_lpe	win.alphabet_ransomware
win.alphalocker	win.alphanc	win.alreay	win.alureon
win.amtsol	win.andromeda	win.anel	win.antilam
win.apocalipto	win.apocalypse_ransom	win.ardamax	win.arefty
win.arik_keylogger	win.arkei_stealer	win.ascentloader	win.asprox
win.athenago	win.ati_agent	win.atmii	win.atmitch
win.atmosphere	win.atmspitter	win.august_stealer	win.aurora
win.avcrypt	win.aveo	win.avzhan	win.ayegent
win.azorult	win.babar	win.babymetal	win.backnet
win.backspace	win.backswap	win.badencript	win.badflick
win.badnews	win.bagle	win.bahamut	win.banatrix
win.bangat	win.banjori	win.bankshot	win.bart
win.batchwiper	win.batel	win.bbsrat	win.bedep
win.beendoor	win.berbomthum	win.bernhardpos	win.betabot
win.biscuit	win.bitsran	win.bka_trojaner	win.blackenergy
win.blackpos	win.blackrevolution	win.blackshades	win.boaxxe
win.bohmini	win.bolek	win.bouncer	win.bozok
win.brambul	win.bravonc	win.breakthrough_loader	win.bredolab
win.brutpos	win.bs2005	win.btcware	win.buhtrap
win.bundestrojaner	win.bunitu	win.buterat	win.buzus
win.byeby	win.c0d0so0	win.cabart	win.cadelspy
win.camubot	win.cannon	win.carbanak	win.carberp
win.cardinal_rat	win.carrotbat	win.casper	win.catchamas
win.ccleaner_backdoor	win.centerpos	win.cerber	win.cerbu_miner
win.chainshot	win.chches	win.cherry_picker	win.chewbacca
win.chinad	win.chir	win.chthonic	win.citadel
win.client_maximus	win.cloud_duke	win.cmsbrute	win.cmstar
win.coalabot	win.cobalt_strike	win.cobian_rat	win.cobint
win.cobra	win.cockblocker	win.codekey	win.cohhoc
win.coinminer	win.colony	win.combojack	win.combos
win.comodosec	win.computrace	win.concealment_troy	win.conficker
win.confucius	win.contopee	win.cookiebag	win.corebot
win.coreshell	win.cradlecore	win.crashoverride	win.credraptor
win.crenufs	win.crimson	win.crisis	win.cryakl
win.crylocker	win.crypmic	win.crypt0l0cker	win.crypto_fortress
win.crypto_ransomeware	win.cryptolocker	win.cryptoluck	win.cryptomix
win.cryptorium	win.cryptoshield	win.cryptoshuffler	win.cryptowall
win.cryptxxxx	win.csext	win.cuegoe	win.cueisfry
win.cutlet	win.cutwail	win.cyber_splitter	win.cybergate

Table A.1.: List of Windows malware families used, Part I.

win.cycbot	win.dairy	win.danabot	win.darkcomet
win.darkmegi	win.darkmoon	win.darkpulsar	win.darkshell
win.darksky	win.darkstrat	win.darktequila	win.darktrack_rat
win.daserf	win.datper	win.ddkong	win.decebal
win.deltas	win.dented	win.deputydog	win.deria_lock
win.derusbi	win.devils_rat	win.dexter	win.dharma
win.diamondfox	win.dimnie	win.dircrypt	win.dma_locker
win.dnspionage	win.dorkbot_ngrbot	win.dorshel	win.doublepulsar
win.downdelph	win.downeks	win.downpaper	win.dramnudge
win.dreambot	win.dridex	win.dropshot	win.dtbackdoor
win.dualtoy	win.dubrute	win.dumador	win.duqu
win.duuzer	win.dyre	win.eda2_ransom	win.ehdevel
win.elirks	win.elise	win.emdivi	win.enfal
win.erebus	win.eredel	win.eternal_petya	win.etumbot
win.evilbunny	win.evilgrab	win.evilpony	win.evrial
win.excalibur	win.exchange_tool	win.extreme_rat	win.fakedga
win.fakerean	win.faketc	win.fanny	win.fast_pos
win.felismus	win.felixroot	win.feodo	win.fileice_ransom
win.final1stspy	win.findpos	win.finfisher	win.fireball
win.firecrypt	win.firemalv	win.flawedammyy	win.floki_bot
win.flusihoc	win.fobber	win.formbook	win.former_first_rat
win.freenki	win.friedex	win.furtim	win.galaxyloader
win.gamapos	win.gameover_dga	win.gameover_p2p	win.gamotrol
win.gandcrab	win.gaudox	win.gauss	win.gazer
win.gcman	win.gearinformer	win.geodo	win.getmail
win.getmypass	win.ghole	win.ghost_admin	win.ghost_rat
win.ghostnet	win.glasses	win.glassrat	win.globe_ransom
win.globeimposter	win.glooxmail	win.glupteba	win.godzilla_loader
win.goggles	win.gold_dragon	win.golroted	win.goodor
win.google_drive_rat	win.goopic	win.gootkit	win.govrat
win.gozi	win.gpcode	win.grabbot	win.graftor
win.grateful_pos	win.gratem	win.gravity_rat	win.greenshaitan
win.gsecdump	win.h1n1	win.hacksfase	win.hackspy
win.hamweq	win.hancitor	win.happy_locker	win.harnig
win.havex_rat	win.hawkeye_keylogger	win.helauto	win.helminth
win.heloag	win.herbst	win.hermes	win.hermes_ransom
win.herpes	win.hesperbot	win.hi_zor_rat	win.hiddentear
win.hikit	win.himan	win.hlux	win.homefry
win.htbot	win.htprat	win.htran	win.http_troy
win.httpbrowser	win.httpdropper	win.hworm	win.hyperbro
win.ice_ix	win.icedid	win.icedid_downloader	win.icefog
win.idkey	win.imminent_monitor_rat	win.infy	win.innaput_rat
win.invisimole	win.isfb	win.ismagent	win.ismdoor
win.ispy_keylogger	win.isr_stealer	win.isspace	win.jackpos
win.jaff	win.jager_decryptor	win.jaku	win.jasus
win.jigsaw	win.jimmy	win.joanap	win.joao
win.jolob	win.jqjsnicker	win.jripbot	win.kagent
win.karagany	win.kardonloader	win.kasperagent	win.kazuar
win.kegotip	win.kelihos	win.keyboy	win.keylogger_apt3
win.keymarble	win.keypass	win.khrat	win.kikothac
win.killdisk	win.kins	win.kleptoparasite_stealer	win.klrd
win.koadic	win.kokokrypt	win.konni	win.koobface
win.korlia	win.kovter	win.kpot_stealer	win.kraken
win.krbanker	win.krdownloader	win.kronos	win.ksl0t
win.kuaibu8	win.kuluoz	win.kurton	win.kwampirs
win.lambert	win.lamdelin	win.latentbot	win.lazarus
win.laziok	win.leash	win.leouncia	win.lethic
win.limitail	win.listrix	win.litehttp	win.lock_pos
win.locky	win.locky_decryptor	win.logedrut	win.logpos
win.lokipws	win.lordix	win.luminosity_rat	win.lurk
win.luzo	win.lyposit	win.madmax	win.magala
win.magniber	win.makadocs	win.makloader	win.maktub
win.malumpos	win.manamecrypt	win.mangzamel	win.manifestus

Table A.2.: List of Windows malware families used, Part II.

win.manitsme	win.mapiget	win.marap	win.matrix_banker
win.matrix_ransom	win.matryoshka_rat	win.matsnu	win.mbrlock
win.mebromi	win.mewsei	win.miancha	win.micrass
win.microcin	win.micropsia	win.milkmaid	win.mimikatz
win.miniasp	win.mirage	win.miragefox	win.mirai
win.misdat	win.misfox	win.miuref	win.mm_core
win.mobi_rat	win.mocton	win.moker	win.mokes
win.molerat_loader	win.monero_miner	win.moonwind	win.morphine
win.morto	win.mosquito	win.moure	win.mozart
win.mpkbot	win.multigrain_pos	win.murkytop	win.murofet
win.mutabaha	win.mykings_spreader	win.mylobot	win.n40
win.nabucur	win.nagini	win.naikon	win.nanocore
win.narilam	win.nautilus	win.navrat	win.necurs
win.nemim	win.neteagle	win.netrepser_keylogger	win.netsupportmanager_rat
win.nettraveler	win.netwire	win.neutrino	win.neutrino_pos
win.new_ct	win.newcore_rat	win.newposthings	win.newsreels
win.nexster_bot	win.nexus_logger	win.ngioweb	win.nitlove
win.nitol	win.njrat	win.nocturnalstealer	win.nokki
win.nozelesn_decryptor	win.nymaim	win.nymaim2	win.oceansalt
win.octopus	win.oddjob	win.odinaff	win.oldbait
win.onekeylocker	win.onhat	win.onionduke	win.onliner
win.oopsie	win.opachki	win.opghoul	win.orcarat
win.orcus_rat	win.ordinypt	win.overlay_rat	win.ovidiystealer
win.owaauth	win.padcrypt	win.paladin	win.pandabanker
win.parasite_http	win.penco	win.petrwrap	win.petya
win.pgift	win.phandoor	win.phorpiex	win.pipcreat
win.pirpi	win.pittytiger_rat	win.pkybot	win.plaintee
win.playwork	win.plead	win.ploutus_atm	win.ployx
win.plugx	win.pngdowner	win.poison_ivy	win.polyglot_ransom
win.pony	win.poohmilk	win.popcorn_time	win.poscardstealer
win.poweliks_dropper	win.powerduke	win.powerpool	win.powersniff
win.predator	win.prikormka	win.prilex	win.psix
win.pss	win.pteranodon	win.punkey_pos	win.pushdo
win.putabmow	win.pvzout	win.pwnpos	win.pykspa
win.qaccel	win.qadars	win.qakbot	win.qhost
win.qtbot	win.quant_loader	win.quasar_rat	win.r980
win.radamant	win.radrat	win.rakhni	win.rambo
win.ramdo	win.ramnit	win.ranbyus	win.ranscam
win.ransoc	win.ransomlock	win.rapid_ransom	win.rapid_stealer
win.rarstar	win.ratabankapos	win.rawpos	win.rcs
win.rdasrv	win.reactorbot	win.reaver	win.red_alert
win.red_gambler	win.redalpha	win.redleaves	win.redyms
win.regin	win.remcos	win.remexi	win.remsec_strider
win.rerdom	win.retefe	win.revenge_rat	win.rgdoor
win.rifdoor	win.rikamanu	win.rincux	win.rising_sun
win.rockloader	win.rofin	win.rokku	win.rokrat
win.rombertik	win.romeos	win.roopirs	win.roseam
win.royal_dns	win.royalcli	win.rtm	win.rtpos
win.ruckguv	win.rumish	win.runningrat	win.rurktar
win.rustock	win.sage_ransom	win.sakula_rat	win.salgorea
win.sality	win.samsam	win.sanny	win.sarhust
win.satan	win.satana	win.sathurbot	win.scanpos
win.scote	win.screenlocker	win.seasalt	win.sedll
win.sedreco	win.seduploader	win.sendsafe	win.serpico
win.shadowpad	win.shakti	win.shapeshift	win.shareip
win.sharpknot	win.shelllocker	win.shifu	win.shimrat
win.shujin	win.shurl0ckr	win.shylock	win.sidewinder
win.sierras	win.siggen6	win.silence	win.silon
win.siluhdur	win.simda	win.sinowal	win.sisfader
win.skarab_ransom	win.skyplex	win.slave	win.slingshot
win.smac	win.smokeloader	win.smominru	win.snatch_loader
win.sneepy	win.snifula	win.snojan	win.snslocker
win.sobaken	win.socks5_systemz	win.socksbot	win.solarbot

Table A.3.: List of Windows malware families used, Part III.

win.sorgu	win.soundbite	win.spedear	win.spora_ransom
win.spybot	win.squirtdanger	win.sslmm	win.stabuniq
win.starcruft	win.starsypound	win.stegoloader	win.stinger
win.stration	win.stresspaint	win.strongpity	win.stuxnet
win.sunorcal	win.suppobox	win.swift	win.sword
win.sykipot	win.synccrypt	win.synflooder	win.synth_loader
win.sys10	win.sysget	win.sysraw_stealer	win.sysscan
win.tabmsgsql	win.taidoor	win.taleret	win.tandfuy
win.tapaoux	win.tarsip	win.tdiscoverer	win.telebot
win.tempedreve	win.terminator_rat	win.teslacrypt	win.thanatos
win.thanatos_ransom	win.threebyte	win.thumbthief	win.thunker
win.tidepool	win.tinba	win.tinyloader	win.tinynuke
win.tinytyphon	win.tinyzbot	win.tiop	win.tofsee
win.torrentlocker	win.treasurehunter	win.trickbot	win.trochilus_rat
win.troldesh	win.trump_ransom	win.tsifiri	win.turnedup
win.tyupkin	win.uacme	win.udpos	win.uiwix
win.unidentified_001	win.unidentified_003	win.unidentified_006	win.unidentified_013
win.unidentified_020	win.unidentified_022	win.unidentified_023	win.unidentified_024
win.unidentified_029	win.unidentified_030	win.unidentified_031	win.unidentified_032
win.unidentified_033	win.unidentified_035	win.unidentified_037	win.unidentified_038
win.unidentified_039	win.unidentified_041	win.unidentified_042	win.unidentified_044
win.unidentified_045	win.unidentified_047	win.unidentified_048	win.unidentified_049
win.unidentified_051	win.unidentified_052	win.unidentified_053	win.unidentified_054
win.unlock92	win.upas	win.upatre	win.urausy
win.urlzone	win.uroburos	win.vawtrak	win.velso
win.venus_locker	win.vflooder	win.virdetdoor	win.virut
win.vmzeus	win.vobfus	win.volgmer	win.vreikstadi
win.vskimmer	win.w32times	win.wannacryptor	win.waterminer
win.waterspout	win.webc2_adspace	win.webc2_ausov	win.webc2_bolid
win.webc2_cson	win.webc2_div	win.webc2_greencat	win.webc2_head
win.webc2_kt3	win.webc2_qbp	win.webc2_rave	win.webc2_table
win.webc2_ugx	win.webc2_yahoo	win.webmonitor	win.wellmess
win.winmm	win.winsloader	win.wipbot	win.wmighost
win.wndtest	win.wonknu	win.woody	win.woolger
win.xagent	win.xbot_pos	win.xbtl	win.xpan
win.xpctra	win.xsplus	win.xtunnel	win.xtunnel_net
win.xxmm	win.yahoyah	win.yayih	win.younglotus
win.yty	win.zebrocy	win.zedhou	win.zeroaccess
win.zerot	win.zeus	win.zeus_mailsniffer	win.zeus_openssl
win.zeus_sphinx	win.zezin	win.zhcat	win.zhmimikatz
win.zloader	win.zoxpng	win.zyklon	

Table A.4.: List of Windows malware families used, Part IV. Some unidentified families had been identified over time, which results in gaps between their labels.

B. YARA rules used to detect MSVC and zlib

```
rule detect_msvc {
 meta:
  author = "Daniel Plohmann"
  description = "Detect presence of MSVC fragments via characteristic functions"

 strings:
  // __except_handler4
  $msvcrt_0 = { 33 C0 64 8B 0D 00 00 00 00 81 79 04 ?? ?? ?? ?? 75 10 8B 51 0C 8B 52
    0C 39 51 08 75 05 B8 01 00 00 00 c3 }
  $msvcrt_1 = { 68 ?? ?? ?? ?? 64 FF 35 00 00 00 00 8B 44 24 10 89 6C 24 10 8D 6C 24
    10 2B E0 53 56 57 A1 }
  $msvcrt_2 = { 55 8B EC 8B 45 14 50 8B 4D 10 51 8B 55 0C 52 8B 45 08 50 68 ?? ?? ??
    ?? 68 ?? ?? ?? ?? E8 ?? ?? ?? ?? 83 C4 18 5D C3 }
  $msvcrt_3 = { 83 C4 04 8B 55 08 89 02 8B 45 14 50 8B 4D 10 51 8B 55 0C 52 8B 45 08
    50 68 ?? ?? ?? ?? 68 ?? ?? ?? ?? E8 ?? ?? ?? ?? 83 C4 18 5D C3 }
  // allshl
  $msvcrt_4 = { 80 F9 40 73 15 80 F9 20 73 06 0F A5 C2 D3 E0 C3 8B D0 33 C0 80 E1 1F
    D3 E2 C3 33 C0 33 D2 C3 }
  // free(Block)
  $msvcrt_5 = { 55 8B EC 51 89 4D FC 8B 45 08 50 E8 ?? ?? ?? ?? 83 C4 04 8B E5 5D C2
    04 00 }
  // __allmul
  $msvcrt_6 = { 8B 44 24 08 8B 4C 24 10 0B C8 8B 4C 24 0C 75 09 8B 44 24 04 F7 E1 C2
    10 00 53 F7 E1 8B D8 8B 44 24 08 F7 64 24 14 03 D8 8B 44 24 08 F7 E1 03 D3 5B C2
    10 00 }
  // ___raise_securityfailure
  $msvcrt_7 = { 55 8B EC FF 15 ?? ?? ?? ?? 6A 01 A3 ?? ?? ?? ?? E8 ?? ?? ?? ?? FF 75
    08 E8 ?? ?? ?? ?? 83 3D ?? ?? ?? ?? 00 59 59 75 08 6A 01 E8 ?? ?? ?? ?? 59 68 09
    04 00 C0 E8 ?? ?? ?? ?? 59 5D C3 }
  // __SEH_prolog
  $msvcrt_8 = { 68 ?? ?? ?? ?? 64 A1 00 00 00 00 50 8B 44 24 10 89 6C 24 10 8D 6C 24
    10 2B E0 53 56 57 8B 45 F8 89 65 E8 50 8B 45 FC C7 45 FC FF FF FF FF 89 45 F8 8D
    45 F0 64 A3 00 00 00 00 C3 }
  // alloca_probe
  $msvcrt64_2 = { 48 83 EC 10 4C 89 14 24 4C 89 5C 24 08 4D 33 DB 4C 8D 54 24 18 4C
      2B D0 4D 0F 42 D3 65 4C 8B 1C 25 10 00 00 00 4D 3B D3 73 16 66 41 81 E2 00 F0
      4D 8D 9B 00 F0 FF FF 41 C6 03 00 4D 3B D3 75 F0 }
  // __finally
  $msvcrt64_3 = { 40 55 48 83 EC 20 48 8B EA 48 63 4D 20 48 8B C1 48 8B 15 ?? ?? ??
      ?? 48 8B 14 CA E8 ?? ?? ?? ?? 90 48 83 C4 20 5D C3 }
  $msvcrt64_0 = { 48 83 EC 28 48 8B 01 81 38 63 73 6D E0 75 1C 83 78 18 04 75 16 8B
      48 20 8D 81 E0 FA 6C E6 83 F8 02 76 0F 81 F9 00 40 99 01 74 07 33 C0 48 83 C4
      28 C3 }
  $msvcrt64_1 = { 48 8B 44 24 40 48 89 05 ?? ?? ?? ?? C7 05 ?? ?? ?? ?? 09 04 00 C0
      C7 05 ?? ?? ?? ?? 01 00 00 00 C7 05 ?? ?? ?? ?? 01 00 00 00 B8 08 00 00 00 48
      6B C0 00 48 8D 0D ?? ?? ?? ?? 48 C7 04 01 02 00 00 00 B8 08 00 00 00 48 6B C0
      00 48 8B 0D }

 condition:
  any of them
}
```

```
rule detect_zlib {
 meta:
  author = "Daniel Plohmann"
  description = "Detect presence of zlib characteristic functions and strings"

 strings:
  // inflate array A
  $values_0 = { 00 01 02 03 04 05 06 07 08 08 09 09 0A 0A 0B 0B 0C 0C 0C 0C 0D 0D 0D
    0D 0E 0E 0E 0E 0F 0F 0F 0F 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11 12 12
    12 12 12 12 12 12 13 13 13 13 13 13 13 13 14 14 14 14 14 14 14 14 14 14 14 14 14
    14 14 14 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 16 16 16 16 16 16 16 16
    16 16 16 16 16 16 16 16 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 18 18 18
    18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
    18 18 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
    19 19 19 19 19 19 19 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
    1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B
    1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1C 00 01 02 03 04 04 05 05 06 06
    06 06 07 07 07 07 08 08 08 08 08 08 08 08 09 09 09 09 09 09 09 09 0A 0A 0A 0A 0A
    0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B
    0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C
    0C 0C 0C 0C 0C 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D
    0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E
    0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E
    0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0F 0F 0F 0F 0F 0F 0F
    0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
    0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
    0F 0F 0F 00 00 10 11 12 12 13 13 14 14 14 14 15 15 15 15 16 16 16 16 16 16 16 16
    17 17 17 17 17 17 17 17 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 19 19 19
    19 19 19 19 19 19 19 19 19 19 19 19 19 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
    1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1B 1B 1B 1B 1B 1B 1B 1B 1B
    1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1C 1C 1C 1C
    1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
    1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
    1C 1C 1C 1C 1C 1C 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D
    1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D
    1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D }
  // inflate array B
  $values_1 = { 00 01 02 03 04 04 05 05 06 06 06 06 07 07 07 07 08 08 08 08 08 08 08
    08 09 09 09 09 09 09 09 09 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0B 0B
    0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C
    0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0D 0D 0D 0D 0D 0D 0D 0D
    0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0D 0E 0E 0E
    0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E
    0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E 0E
    0E 0E 0E 0E 0E 0E 0E 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
    0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F
    0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 00 00 10 11 12 12 13 13 14 14
    14 14 15 15 15 15 16 16 16 16 16 16 16 16 17 17 17 17 17 17 17 17 18 18 18 18 18
    18 18 18 18 18 18 18 18 18 18 18 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
    1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
    1A 1A 1A 1A 1A 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B
    1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
    1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C
    1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1D 1D 1D 1D 1D 1D 1D
    1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D
    1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D
    1D 1D 1D 00 01 02 03 04 05 06 07 08 08 09 09 0A 0A 0B 0B 0C 0C 0C 0C 0D 0D 0D 0D
    0E 0E 0E 0E 0F 0F 0F 0F 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11 12 12 12
    12 12 12 12 12 13 13 13 13 13 13 13 13 14 14 14 14 14 14 14 14 14 14 14 14 14 14
    14 14 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 16 16 16 16 16 16 16 16 16
    16 16 16 16 16 16 16 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 18 18 18 18
    18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18
    18 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19
    19 19 19 19 19 19 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A
```

```
    1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B
    1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1C }
  // characteristic strings
  $version = "inflate 1.2."
  $version11 = "inflate 1.1."
  $version10 = "inflate 1.0."
  // we did not find any inflate version below 1.0.4
  $length = "incorrect length check"

 condition:
  any of them
}
```