Robust Static Analysis of Portable Executable Malware

by Katja Hahn
Register INM 11

Master Thesis in Computer Science

HTWK Leipzig
Fakultät Informatik, Mathematik und Naturwissenschaften

First Assessor: Prof. Dr. rer. nat. habil. Michael Frank (HTWK Leipzig)
Second Assessor: Prof. Dr. rer. nat. Karsten Weicker (HTWK Leipzig)

Leipzig, December 2014

Abstract

The Portable Executable (PE) format is an architectural independent file format for 32 and 64-bit Windows operating systems. PE format related properties that violate conventions or the specification are called PE malformations. They can cause problems to any program that parses PE files, among others, crashes, refusal to read the file, extracting and displaying wrong information, and inability to parse certain structures. Malware authors use PE malformations to avoid or prolong malware analysis and evade detection by antivirus scanners.

This master thesis analyses PE malformations and presents robust PE parsing algorithms. A static analysis library for PE files named PortEx serves as example. The library is hardened successfully against 103 275 PE malware samples and a collection of 269 malformed proof-of-concept files. PortEx is also used to extract statistical information about malicious and clean files, including anomalies in the PE format. The author identifies 31 file properties for heuristic analysis of PE files.

Contents

Chapter 1

Introduction

Malware analysis and detection tools are vulnerable to malformations. Malware uses, among others, malformed Portable Executable (PE) structures to break or deceive them. The present chapter discusses how defence mechanisms by malware arose and how robust parsing of PE files can be achieved.

1.1 Purpose of Malware Analysis

The primary task of employees of the first antivirus companies was manual analysis of every malware to find out detection and disinfection mechanisms for the antivirus software they produced. With the rise of newly created malware to over 200 000 pieces per day in 2014[1], the extraction of new detection signatures is mainly automated today. General detection methods, which cover a broad range of malware at once, have been added as well.

Cohen has proven that the detection of viruses is an undecidable problem (see [Coh84]). So it comes of no surprise that there are cases where automatic malware analysis and general detection methods fail. Employees in antivirus companies perform manual or semi-automated analysis for these cases and they help to understand new attacks, malware defence techniques, and trends. Other malware analysts are hired to prevent and respond to malware related incidents within a company. They figure out the damage that was done and perform disinfection and recovery if possible.

1.2 PE Format-Related Malware Defence

Malware authors have developed techniques to deceive or break malware analysis and detection tools. One group of these anti-analysis techniques involves the

[1] AV-Test, May 2014, `http://www.av-test.org/en/statistics/malware/`

modification of PE format properties or structures to malformed ones. The PE format describes the buildup of EXE and dynamic-link library (DLL) files on Windows operating systems.

The main problem is the gap of knowledge around the behaviour of the Windows operating system while it loads and executes a PE file. The developers of analysis tools rely on the documentation of the PE format, which does not reflect reality. As a result, malware authors are able to apply modifications to PE files, that the Windows operating system treats differently than it is expected by the developers of PE analysis tools. The tools break, refuse to load the file, or show incorrect information.

1.3 Robust PE Malware Analysis

The present thesis focalises on malware analysis based on information that is extracted from the PE format. It covers the implementation of a PE format parser that is robust against malformed files and additional analysis techniques that build upon the format parser. These techniques include the recognition of PE file anomalies, PE file visualisation, entropy and hash value calculation, string extraction, packer identification, and detection of embedded files.

The algorithms for the aforementioned analysis techniques are collected in a library—called *PortEx*—and provided via a programming interface for software developers. The implementation solely relies on the information of the file on disk. The analysed malware is not executed, which means there is no risk of infection.

The robustness of the parser is achieved by the following steps:

- research about PE format malformations
- implementation of anomaly detection
- robustness tests with 103 275 malicious files and 269 malformed proof-of-concept files
- searching the file samples for formerly unknown or undocumented malformations
- compiling a collection of file samples with malformations
- adjusting the naïve implementation to a robust one, among others, by partial simulation of the PE loading process that the operating system applies

The visualisation of a PE file's structure provides a quick overview as people can process images more easily than a logfile consisting of addresses, names, and

values. The visualisation is not sufficient on its own, but helps to find where to put the focus for further analysis.

String extraction, entropy and hash value calculation, packer identification, and embedded file detection are not new, but the author considers them as basic tools for malware analysis.

PortEx is also used to collect statistical information about PE files. The present thesis compares PE malware with clean PE files based on the statistical results and proposes how to use the findings for further analysis or detection of PE malware.

The author's intention is that tools building upon the library will not suffer from malformed PE files, and that documentation and compilation of malformed files enables other developers to write more robust software. The anomaly detection may also help malware analysts to correct properties of PE files that formerly broke other analysis tools.

1.4 Limitations

The software arsenal for malware analysis includes various tools for different tasks. The choice of tools depends on the malware type, the affected operating systems and file types, and the malware's behaviour. There is no single tool that suffices for every task and stage of malware analysis. Likewise, it is not the goal of the library *PortEx* to cover all steps that are involved in analysing malware. *PortEx* is limited to the information the PE format provides and to analysis techniques that do not require the malware to be executed.

The present thesis does not implement instruction-based analysis techniques or analysis techniques that run the malware. These are only of interest if they lead to a defence response of malware that affects the implementation of basic static analysis tools.

1.5 Roadmap

Chapters 2 and 3 set the theoretical basis for the present thesis. Chapter 2 introduces malware types, file infection strategies by viruses, malware analysis, antivirus detection, and the countermeasures to malware analysis and detection techniques. Chapter 3 presents the PE format and malformations of PE files. This includes already known malformations, as well as malformations that are discovered during the implementation of *PortEx*.

Requirements of *PortEx*, technologies, API design, robust parsing, and implementation of other features of *PortEx* are covered in chapter 4.

Chapter 5 compares the features of *PortEx* to similar tools and libraries for basic static analysis, evaluates their robustness, and concludes with statistical information extracted by *PortEx* and possible uses in malware detection and analysis algorithms.

Chapter 2

Malware

Malware—the short term for *malicious software*—is every program that 'causes harm to a user, computer, or network' [SH12, p. xxviii].

The present chapter lays the theoretical foundations for malware analysis, including the malware's mechanisms to avoid detection by antivirus programs since they also affect analysis tools.

The following section provides an overview of the different types of malware with regard to their behaviour. An overview of file infection strategies is given in section 2.2. Section 2.3 introduces common techniques of malware analysis, which explains the role of the library *PortEx* in the analysis process. Section 2.4 describes how antivirus programs detect malware. Countermeasures to malware detection and analysis are the topic of section 2.5.

2.1 Malware Types

Malware analysis is the focus of the present thesis. As such, it is important to know typical behaviours of malware, how it intrudes and infects a system, how it hides and how it harms. Malware analysts and antivirus companies differentiate and name malware, among others, by its behaviour (see [Mic14]). The following behavioural types are not mutually exclusive, malware can combine the features of several types.

Definition 1 (Hijacker) *A* hijacker *modifies browser settings without the user's consent.*

Typical hijackers replace the default search engine, the browser's home page, error pages, and new tab page with their own content. They also make it difficult to revert the changes and may cause problems if the user tries to uninstall them. Examples are *Conduit Search Protect* and *Babylon Toolbar*.

Definition 2 (Trojan) *A* trojan horse*, or short* trojan*, is a malicious program that tricks the user into running it by providing useful functionality or making the user believe that it is useful and benign. (cf. [Szo05, p. 37])*

Trojan horses are combined with features of, e. g., downloader, dropper, backdoor, information stealer. Trojans make up the majority of malware. In the first quarter of 2014 PandaLabs has detected '15 million new malware strains' of which 71.85 per cent where trojans (see [Pan14]).

Definition 3 (Downloader) *A* downloader *is a piece of software that downloads other malicious content, e. g., from a website, and installs or executes it. (cf. [SH12, p. 3] and [Szo05, p. 39])*

Definition 4 (Dropper) *A* dropper *is a program that writes other malware to the file system. The dropper may perform installation routines and execute the malware. (see [Sym14] and cf. [SH12, pp. 39, 40])*

The difference to a downloader is that the dropper already contains the malicious code in itself (see [Mic14]).

Definition 5 (Rootkit) *A* rootkit *is a software that has the purpose of hiding the presence of other malicious programs or activities. (cf. [SH12, p. 4])*

A rootkit may conceal login activities, log files and processes. Rootkits are often coupled with backdoor functionality (see definition 6).

Definition 6 (Backdoor) *A* backdoor *allows access to the system by circumventing the usual access protection mechanisms. (cf. [SH12, p. 3])*

The backdoor is used to get access to the system later on. A special kind of backdoor is the remote administration tool (RAT) (see [Ayc06, p. 13]). It allows to monitor and access a computer far off, and is used for both malicious and non-malicious purposes. A RAT enables help desk staff to fix computer problems remotely; and employees can work from home by accessing the company's computer. RATs are used maliciously, e. g., for fun and voyeurism. Such RAT users play pranks, spy on people via webcam, and try to scare their victims by controlling their machine. The *ArsTechnica* article *Meet the men who spy on women through their webcams*[1] describes the aforementioned activities by users of `hackforums.net`.

Definition 7 (Spammer) Spam-sending malware*, or short* spammers*, use the victim's machine to send unsolicited messages—so called* spam*. (cf. [SH12, p. 4] and [Szo05, pp. 40, 41])*

Spammers may send their messages, e. g., as email, SMS, or postings and comments in online communities.

Definition 8 (Stealer) *An* information stealer*, or short* stealer*, is a malicous program that reads confidential data from the victim's computer and sends it to the attacker. (cf. [SH12, p. 4])*

[1] `http://arstechnica.com/tech-policy/2013/03/rat-breeders-meet-the-men-who-spy-on-women-through-their-webcams/` (last access Thursday 16th October, 2014)

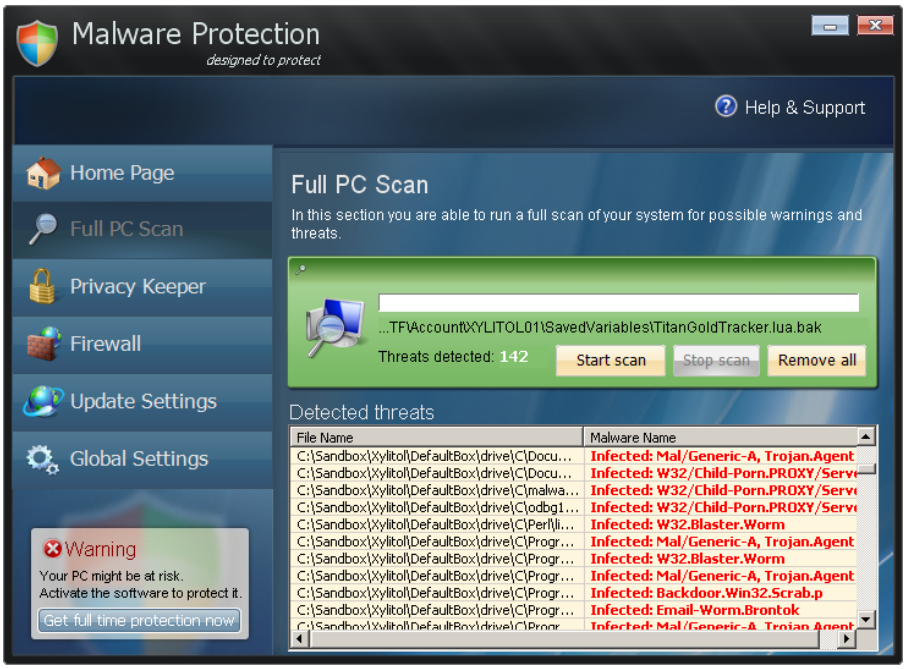

Figure 2.1: The scareware *Malware Protection* pretends it has found infections on the system to trick the user into buying a license

Examples for information stealers are: keyloggers, sniffers, password hash grabbers (see [SH12, p. 3]), and also certain kinds of trojans. A trojan stealer convinces the users that it is benign to make them input confidential data. An example is a program that claims to add more money to the user's PayPal account; actually it sends the PayPal credentials the user puts into the program to the attacker's email address.

Definition 9 (Botnet) *A* botnet *is a group of backdoors installed on different machines that recieve and execute instructions from a single server. (cf. [SH12, p. 3])*

Botnets are installed without consent of the computer's owners and may be used to perform distributed denial of service (DDoS) attacks or to send spam (see definition 7).

Definition 10 (Scareware) Scareware *tries to trick users into buying something by frightening them. (cf. [SH12, p. 4])*

A typical scareware example is a program that looks like an antivirus scanner and shows the user fake warnings about malicious code that was found on the system. It tells the user to buy a certain software in order to remove the malicious code. Figure 2.1 shows the scareware *Malware Protection*, which pretends to be an antivirus software.

Figure 2.2: Malware construction kit example

Definition 11 (Malware construction kit) *A* malware construction kit *is a program that generates malware or malware sources based on user-defined settings. (cf. [Szo05, p. 261])*

Malware construction kits enable people without any programming knowledge to create their custom malware. Simpler kits just change small settings, like email addresses or FTP accounts that are the recipients for information that the malware found on the victim's computer. More sophisticated kits employ anti-analysis and anti-detection techniques and may generate a wide range of different malware binaries. Figure 2.2 shows an example of a malware construction kit with anti-analysis features.

Definition 12 (Virus) *A* virus *recursively replicates itself by infecting or replacing other programs or modifying references to these programs to point to the virus code instead. A virus possibly mutates itself with new generations. (cf. [Szo05, p. 27, 36])*

A typical virus will be executed if the user executes an infected file. Such an infected file is called *host file* referring to the terminology that is used for biological parasites. Viruses traditionally spread to other computers via transported media like floppy disk, USB flash drive, CD, or DVD.

A virus is called *germ* if it is in its original form, prior to any infection (see [Szo05, p. 39]). The initial installation of the germ code is done by a dropper, afterwards the virus can 'replicate on its own' (see [Szo05, p. 40]).

An *intended* virus failed to replicate due to a bug or an incompatible environment, e. g., an operating system that it was not written for (see [Ayc06, p. 14]).

Dormant viruses are in a passive state, they reside on the machine without infecting anything, either waiting for a trigger or a compatible system to spread to (cf. [Ayc06, p. 14]). The opposite to *dormant* is *active*.

Definition 13 (Worm) *'*Worms *are network viruses, primarily replicating on networks.' [Szo05, p. 36]*

Typically, worms do not need a host file and execute themselves without the need of user interaction (see [Szo05, p. 36]). There are exceptions from that, e. g., worms that spread by mailing themselves need user interaction. A worm is a subclass of a virus by definition 13.

A report by PandaLabs about the newly created malware strains in the first quarter of 2014 reveals: 10.45 per cent have been viruses and 12.25 per cent have been worms (see [Pan14]). Note that PandaLabs does not see *worms* as a subclass of *viruses*, but as mutually exclusive groups in contrast to definition 12. That means 22.70 per cent of newly created malware strains detected at PandaLabs have been viruses by definition 12.

2.2 File Infection Strategies

The PE file format is one host format of file infecting viruses. File infections can introduce malformations because most infection types require modifications of the host file. Therefore, it is useful to know file infection strategies of viruses to understand how certain malformations occur, but also for the identification of infection types during analysis.

Depending on the file infection strategy it is sometimes possible to remove or disable the malicious code from an infected file. This process is called *disinfection* and performed by antivirus software. Disinfection does not necessarily restore the file to its original form.

The following infection strategies work for most executable file formats, including the PE format.

Overwriting

Overwriting viruses employ the simplest strategy. They search for other files on disk and copy their own body in place of them (see [Szo05, p. 115]). Infections by overwriting viruses can cause severe damage to the system because they destroy the files they are overwriting and render disinfection impossible. Users can recognise the side effects of the damage soon, which is why this infection strategy is usually not very successful (see [Szo05, p. 115]).

A variant of the overwriting virus only replaces the beginning of the host file with its own body (see [Szo05, p. 116]). The infected file keeps the original size of the host file. The overwritten part of the host file is destroyed.

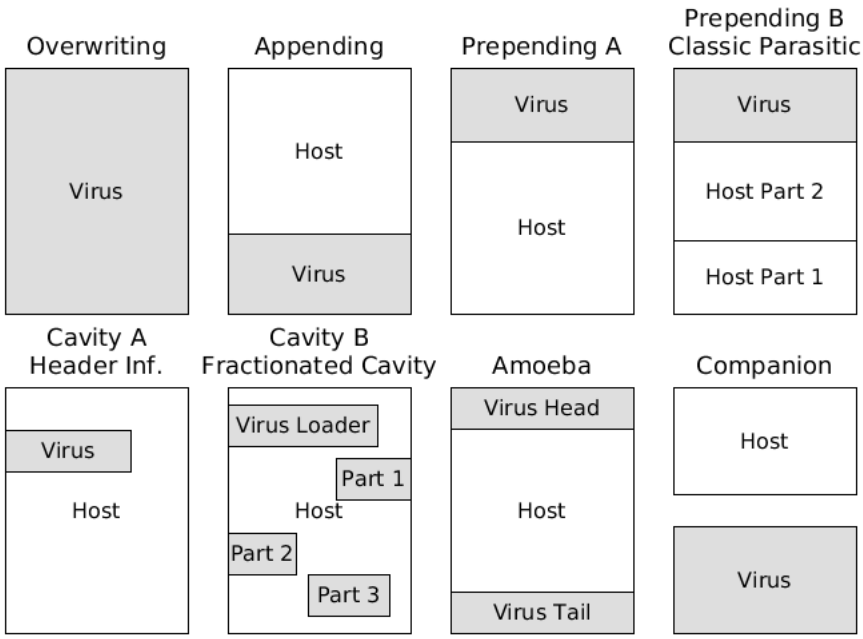

Figure 2.3: File infection strategies

Appending

Appending viruses write their own body after the end of the host file (see figure 2.3) and modify a jump instruction at the beginning of the host file to point to the virus code (see [Szo05, p. 117]). The appending virus typically passes control to the host file again after it has done its tasks, so the user does not get aware of the infection (see [Szo05, p. 118]).

Some file formats, like the PE format, define an entry point, which is an address that points to the start of code execution. Viruses appending to these file formats may change the address of the entry point to point to the virus code, or add a jump instruction right after the entry point to jump to the virus code.

Files that were infected by appending viruses usually can be disinfected.

Prepending

Infection by prepending is done by writing the virus code to the front of the file (see [Szo05, pp. 118–120]). The original file stays intact, which makes disinfection possible.

To hide the infection from the user, the prepending virus executes the host file, e. g., by copying the host file as temporary file to disk and using a function call like `system()` to run the host file (see [Szo05, p. 120]).

A subtype of infection by prepending is the *classic parasitic infection* (see *Prepending B* in figure 2.3). A virus that employs this strategy replaces the beginning of the host file with its own body and appends the overwritten part to the host file (see [Szo05, pp. 120, 121]).

Cavity Infection

Cavity viruses overwrite parts of the host file that are not necessarily used by the host file (see [Szo05, p. 121]). These are usually parts that contain only zeroes—so called *caves* of the file (see [Szo05, p. 121]). An example is the cave between the file header and the actual start of the file. If a virus infects this cave, it is called *header infection* (see *Cavity A* in figure 2.3). Viruses that infect a single cave must be small enough to fit into them.

The cavity virus may save the original entry point of the host file and pass control to the host file after it is done.

A subtype of the cavity infection is the *fractionated cavity infection* (see *Cavity B* in figure 2.3), where the virus splits itself and writes the code fractions into several caves. The first fraction contains the loader code, which is responsible to put the virus' fractions back together in memory (see [Szo05, pp. 122, 123]).

Disinfection of a host file with cavity infection can be complicated and is sometimes impossible if the overwritten parts cannot be restored (see [Szo05, p. 123]).

Amoeba Infection

This infection strategy is rare according to Szor [Szo05, p. 124]. The virus splits itself into two parts. The head is prepended to the host file and the tail is appended. If the file is executed the head will load the tail of the virus to execute it.

The amoeba virus may reconstruct the host file, write it to disk as temporary file and execute it.

Companion Infection

Companion viruses do not modify the host file. They take advantage of the order the operating system executes files, so they are executed instead of the host file.

There are three types of companion viruses:

1. The *regular companion* places itself in the same directory as the host file, having the same filename, but a different file extension. If the user only gives the filename without the extension, the regular companion will be executed instead of the host file (see [Bon94, section 2.2.1]). For example

files with the extension .COM are searched before files with the extension .EXE on MS-DOS. A companion virus can infect files with .EXE extension by placing itself in the same directory and the same filename with a .COM extension (see [Bon94, section 2.2.1]).

2. The *PATH companion* takes advantage of the order of directories that the operating system uses to search for files. Operating systems usually have a variable called PATH that determines the search order for executable files. The PATH companion has the same name as its host file and places itself in a directory that takes precedence over the directory of the host file. (see [Bon94, section 2.2.2]). An example are viruses that mimic common DLL files. Applications that import functions of a DLL can be tricked into loading the functions of the virus instead of the actual DLL file.

3. The *alias companion* uses user-defined command-line macros, aka aliases, to get executed (see [Bon94, section 2.2.2]). Aliases are used as a shortcut for long command sequences. A virus can create an alias that replaces a common command with execution of the companion (see [Bon94, section 2.2.3]).

The companion virus exists beside its host file. Deletion of the companion file or the alias in case of an alias companion will remove the infection.

2.3 Malware Analysis

Definition 14 'Malware analysis *is the art of dissecting malware to understand how it works, how to identify it, and how to defeat or eliminate it.' [SH12, p. xxviii]*

Malware analysis is not only performed by employees of antivirus companies, but also respondents to computer security incidents of other companies. These may be security specialists, system engineers, or network adminstrators. They perform malware analysis to find out which machines and programs are affected, how the malware can be removed, what vulnerabilities of the system it used, which data has been destroyed or sent to the attacker, and how to prevent another intrusion.

There are two kinds of malware analysis—static and dynamic. Malware analysts usually apply both, starting with basic static analysis to get an overview of the file. The two kinds of malware analysis, their techniques, and tools are explained in the present section.

2.3.1 Static Analysis

Definition 15 Static analysis *is the examination of a program without running it (see [SH12, p. 2]).*

Static analysis includes, e. g., viewing file format information, finding strings or patterns of byte sequences, disassembling the program and subsequent examination of the intructions.

Static analysis is limited: The malware may be encrypted, compressed or otherwise obfuscated. If instructions are examined, static analysis can be extensive because all possible paths of execution have to be considered. But static analysis is preferred for the first examination because it is safe: The code is never executed, thus, the malware cannot cause any harm.

Basic Static Analysis

A malware analyst uses basic static analysis to get an overview and make first assumptions about the malware's function or behavioural type (cf. [SH12, p. 9]).

Sikorski and Honig compare basic static analysis to 'looking at the outside of a body during an autopsy' [SH12, p. 65]. This analogy illustrates two facts about basic static analysis: first, the malware is a *dead body* (it is not run); second, the malware's inner workings, its instructions and paths of execution, are not part of basic static analysis as it is only looked at from the outside.

Basic static analysis employs the following techniques (see [SH12, p. 9]):

1. 'Using antivirus tools to confirm maliciousness' [SH12, p. 9]
2. 'Using hashes to identify malware' [SH12, p. 9]
3. Getting information from 'strings, functions, and headers' of a file (see [SH12, p. 9])

Sikorski and Honig recommend VirusTotal[2] (see [SH12, p. 10]), which is a website that generates antivirus reports about uploaded files. The reports tell how many and which antivirus engines detect the file as malicious and may include additional information for certain file types using automated static analysis.

MD5 and SHA-256 hashes are commonly used to label, identify, and find malware that was already analysed (see [SH12, p. 10]). VirusTotal allows to search for reports by hash values, which are either computed from the whole file or sections of it. Malware sharing sites like Open Malware[3] allow search by hash as well.

Binary files may contain strings, like error messages, dialog text, sometimes also names of functions, imports, exports, URLs, or email addresses. Programs like *strings.exe*[4] filter character sequences from binary files and display them (cf. [SH12, pp. 11, 12]).

[2] `https://www.virustotal.com/` (last access Thursday 16th October, 2014)
[3] `http://www.offensivecomputing.net/` (last access Thursday 16th October, 2014)
[4] `http://technet.microsoft.com/en-us/sysinternals/bb897439.aspx` (last access Thursday 16th October, 2014)

Packer identifiers (packed malware is part of subsection 2.5.2) and parsers for certain file formats are also tools of basic static analysis.

Advanced Static Analysis

Advanced static analysis is the examination of a program's instructions (cf. [SH12, pp. 65–85]).

These instructions may be present in different levels of abstraction. A low abstraction level is machine code, which consists of machine instructions, so called *opcodes*. The lowest human-readable abstraction level is assembly language. Assembly language assigns mnemonics to machine instructions and operations and allows the usage of symbolic addresses and labels, thus avoids manual address calculations by the programmer. High-level code is, e. g., C source code.

Assembly and high-level languages have to be translated to machine code to be executed. The translation from a higher to a lower abstraction level is called *compilation*. The process of translating assembly language to machine code is also called *assembly* or *assembling*.

Compiled malware is usually translated back into a human-readable form before examination. This translation is done via one of the following:

1. decompilation: translation from a lower-level code to high-level language code
2. disassembly: translation from machine code to assembly [SH12, p. 66]

Figure 2.4 illustrates relations of (de-)compilation and (dis-)assembly and the different abstraction levels of code.

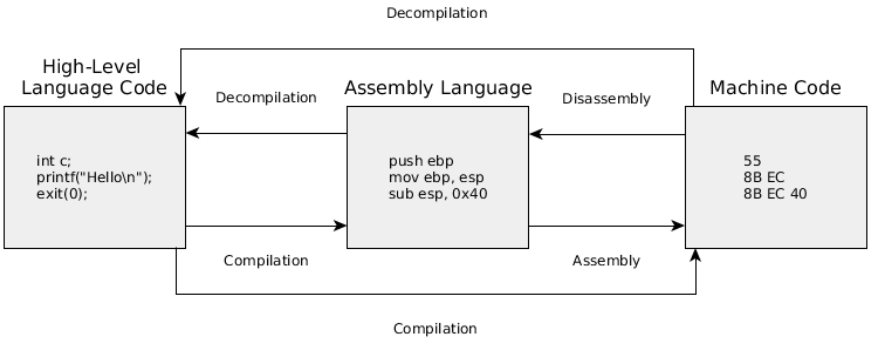

Figure 2.4: Compilation, assembly, and reversing processes; referring to [SH12, p. 67]

Decompilation is the reverse process to compilation. It translates the program's code from a lower abstraction level into a high-level language code, e. g., from

Java bytecode (*.class* files) into Java source code (*.java* files). Decompilation is usually not able to reconstruct the original source code because there is an information loss in the compilation process (cf. [SH12, p. 67]).

Decompilation works well if the program consists of code that preserves most information. Examples are the decompilation of Java bytecode to Java source code and decompilation of Common Intermediate Language (CIL) code to C#.

Disassembly is the translation of machine code into assembly language (cf. [SH12, p. 67]). Sikorski and Honig state: 'Assembly is the highest level language that can be reliably and consistently recovered from machine code when high-level language source code is not available.' [SH12, p. 67] Therefore, disassembly is preferred over decompilation if only machine code is available.

2.3.2 Dynamic Analysis

Definition 16 Dynamic analysis *is the examination of a program while running it (see [SH12, p. 2]).*

Dynamic analysis includes, e. g., observing the program's behaviour in a virtual machine (VM) or a dedicated testing machine, or examining the program in a debugger. It is usually performed 'after basic static analysis has reached a dead end' [SH12, p. 39].

Dynamic analysis is able to circumvent anti-static analysis tricks like packing and obfuscation. It only registers the execution path that the malware takes during observation. This saves time compared to examining all paths with static analysis techniques, provided that the actual malicious behaviour is observed.

However, malware is able to trick dynamic analysis by testing the presence of a virtual environment or dynamic analysis tools and behaving differently. The malware might not show any malicious activity or just terminate. There is also the chance for the malware to exploit bugs in the sandbox environment and infect or harm the host computer or other computers on the network. So dynamic analysis comes with a risk (cf. [SH12, p. 40]).

Basic Dynamic Analysis

Basic dynamic analysis is a fast way to examine the malware's behaviour. The malware is executed and observed with monitoring programs.

Techniques of basic dynamic analysis include:

1. automatic analysis using sandboxes (see [SH12, pp. 40–42])
2. monitoring processes (see [SH12, pp. 43–50])
3. monitoring file changes (see [Szo05, pp. 586–588])

4. comparing registry snapshots (see [SH12, p. 50])
5. faking a network and monitoring transmissions (see [SH12, pp. 51–56])
6. infection tests with goat files (see [Szo05, pp. 588–590])
7. tracing system calls (see [Szo05, pp. 596, 597])

Dynamic analysis needs a safe environment that can be restored to a clean state. That is either a dedicated physical machine or a VM. Tools like *Norton Ghost*[5] allow to backup and restore the state of a physical machine. VMs have the ability to take *snapshots*, which are also backups of the current state.

Some companies provide sandboxes for malware analysis. These sandboxes are VMs or emulators that automatically generate a report about the tested file. The reports may contain information about network activitiy, registry changes, file operations, and also static analysis results of the malware (cf. [SH12, p. 41]). An example for such a sandbox is *cuckoo*[6].

Typical tools for malware observation are process and file monitoring tools like Process Monitor[7] and Process Explorer[8]. They track registry changes, file activities (e. g. modification, creation), spawning of child processes, and more.

The Windows registry stores configurations for the operating system. Malware changes Windows registry entries, e. g., to become persistent by making the system execute the malware after boot. Malware analysts take registry snapshots before and after execution of the malware. Afterwards they compare both snapshots with a tool that filters the differences. The resulting report shows all changes done to the Windows registry.

Certain malware types, like downloaders and stealers, will only show their malicious behaviour if there is a working Internet connection. Having a working Internet connection during dynamic analysis is risky because the malware might spread via the network. That is why there are tools to simulate network services. Examples are INetSim[9], which simulates the Internet, and ApateDNS[10], which redirects DNS requests.

These network programs work together with packet capture and analyser tools like Wireshark[11]. They intercept and log any outcoming and incoming network traffic, thus, make the traffic visible for the malware analyst. They, e. g., recognise the

[5] `http://www.symantec.com/themes/theme.jsp?themeid=ghost` (last access Thursday 16th October, 2014)
[6] `http://www.cuckoosandbox.org/` (last access Thursday 16th October, 2014)
[7] `http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx` (last access Thursday 16th October, 2014)
[8] `http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx` (last access Thursday 16th October, 2014)
[9] `http://www.inetsim.org/` (last access Thursday 16th October, 2014)
[10] `https://www.mandiant.com/resources/download/research-tool-mandiant-apatedns` (last access Thursday 16th October, 2014)
[11] `https://www.wireshark.org/` (last access Thursday 16th October, 2014)

attempt of a downloader to access files on the Internet, or see what information a stealer sends and where to.

Goat files are created by malware researchers to analyse file infection strategies of viruses (see [Szo05, p. 588]). Typical goat files contain only do-nothing (NOP) instructions (see [Szo05, p. 221]). If a virus infects such a file, the infected parts are readily distinguishable from the non-infected parts (see [Szo05, p. 588]). Thus, goat files ease the extraction of the virus body and help to understand the infection technique.

Advanced Dynamic Analysis

The main tool for advanced dynamic analysis is a *debugger* (see [SH12, p. 167]). Debuggers are used to examine a software while it is running. A debugger can show and change contents of variables, registers, parameters, and memory-locations, modify instructions, step through the execution path one instruction at a time (*single-step*), and pause the execution at predefined points called *breakpoints*.

Source-level debuggers are mostly used by software developers to test their products. They operate on high-level language code. Assembly-debuggers are more relevant for malware analysts and reverse engineers because they work if high-level language code is not available (see [SH12, p. 168]).

Debugging is either done in *user mode* or in *kernel mode*. These are processor privilege levels of Windows (see [SH12, p. 158]). Programs usually run in *user mode*. Exceptions are operating system code and hardware drivers (see [SH12, p. 158]). Processes that run in *kernel mode* share memory addresses and resources, whereas processes in *user mode* have their own memory, resources, and more restrictions regarding available instructions, registers, and hardware access (see [SH12, p. 158]). There are some malicious programs that run in *kernel mode*. Malware analysts, who want to perform advanced dynamic analysis on such malware, need a debugger that supports kernel-debugging, e. g., *WinDbg*[12].

2.4 Malware Detection by Antivirus Software

The detection techniques of antivirus products influence how malware defends itself. Thus, it is necessary to know about detection techniques to understand malware defence.

Antivirus scanners—despite their name—detect not only viruses, but malware of any kind. There are two types of scanners. *On-demand scanners* are executed by the user if he or she wants to analyse a certain file (see [Szo05, p. 391]). *On-access*

[12] `http://msdn.microsoft.com/en-us/windows/hardware/hh852365.aspx` (last access Thursday 16th October, 2014)

scanners reside in memory and scan files on certain events, e. g., when files are 'opened, created or closed' [Szo05, p. 392].

Detection techniques are also divided into *static* and *dynamic* ones. *Static detection* does not run the code, whereas *dynamic detection* observes the malware's behaviour live or uses an emulator. The following sections describe malware detection techniques that are used by antivirus scanners.

2.4.1 String Scanning

String scanning is a static detection method. It compares byte sequences in a database with the actual byte sequences of a file (see [Szo05, p. 393]). If a subsequence in the file matches one of the sequences in the database, the file is declared malicious (see [Szo05, p. 393]). Such a byte sequence to identify a malware is also called *signature* or *string*. The signature needs to be unique enough, so that it is unlikely to exist in clean programs (see [Szo05, p. 393]).

This technique is only able to recognise known malware. Therefore, string scanners need regular updates of the signature database to be able to detect newly created malware.

Extensions of this method include the use of wildcards in the signature, the use of regular expressions that represent a group of byte sequences, or the allowance for a certain number of byte mismatches (see [Szo05, pp. 395–397]).

Example 1 *The following listing demonstrates the string scanning extensions* wildcards *and* mismatches *with matching and mismatching byte sequences for a sample signature.*

```
1  Wildcards
2  ---------
3  signature:           0E 1F 0E 07 E8 ?? ?? E8 ?? ?? 3A C6 73
4  byte sequence found: 0E 1F 0E 07 E8 31 C4 E8 00 FF 3A C6 73
5  --> match
6  byte sequence found: 0E 00 0E 07 E8 A5 C4 E8 22 CF 3A C6 73
7  --> mismatch            --
8
9  Mismatches allowed: 2
10 ---------------------
11 signature:           73 72 63 64 6F 63 69 64 3A 20
12 byte sequence found: 73 72 63 02 6F 63 69 64 00 20
13 --> match                  --                --
14 byte sequence found: C6 A2 63 64 6F 63 69 64 3A 00
15 --> mismatch         -- --                      --
```

One signature usually represents one piece of malware, but there are also signatures that represent a group. Malicious programs that have common roots and characteristics are grouped into a *family*. These family members or *variants* of a malware emerge, when malware writers modify existing malicious code, or use construction kits to build a personalized version of a certain kind of malware. Signatures that cover several members of a malware family at once are called *generic detection signatures*.

Definition 17 (Generic Detection) *'Generic detection scans for several or all known variants of a family of computer viruses using a simple string' or 'an algorithmic detection'. [Szo05, p. 397]*

Generic detection is also able to detect new variants of a known malware family, as long as the part containing the signature was not modified.

2.4.2 Speed and Accuracy Improvements for String Scanning

A *naïve string scanning algorithm* looks up every subsequence of bytes in a file with the signature database that belongs to the antivirus scanner. Antivirus companies strive to create antivirus software that does not noticeably slow down the clients' computers. The following methods are used to improve speed and detection accuracy of string scanning techniques. Their effect on size and location of the scanned area is demonstrated in Figure 2.5.

Hashing

Hashing does not affect the scanned area of the file, but it improves the speed to look up signatures in the database. A hash value is created from the first 16 to 32 bytes of a scanned byte sequence (see [Szo05, p. 397]). The hash value is used as index for the hash table that contains the corresponding signatures. The signatures must not contain any wildcards because the hashes would not match if a single byte was changed (see [Szo05, p. 397]). Some scanners, therefore, do not allow any wildcards; others only hash a prefix of the signature and allow the rest of the signature to contain wildcards (see [Szo05, p. 397]). Hashing can be combined with any of the other string scanning improvement techniques below.

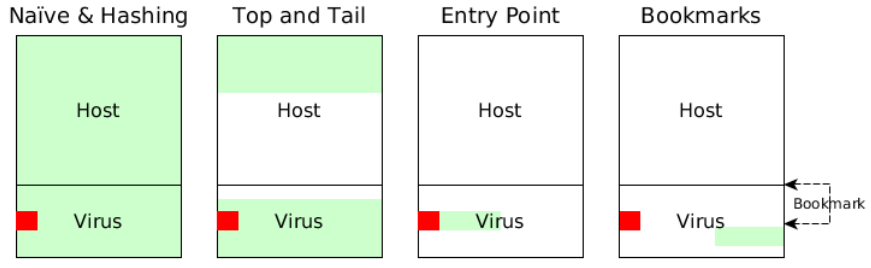

Figure 2.5: The scanned area (green) of an infected file with different string scanning strategies applied. The red rectangle marks the entry point. The virus example is an appending file infector that modifies the entry point of the host file to point to the virus body.

Top-and-Tail Scanning

Instead of scanning the entire file for a signature match, only the top and tail of the file are used. A top-and-tail scanner takes, e. g., only the first and the last 2 kb of the file into account (see [Szo05, p. 398]). This is especially useful to detect prepending, appending, or overwriting viruses (see [Szo05, p. 398]). Top-and-tail scanning improves the speed, but may fail to recognise cavity infections.

Entry Point Scanning

The entry point of a file marks the offset where the execution starts. The entry point is often defined in headers of the file format. Viruses commonly target the entry point, e. g., they modify it to point to the virus code. Because of that it is possible to improve scanning speed and accuracy by scanning for a signature at the entry point (see [Szo05, p. 399]). Entry point scanning can only be applied for signatures that are located at the entry point. Entry point signatures are marked as such in the signature database.

Bookmarks

Bookmarks are saved alongside the malware signature (see [Szo05, p. 397]). They represent the distance from the start of the malware body to the detection string (see [Szo05, p. 397]). Bookmarks increase detection accuracy and speed (see [Szo05, p. 397]).

Bookmarks are also useful to save information necessary for disinfection. Szor states 'it is a good practice to choose bookmarks that point to an offset to the stored original host program header bytes. Additionally, the size of the virus body stored in the virus is also a very useful bookmark.' [Szo05, p. 398]

2.4.3 Algorithmic Scanning

The standard detection algorithm of an antivirus scanner cannot cover every malware (see [Szo05, p. 405]). Malware-specific detection algorithms are necessary for such threats. While early implementations were actually hardcoded procedures in the scanner, the products today use portable code that is part of the malware detection database (see [Szo05, p. 405]).

These additional detection algorithms worsen the performance, which is why algorithmic scanning applies filtering techniques to avoid unnecessary scanning (see [Szo05, p. 406]). A malware-specific filter defines, e. g., the affected file type, certain values or flags in the header, or file characteristics that are typical for the malware (see [Szo05, p. 406]). The detection routine is only executed for a specific file if the file passes the filter.

2.4.4 Integrity Checking

All file infecting viruses that are not companion viruses rely on file modifications. Integrity checkers detect file modifications to determine suspicious changes (see [Ayc06, p. 70]). Integrity checking is a static detection technique.

An integrity checker has a database of checksums for all files that shall be watched (see [Ayc06, p. 70]). These checksums must be initially computed from a clean system (see [Ayc06, p. 70]). The integrity checker uses the checksums to determine if modifications have been performed on a file (see [Ayc06, p. 70]).

There are three types of integrity checkers depending on their implementation:

- An *offline integrity checker* compares the checksums periodically, e. g., every week (see [Ayc06, p. 70]).
- *Self-checking programs* perform the integrity check on themselves upon execution (see [Ayc06, p. 70]). This is a typical defence mechanism for antivirus software (see [Ayc06, p. 71]).
- *Integrity shells* perform the checksum comparison immediately before execution of a program (see [Ayc06, p. 71]).

Alternative terms for the three types are described in [Rad94, p. 8].

While integrity checkers are able to detect known and unknown threats, they can only alarm the user after the infection occured (see [Ayc06, p. 80]). They cannot distinguish between legitimate file modifications and unauthorised ones (see [Ayc06, p. 80]). This decision is passed to the user (see [Ayc06, p. 80]).

2.4.5 Code Emulation

Code emulation is a dynamic detection technique. An emulator simulates the operating system, the memory management, the central processing unit, and other hardware of the system (see [Ayc06, pp. 75, 76]). The emulator imitates the execution of a suspicious program (see [Szo05, p. 413]). The data that was gathered via emulation is analysed using *dynamic heuristic analysis* (see [Ayc06, p. 74], subsection 2.4.6).

Code emulation is also used to apply generic decryption to encrypted malware. The emulator simulates the malware's decryption routine (see [Ayc06, p. 74]). It determines heuristically when the malware has finished decrypting itself (see [Ayc06, p. 75]). The antivirus scanner subsequently identifies the plain body of the malware using string scanning or other static detection methods (see [Szo05, p. 415]).

2.4.6 Heuristic Analysis

Heuristic methods find a solution for a problem using incomplete knowledge. Heuristic analysis describes all malware detection methods that use 'a rule-based approach to diagnosing a potentially-offending file' [HL, p. 6]. These detection methods are not optimal, they create false positives and false negatives; but they are able to recognise unknown threats and detect variants of known malware. Heuristic analysis is done in two steps: data gathering and data analysis.

Step 1: Data Gathering

The heuristic scanner collects patterns of the file. The collected patterns are grouped into one of two types: *boosters* or *stoppers*.

Definition 18 (booster) Boosters *are patterns in heuristic analysis that indicate malware-like behaviour or appearance (see [Ayc06, p. 69]). They increase the likelyhood for the file to be identified as malicious (see [Ayc06, p. 69]).*

Definition 19 (stopper) Stoppers *are patterns in heuristic analysis that indicate behaviour or appearance that is untypical for malware. Stoppers decrease the likelyhood for the file to be indentified as malicious (see [Ayc06, p. 69]).*

Static heuristic analysis collects information about the file format and the file's instructions (see [Szo05, p. 211]). Typical boosters for static heuristic analysis are blacklisted strings found in the file (e. g. obscene words or the term *virus*), use of undocumented functions, presence of unusual instructions, no existing caves (indicator of cavity-infection), or self-modifying code (see [Szo05, p. 211] and [Ayc06, p. 69]).

Example 2 *Some file-infecting viruses add jump instructions right after the entry point to pass execution to the virus body. Therefore, jump instructions right after the entry point are an indicator for a file infection and classified as* booster *by heuristic scanners.*

Dynamic heuristic analysis collects data of a program by emulating its execution (see subsection 2.4.5) or observing its behaviour live on the system. Boosters are suspicious behavioural patterns. These include registry changes, file modifications, replication, and hiding techniques.

Step 2: Data Analysis

The analysis of the data collected in step 1 may involve artificial neural networks or other machine learning techniques. It can also be done by assigning weights to the boosters and stoppers and calculating the sum; the analysed file is identified as malicious if the sum is larger than a predefined threshold (see [Ayc06, p. 70]).

2.4.7 Comparison of Malware Detection Methods

String scanning is able to identify malware precisely if non-generic methods are used (see [Ayc06, p. 79]). Thus, it enables malware-specific disinfection of the system. String scanning cannot cope with unknown threats, unless the threat is a variant of a known malware and covered by a generic detection signature.

Algorithmic scanning is the last resort if signature extraction for a malicious file is not possible. The algorithms are part of the signature database of string scanners; as such they rely on database updates. It takes time for malware analysts to examine the malware and create a malware-specific detection algorithm. Algorithmic scanning is able to identify known malware, and can only detect unknown threats if they are covered by a generic detection algorithm.

Integrity checkers 'boast high operating speeds and low resource requirements' [Ayc06, p. 80]. They are only effective against file infecting viruses. They detect known and unknown threats and create false positives. Integrity checkers cannot identify malware.

Code emulation is able to detect known and unknown threats using dynamic heuristic analysis, with the possibility of false positives. Emulators are also able to decrypt most encrypted malware, which is the first step for further analysis, e. g., by string scanning. Emulators are slow and their presence can be detected by malware. Code emulation is safe compared to observation of programs that already run on the system.

Heuristic analysis can be static or dynamic. It detects known and unknown malware. False positives are possible, exact identification is impossible, disinfection can only be done with generic methods.

There is no single superior detection method that suffices for every case. While known threats are handled well with signature scanning and algorithmic scanning, the detection of unknown threats always imposes the risk of false positives. It is necessary to apply several detection methods to get the best results. Their implementation is a tradeoff between detection rate, accuracy, and performance.

2.5 Anti-Detection and Anti-Reversing

The previous section dealt with malware detection methods. Malware authors react to these methods by applying defence mechanisms to malware. Such malware employs techniques to evade detection and hamper malware analysis. Malware defence increases the skill level that is necessary to reverse-engineer the malware, thus, can delay or prevent analysis. The present section gives an overview to anti-detection and -reversing techniques and discusses the place and impact of file-format related defence.

2.5.1 Obfuscation

Definition 20 (Obfuscation) Obfuscation *is the deliberate act of making machine code or higher level code difficult to understand by humans.*

Some obfuscation techniques are:

1. substitution of variable names or subroutine names with deceiving or non-descriptive strings.
2. encoding or encryption of strings in a binary file
3. encoding or encryption of byte sequences
4. adding superfluous code, structures, or functions that do nothing useful, or are never executed; e. g., conditional jumps that always yield false
5. breaking conventions; e. g., coding conventions for certain programming languages or conventions for the structure of a file format
6. replacing code or data structures with more complex, but equivalent code or structures

In section 2.3.1, string extraction is described as a basic analysis technique to get passwords, email addresses, messages, or similar information from a file. A countermeasure by malware is to store only encrypted strings and to decrypt them if they are needed.

Malware may not only encrypt plain strings, but also the code of its body, leaving only the decrypter's code plain. This is a countermeasure to string-based detection of antivirus software (see subsection 2.4.1) and malware analysis with disassemblers (see section 2.3.1). As long as the code of the decrypter is long and unique enough, the extraction of signatures is still possible (see [Szo05, p. 234]). But some viruses use a different decrypter for every new generation, which makes signature extraction difficult or impossible. Viruses that have a predefined set of different decrypters are called *oligomorphic*; viruses that mutate their decrypters—thus, are able to create millions of different forms—are called *polymorphic* (see [Szo05, pp. 234, 236]).

Malware authors create obfuscated programs manually or with obfuscation tools. Polymorphic viruses are an example of manually crafted obfuscation by encryption. One commonly used obfuscation tool is the packer, which is explained in the next subsection. Code obfuscation can be partially removed with deobfuscators. Encrypted malware can often be decrypted, either manually or automated.

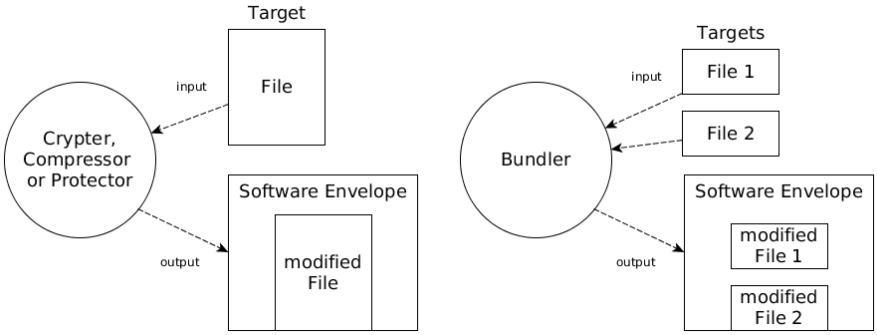

Figure 2.6: Functionality of packers

2.5.2 Packed Malware

Definition 21 (Packer) *A* packer *is an application that takes an executable file, possibly modifies it, and puts it into a 'software envelope'—the* stub*. The resulting file is called* packed file*. If the packed file is executed, the stub recovers the original file and runs it in memory. (cf. [MMF10, p. 1])*

Definition 22 (Target) *The* target *is an executable file 'in its natural form prior to being operated on by a packer.' [MMF10, p. 1]*

The modifications that a packer applies to a target are compression, encryption, anti-unpacking tricks, or other obfuscation methods. Depending on these modifications, packers are categorised into three types (cf. [Sec08, p. 73]):

1. compressor: applies compression to the target
2. crypter: applies encryption to the target
3. protector: applies encryption and compression to the target

Another subtype of a packer is the *bundler*, which puts several targets into a single software envelope (cf. [Sec08, p. 73]). Upon execution, all executable files packed into the software envelope are run in memory. A bundler may as well fall into one of the aforementioned subcategories: compressor, crypter, or protector. The modifications are applied to all targets. The principle functionality of packers is illustrated in Figure 2.6.

Packers have legitimate purposes, like shrinking the file size and protection of the software against illicit reverse engineering. Bundlers ease the distribution of a product because it can be provided as one file compared to having several files.

The same features are used by malware authors. They pack malware with compressors, crypters, or protectors to avoid detection by antivirus software and to exacerbate malware analysis. They use bundlers to bind malicious programs

with legitimate ones. The result is a trojan—the legitimate application serves as a decoy and the malicious code is carried out silently besides the decoy.

Antivirus scanners handle packed malware either with a generic signature that detects the software envelope, or by employing automated unpacking techniques (see subsection 2.4.5). Generic detection of packed files only makes sense if the packer was specifically written for malicious use, otherwise legitimate programs are detected as well. Some malware packers generate a unique stub for every packing process, which makes generic detection difficult.

Static malware analysis of packed files is only possible after unpacking. Generic unpacking can be done by emulation of the stub's unpacking algorithm (see subsection 2.4.5); but there is also the possibilitiy to use packer-specific unpackers or to perform unpacking manually.

2.5.3 Anti-Virtual Machine

Malware uses anti-virtual machine (anti-VM) techniques to avoid its analysis in a VM (see [SH12, p. 369]). If such malware detects that it is run in a VM, it may not show any malicious behaviour, terminate execution, or try to escape the virtual environment to infect the host system (see [SH12, pp. 369,380]).

VM systems leave traces, so called *artefacts*, which malware uses to detect the VM (see [SH12, p. 370]). These artefacts are, e. g., typical processes, registry entries, or files and folders.

Example 3 *To determine if a file is run using VMWare*[13]*, the processes and the registry can be searched for the string* VMWare*. Processes like* VMwareTray.exe *are likely to be found. A registry search might yield the following results (registry example by [SH12, p. 371]):*

```
1  [HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\
       Logical Unit Id 0]
2  "Identifier"="VMware Virtual IDE Hard Drive"
3  "Type"="DiskPeripheral"
4
5  [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Reinstall\0000]
6  "DeviceDesc"="VMware Accelerated AMD PCNet Adapter"
7  "DisplayName"="VMware Accelerated AMD PCNet Adapter"
8  "Mfg"="VMware, Inc."
9  "ProviderName"="VMware, Inc."
10
11 [HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E96F-E325-11CE-BFC1
       -08002BE10318}\0000]
12 "LocationInformationOverride"="plugged into PS/2 mouse port"
13 "InfPath"="oem13.inf"
14 "InfSection"="VMMouse"
15 "ProviderName"="VMware, Inc."
```

A malware analyst has to find and patch detection mechanisms of such malware, or remove artefacts from the VM if possible. To avoid detection of the process *VMwareTray.exe* in example 3, the analyst can uninstall *VMWare Tools* to get

[13] `http://www.vmware.com/` (last access Thursday 16th October, 2014)

rid of the *VMwareTray.exe* process, or modify the string *VMwareTray.exe* in the malware using a hex editor (see [SH12, p. 373]). Malware analysts can also use a different VM that might not be detected by the piece of malware they want to analyse. To reduce the chance that the host system gets infected, malware analysts commonly use a different operating system for the host system than for the VM.

2.5.4 Anti-Emulation

Anti-emulation techniques hinder or prevent analysis by a code emulator. There are three categories of anti-emulation techniques: Malware *outlasts*, *outsmarts*, or *overextends* the emulator (see [Ayc06, p. 99]).

Outlast

Emulation needs more time than static detection techniques; but the time an antivirus program can spend on emulation without loosing the patience of the user is limited. Some malicious programs take time before they show any malicious behaviour. If the emulator cannot spend that much time, the malware *outlasts* the emulator (see [Ayc06, p. 99]).

Outlasting malware might wait for a certain event that triggers the malicious behaviour. If that event is not part of the emulation, the malware will not be detected. Malware also outlasts emulators by delaying execution with do-nothing instructions, garbage instructions, or sleep calls (see [Ayc06, p. 99]). Self-replicating malware might choose to infect only at random. In this case an emulator needs luck to observe self-replication (see [Ayc06, p. 99]).

Emulators countermeasure outlasting by skipping sleep-calls, do-nothing instructions, and repetitive code. The emulator controller may also decide to re-run the emulation, e. g., it can save the branches of execution that have not been taken and emulate them in the next run (see [Ayc06, p. 78]). This enables the detection of malware that runs only sporadically.

Outsmart

Outsmarting is done by restructuring the code or the file format, so it appears harmless (see [Ayc06, p. 100]). This includes avoidance of detection by heuristic analysis (see [Ayc06, p. 100]) and file format malformations. The latter are explained in subsection 2.5.7.

Antivirus companies can only countermeasure outsmarting by improving the emulator or the component that does the heuristic analysis (see [Ayc06, p. 99]).

Overextend

Similar to the anti-VM techniques in subsection 2.5.3 overextending malware tries to detect or even attack the emulator (see [Ayc06, p. 100]).

Emulators do not perfectly imitate a system. They use optimisations to reduce complexity and improve performance (see [Ayc06, p. 78]). This includes the reduction of supported instructions and libraries, and returning fixed values for certain system calls. Malware detects these discrepancies, e. g., by asking the current system time twice (see [Ayc06, p. 100]). If the returned values are the same, the malware concludes that it is emulated (see [Ayc06, p. 100]).

Checks for proper execution of undocumented instructions, or for the availability of rarely used standard libraries and external resources—like websites—can also overextend an emulator (see [Ayc06, p. 100]). The countermeasures for outlasting can also be detected, e. g., an emulator that skips sleep calls is detected by comparison of the system time before and after a sleep instruction.

Countermeasures against overextending malware are improvements to the simulation quality, effectively rising the complexity of the emulator (see [Ayc06, p. 99]).

2.5.5 Anti-Debugging

Malware employs anti-debugging techniques to slow down or prevent its analysis by a debugger. These techniques include the use of debugging detection to behave differently if executed in a debugger.

One debugging detection method uses certain functions of the Windows API, like *IsDebuggerPresent*, or *OutputDebugString* (see [SH12, pp. 352, 353]). Their return values indicate if a debugger is attached to the malware. Other malicious programs re-implement these Windows API functions to achieve the same, and read process information that indicates the use of a debugger (see [SH12, p. 353]).

A malware analyst can modify the malware and force it to take the execution path that it would take without a debugger, or change the process information that indicates a present debugger (see [SH12, p. 354]). Debuggers like OllyDbg have plugins to hide the presence of the debugger (see [SH12, p. 354]).

Another detection technique checks the system for residue of debugging tools (see [SH12, p. 356]). These may be certain registry entries, running processes, or files and folders on the system (see [SH12, p. 356]). A malware analyst handles these techniques similar to the VM artefact detection that was explained in subsection 2.5.3.

Malware also detects debuggers by looking for typical debugger behaviour during execution (see [SH12, p. 356]). As explained in section 2.3.2 debugging involves breakpoints and single-step execution of the file. These actions modify the

code of the process (see [SH12, p. 356]). The modifications can be detected by, e. g., code checksum comparisons. Timing checks are also performed (see [SH12, p. 357]): Process execution is slowed down during examination in a debugger because the debugger halts execution at breakpoints (see [SH12, p. 358]). That means the comparison of timestamps may reveal the presence of a debugger.

Some anti-debugging techniques do not rely on detecting the presence of a debugger, but interfere with its functionality (see [SH12, pp. 359–363]). An example are *thread local storage (TLS) callbacks*. Debuggers usually pause program execution at the entry point of the executable, which is defined in the file's headers; but the instructions at the entry point are not the first ones that are executed (see [SH12, p. 359]). *TLS callbacks* are used to execute code before the entry point, and this code might not be visible in a debugger (see [SH12, p. 359]). *TLS callbacks* can be recognised by basic static analysis (see [SH12, p. 360]). Debuggers like OllyDbg allow to change the settings to pause execution before the entry point (see [SH12, p. 361]).

Debuggers also have vulnerabilities that enables malware to cause debuggers to crash (see [SH12, p. 361]). These are file format malformations that the debugger cannot handle. They are discussed in section 3.5.

2.5.6 Anti-Disassembly

The goal of anti-disassembly is to prevent automated disassembly of machine code and make any code unavailable before it is run (see [Ayc06, p. 103]).

Encrypted and polymorphic viruses (subsection 2.5.1), and packed files (subsection 2.5.2) are examples of anti-disassembly techniques. Other techniques are (cf. [Ayc06, p. 104]):

- dynamically generated code
- code that modifies itself while it is run
- code that was encrypted several times with different keys or encryption algorithms
- encrypted code that is only decrypted and executed in chunks in memory; chunks that are not needed any more, are encrypted again
- encrypted code, whose key was generated from system information, e. g., a hardware ID
- mixing data and code in a way that makes precise separation of them impossible

Anti-disassembly techniques also affect other static analysis methods:

- Decompilation will not be possible if disassembly already fails.
- File format information cannot be extracted if the most part of it is encrypted.
- String extraction of encrypted parts does not yield any readable results.
- Self-modifying code changes the hash values of the file each time it is run, which renders the hash values useless for malware labelling.
- Any analysis that builds upon the disassembled code does not work if disassembly fails.

Encrypted malware sometimes saves the encryption key within the decrypter or uses a vulnerable encryption algorithm. Antivirus scanners identify encrypted malware by using these passwords or vulnerabilities to decrypt them (see [Szo05, p. 409]). The decrypted parts are compared to the signatures in the database for identification of the malware. This is called X-RAY scanning and applied to certain areas of the file, e. g., at the entry point (see [Szo05, p. 409]).

Sometimes, the malware analyst or antivirus scanner must resort to dynamic analysis and heuristics. If static decryption is not possible, dynamic decryption by code emulation may work. Decrypters can be detected dynamically by heuristic analysis of the instructions that are executed during emulation (see [Szo05, p. 419]). Self-modifying malware can be detected by *geometric detection*, which is a dynamic heuristic analysis based on the alterations that the malware does to the file structure (see [Szo05, p. 421]).

2.5.7 File Format Malformations

Antivirus scanners and malware analysis tools use file format parsers to extract information from a file. Antivirus scanners also rely on file format detectors to apply filtering (see subsection 2.4.3) and to determine the appropriate file format parser for further analysis.

File format malformations are unusual structures or properties in a file, which cause malfunctions in file format detectors or parsers. If the file format detector misidentifies the format of a file, the antivirus scanner will not apply the correct steps for further analysis. File format parsers are part of basic-static-analysis tools, disassemblers, debuggers, heuristic scanners, and emulators. As such, file format malformations are anti-disassembly, anti-debugging, anti-static-analysis, anti-heuristics, and anti-emulation techniques.

Jana and Shmatikov demonstrate the vulnerability of file format detectors and parsers (see [JS12]). They prove that malware can evade all of the 36 tested antivirus scanners by using only file format malformations; and they conclude ‘that file processing has become the weakest link of malware defense [by antivirus software]’ [JS12, p. 1].

2.6 Summary

The defence mechanisms of malware are a direct reponse to the detection and analysis techniques by antivirus scanners and malware analysts. Antivirus companies and authors of malware analysis tools react in the same manner by adjusting their products to cope with the updated malware's defence mechanisms.

This arms race is ongoing. The commercialisation of malware construction kits and tools for undetection and protection make the creation of sophisticatedly armoured malware possible for everyone. Because these techniques continuously evolve, it is impossible for antivirus companies to provide complete protection for their customers. It is all the more important that malware analysts and antivirus companies keep up with the new threats.

The present chapter presented file format malformations as 'the weakest link' of defence against malware (see [JS12, p. 1]). The PE format is so complex that it opens opportunities for such malformations. The following chapter introduces the PE format and explains the technical details of PE malformations.

Chapter 3

Portable Executable Format

The library *PortEx* extracts information from the PE format to assist in analysing malware. Knowledge about the PE format is neccessary to understand why the extracted information is useful for malware analysis, how PE malformations work and affect parsers, and how this knowledge can be applied to build robust analysis tools.

Microsoft introduced the PE format in 1993 with the release of Windows NT 3.1. It is the successor of the New Executable (NE) file format for 16-bit systems. The PE format has been incrementally changed since then. It supports not only 32-bit, but also 64-bit system architectures today. The PE format is described in the Microsoft Portable Executable and Common Object File Format Specification (PE/COFF specification) [Mic13].

This chapter defines frequently used terms in section 3.1, presents the PE file structure in section 3.2, and winds up with PE malformations in section 3.5.

3.1 General Concepts

This section explains general concepts and frequent terms that are also used by the PE/COFF specification and necessary to understand the descriptions of the PE format.

PE File Types

The two PE file types are DLL and EXE files. The differentiation between these file types is solely a semantic one.

DLL files are meant to export functions or data that other programs can use. Therefore, they usually only run within the context of other programs. They

can have various file extensions, including *.sys*, *.dll*, *ocx*, *.cpl*, *.fon*, and *.drv* (cf. [Mic07]).

EXE files run in their own process instead of being loaded into an existing process of another program. They usually have the file extension *.exe* and do not export any symbols.

Both file types share the same format and hybrids are possible, e. g., an EXE file that exports symbols.

Special Terms

The following definitions are special terms that are necessary to understand the PE format. These special terms are also used in the PE/COFF specification. The first three definitions are related to creation and usage of EXE and DLL files.

Definition 23 (linker) 'Linking *is the process of collecting and combining various pieces of code and data into a single file that can be loaded (copied) into memory and executed. [...] On modern systems, linking is performed automatically by programs called* linkers*.' [BO11, p. 654] The output of a linker is called* image file*.*

Definition 24 (loader) *A* loader *is a program that loads a file into main memory.*

Definition 25 (image file) Image files *have been processed by a linker and are used as input for the loader of the operating system (cf. [Mic13, p. 8]).*

EXE and DLL files are considered as *image files* by the PE/COFF specification (cf. [Mic13, p. 8]). The relationship of the terms *linker*, *loader*, and *image file* is illustrated in figure Figure 3.1.

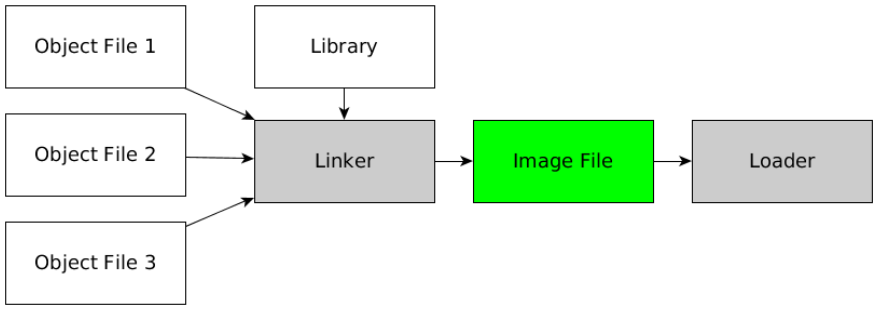

Figure 3.1: The linker combines object files and libraries to an image file, which is used as input by the loader

The following five definitions are related to addresses in the PE format. Addresses are either physical or virtual (in-memory), and either relative or absolute.

Definition 26 (physical address) *A physical address is the offset of a certain byte in a file as it is written on disk.*

Physical addresses are necessary to access parts of the PE file that must be read from disk.

Definition 27 (base address) *The* base address *is the address of the first byte where the image file is loaded in memory (see 'ImageBase' [Mic13, p. 18]).*

PE files specify a preferred base address in a field called `ImageBase`. If the image file cannot be loaded at the preferred address into process space, another base address is applied, which is known as *rebasing*.

Definition 28 (RVA) Relative virtual addresses (RVA) *are used while an image file is loaded in memory. They are relative to the base address of the image file or to another RVA (cf. [Mic13, p. 9]).*

RVAs are a way to specify addresses in memory independently from the base address. This makes it possible to rebase the file without having to re-calculate all in-memory addresses in the file. Because of that they are commonly used in the PE format.

Definition 29 (VA) *Virtual addresses (VA) are absolute in-memory addresses (see [Mic13, p. 9]).*

Although the PE/COFF specification defines a VA this way, it uses the term also for addresses that are actually relative to the image base (e. g., the VA for a data directory entry, see [Mic13, p. 22]).

Definition 30 (entry point) *The* entry point *is a RVA to the starting address for EXE files, or to the initialisation function for device drivers for DLL files (see `AddressOfEntryPoint` [Mic13, p. 17]).*

The entry point was already mentioned in chapter 2 as a common working point for file infecting viruses and antivirus scanners.

Example 4 *An EXE file is loaded to the base address* 0x 40 00 00 *and the entry point is* 0x 31 2E *(a RVA). The effective start of execution is then* 0x 40 31 2E*, which is the VA for the entry point.*

Definition 31 (section) *A 'basic unit of code or data within a PE or COFF file' [Mic13, p. 9] is called a* section*. Sections are defined by their section header in the Section Table (cf. [Mic13, pp. 24–29]).*

3.2 Standard Structure

This section describes the PE format as it is intended by the PE/COFF specification. It differs from the possible structures of PE files in reality, but is used to differentiate between normal and anormal (aka malformed) structures.

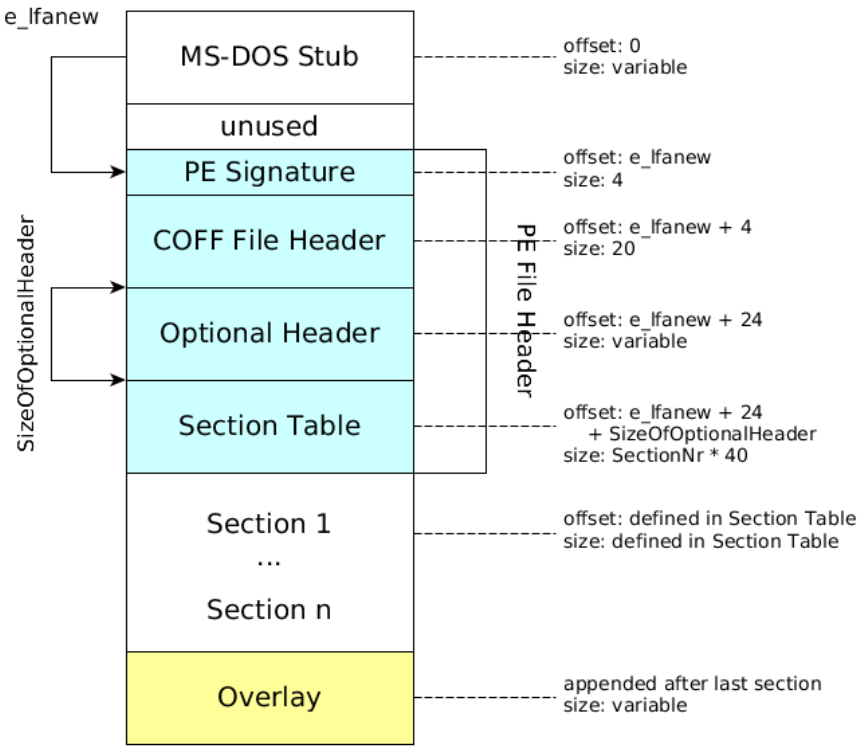

Figure 3.2: Structure of a PE file

Figure 3.2 illustrates the normal structure of a PE file. It consists of the MS-DOS Stub, the PE File Header, and the sections. The overlay is optional data appended to the file. The different parts of the PE are explained hereafter.

A PE file starts with the MS-DOS Stub. This is an application that is able to run in MS-DOS. The standard MS-DOS Stub prints the message 'This program cannot be run in DOS mode'.

The *PE File Header* is placed after the MS-DOS Stub. Different resources use the term *PE File Header* with variable meanings (cf. [Mic13, p. 11], [Rev11] and 'NT Headers' in [Alb13]), so the following definition is used in the present thesis.

Definition 32 (PE File Header) *The* PE File Header *consists of the PE signature, the COFF File Header, the Optional Header, and the Section Table.*

The operating system determines the file format by looking for specific signatures. File format signatures are usually at the very beginning of the file. The PE signature ('PE\0\0') is placed after the MS-DOS-Stub because the MS-DOS Stub has a file format signature ('MZ') itself. The offset to the PE signature is defined in the `e_lfanew` field of the MS-DOS Stub, thus enables Windows to properly execute the PE file.

All substructures of the PE File Header are consecutively arranged, thus located at a fixed offset from the beginning of the PE signature. The offset of the Section Table depends on the `SizeOfOptionalHeaders` field, which is located in the COFF File Header. The remainder of the PE file contains data at addresses, which are defined in the PE File Header.

The sections of the PE file may contain any data, only some sections have a special meaning and are explained in *Special Sections*.

Table 3.1 shows the contents of the MS-DOS Stub and the PE File Header. The `MagicNumber` field of the Optional Header determines whether the image file allows a 64-bit address space (PE32+) or is limited to a 32-bit address space (PE32). Some field sizes vary depending on the `MagicNumber`, and one field has been removed in PE32+ files (`BaseOfData`, see [Mic13, 18]).

Table 3.1: MS-DOS Stub and PE File Header Contents

Name	**Contents**	**Size in Bytes PE32/PE32+**
MS-DOS Stub	among others, the 'MZ' signature and the pointer to the PE Signature `e_lfanew`	variable
PE Signature	the string 'PE\0\0'	4/4
COFF File Header	among others, type of target machine, number of sections, time date stamp (when the file was created), size of Optional Header, file characteristics	20/20
Optional Header	Standard Fields, Windows Specific Fields, Data Directory	variable
Standard Fields	among others, magic number, size of code, linker versions, entry point	28/24
Windows Specific Fields	among others, operating system the file can run on, `ImageBase`, `SizeOfHeaders`, `SizeOfImage`, file alignment, section alignment, DLL characteristics, number of data directory entries	68/88
Data Directory	each entry consists of address and size for a table or string that the system uses, e. g., import table, export table, resource table	variable

Continued on next page

Table 3.1 – *Continued from previous page*

Name	Contents	Size in Bytes PE32/PE32+
Section Table	each entry is a section header; a section header describes, among others, characteristics, size, name, and location of a section	variable

Definition 33 (overlay) *Data that is appended to a PE file and not mapped into memory, is called* overlay*.*

The overlay is used by some applications as a way to store data without having to deal with the PE format or to prevent the operating system from loading the data to memory.

3.3 Special Sections

Sections may contain data that is only relevant for certain applications or not relevant at all; but some sections have special meaning. Their format is described in the PE/COFF specification [Mic13, pp. 65–91].

PE parsers and the Windows loader determine special sections by entries in the Data Directory of the Optional Header or certain flags in the Section Table (cf. [Mic13, p. 65]). Special sections have typical section names, which are also used in the PE/COFF specification to refer to the sections. These names are not mandatory, but a convention. That is why they are not reliable for finding certain sections in a PE. Not only malware writers misuse the section names to obscure their purpose, but also legitimate compilers and packers violate the convention (e. g. the packer UPX [OMR]). A subset of these special sections is described below.

Resource Section

Resources of a PE can be icons, text, windows or copyright information, among others. They are saved as an entry in the *Resource Section*, which also has the name *.rsrc Section*. The Resource Section is build up as a tree. Each path from the root node to a leaf represents one resource. While 2^{31} tree levels can be used according to the PE/COFF specification, Windows only uses three levels with the first level node being the type, the second being the name, and the third being the language information (see [Mic13, p. 89]).

Figure 3.3 illustrates the structure of a resource tree. This example tree has two resources. The first level node is the *Type Directory*. It specifies the type of a

resource by an ID in the *Resource Type* field. The second level nodes are the *Name Directories*. Name Directory entries have an address to the name entry of the resource. The name entry consists of the name length and the actual string in Unicode. The third level nodes are the *Language Directories*. They define the language ID for the resource and have a pointer to a *Data Entry*. The *Data Entry* defines the location of the actual resource bytes.

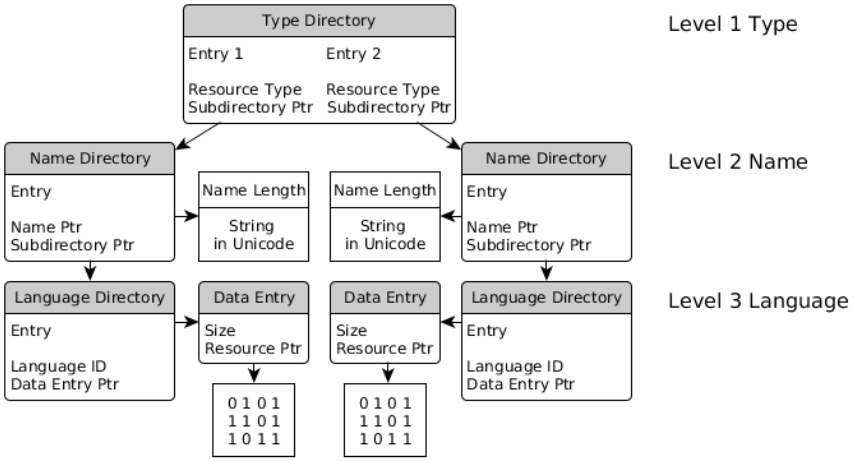

Figure 3.3: Resource tree with two resources, referring to [Kat13]

Export Section

The *.edata Section* or *Export Section* is generally found in DLLs. It is responsible to make data or code available for other PE files. Exported functions or variables are hereby called *symbols*.

Every export has an ordinal number, which is used to obtain the address to the symbol. A programmer, who wants to use an exported symbol from a DLL, must import it. The programmer has two ways to do so: The straightforward way is an import by *ordinal*; but it is also possible to import symbols by a *public name*, which most symbols have. In the latter case the system has to look up the ordinal first. The possibility to import by name exists for convenience (cf. [Pie02b]).

The Export Section begins with the Export Directory Table, which contains general information and addresses to resolve imports from this section. The Export Directory Table points to the Export Address Table (see [Mic13, 82]).

The Export Address Table is an array that contains the addresses to the exported symbols. These addresses either point to code or data within the image file, or forward to the exported symbols of other DLLs (called *forwarder address*). That means a PE file can export symbols that are located in other PE files (cf. [Mic13, p. 75]). The ordinal of a symbol is the index to its address in the Export

Address Table (see [Mic13, p. 73]). The *ordinal base* defines the starting ordinal number for exports (see [Mic13, p. 74]).

Addresses to public names of symbols are in the Export Name Pointer Table. These names are null-terminated ASCII strings. An Ordinal Table is used to map the public names to the corresponding ordinals. Every symbol that has a public name has an entry at the same position in the Ordinal Table and the Export Name Pointer Table (see [Mic13, p. 76]).

Listing 3.1 shows example contents for a DLL with two exported symbols: `DLL2Print` and `DLL2ReturnJ`.

Listing 3.1: Example for Export Section contents, output by *PortEx*

```
Export Directory Table
......................

Minor Version: 0 (0x0)
Address Table Entries: 2 (0x2)
Ordinal Base: 1 (0x1)
Name Pointer RVA: 31664 (0x7bb0)
Export Flags: 0 (0x0)
Ordinal Table RVA: 31672 (0x7bb8)
Number of Name Pointers: 2 (0x2)
Major Version: 0 (0x0)
Time/Date Stamp: 1317493556 (0x4e875b34)
Name RVA: 31676 (0x7bbc)
Export Address Table RVA: 31656 (0x7ba8)

Export Address Table
....................

0x1030, 0x1050

Name Pointer Table
...................

RVA    ->  Name
***************
(0x7bc5,DLL2Print)
(0x7bcf,DLL2ReturnJ)

Ordinal Table
.............

1, 2

Export Entries Summary
----------------------

Name, Ordinal, RVA
..................
DLL2Print, 1, 0x1030
DLL2ReturnJ, 2, 0x1050
```

Example 5 *If `i` is the position of a public name in the Export Name Pointer Table, the address of the symbol will be determined by the following algorithm (see [Mic13, p. 76]).*

```
ordinal = ExportOrdinalTable[i]
symbolRVA = ExportAddressTable[ordinal - OrdinalBase]
```

Thus, the symbol addresses of Listing 3.1 were calculated as follows:

```
publicName = "DLL2ReturnJ"
i = SearchExportNamePointerTable(publicName) // i == 1
ordinal = ExportOrdinalTable[i] //ordinal == 2
//ordinal base == 1 as defined in Export Directory Table
exportAddrTableIndex = ordinal - OrdinalBase // == 1
symbolRVA = ExportAddressTable[exportAddrTableIndex] //symbolRVA == 0x1050
```

Import Section

Every image file that imports symbols has an *Import Section*, also called *.idata Section*. Figure 3.4 provides a general view of its structure. The PE/COFF specification defines the structure of the Import Section at [Mic13, pp. 77–79].

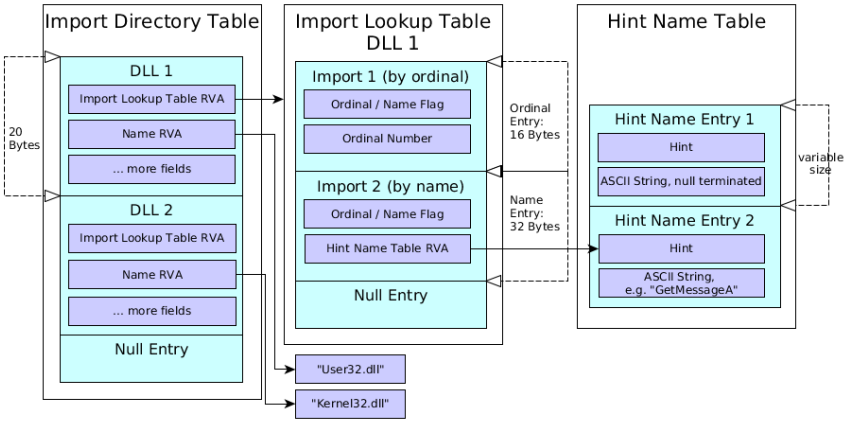

Figure 3.4: Typical Import Section layout

The main structure is the Import Directory Table. Every entry of the Import Directory Table represents the imports from a single DLL. Each entry points to its DLL name and an Import Lookup Table, which represents the imported symbols from that DLL. As explained in section *Export Section* there are two ways to import symbols: by name or by ordinal. Therefore, the structures in the Import Lookup Table either contain the address to a public name or an ordinal. The example in figure 3.4 imports two symbols from *kernel32.dll*, one by ordinal and one by the name *GetMessageA*.

Import Lookup Table entries that import by name (e.g. *Import 2* in figure 3.4), have a pointer to an entry in the Hint-Name Table. Hint-Name Table entries have two fields: a hint and an ASCII name for the import. Each hint is an index to the Export Name Pointer Table (see section 3.3) of the DLL, from which the current file is importing (see [Mic13, p. 79]). Hinting is used to speed up the lookup of imports by name (see [Pie02b]).

Null entries mark the end of the Import Directory Table and the Import Lookup Table (see [Mic13, pp. 77, 78]).

Alongside the Import Lookup Table, there is also an almost identical table, called Import Address Table (IAT). The IAT has the same buildup as the Import Lookup Table. Its is used to *bind* imports. Binding is the process of precomputing and writing actual in-memory addresses of the imported symbols into the IAT before runtime (see [Pie02b]). This way the Windows loader doesn't have to look up the addresses each time they are needed, which speeds up the loading process (see [Pie02b]).

3.4 Mapping in Memory

A PE file is mapped into memory as part of the loading process. This section describes the mapping process in a simplified manner. See figure 3.5 for a visual description.

Figure 3.5: Mapping a PE file from disk (left side) to memory (right side) by [Alb13].

The loader allocates the necessary memory space as defined by the `SizeOfImage` in the Optional Header. Then it maps certain ranges of physical addresses to their virtual counterparts. These ranges represent the location of headers and sections.

The `ImageBase` defines the preferred base address (see definition 27 page 35). The default base address is 0x 40 00 00. The `SizeOfHeaders` defines the number of bytes that are necessary to load the headers up to and including the

Section Table into memory starting at the base address (cf. [Mic13, p. 19]). The `SizeOfHeaders` is 0x 2 00 in figure 3.5.

The sections are mapped after the headers using fields from the Section Table. The virtual address ranges of the sections are defined by the `VirtualAddress` (relative start of the section in memory) and `VirtualSize` (size of the section in memory, cf. [Mic13, pp. 24,25]). Their physical counterparts are the `PointerToRawData` (start of the section on disk) and `SizeOfRawData` (size of the section on disk).

Example 6 *The second section in figure 3.5 has the physical starting address* 0x 4 00 *and the physical size* 0x 2 00*. Thus, the physical end of the section is at offset* 0x 6 00*. The same section has its relative virtual starting address at* 0x 20 00*. The absolute address in memory is obtained by adding the base address of* 0x 40 00 00*, the resulting virtual start address is* 0x 40 20 00*. The virtual size of the second section is* 0x 10 00*, therefore, the virtual end address is* 0x 40 30 00*. The virtual size of this section is* 0x 8 00 *bytes larger than its physical size, this overhead is filled with zeroes.*

3.5 PE Malformations

There is a gap between the PE/COFF specification and the actual behaviour of the Windows loader. Windows has to maintain backward compatibility with obsolete compilers and files (see [VP11, slide 8]) and behaves fault tolerant while facing invalid structures or fields. That is why the PE/COFF specification does not reflect the reality. The structure of a PE file according to the PE/COFF specification is hereby defined as the *normal* or *intended* structure.

Vuksan and Pericin define *file format malformations* as 'special case conditions that are introduced to the file layout and specific fields in order to achieve undesired behavior by the programs that are parsing it' [Rev11, p. 4]. However, whether certain special case conditions in files are accidental or intended usually cannot be determined. So the author decided to leave out the intention of the person, who introduced the malformation, for definition 34.

Definition 34 (PE malformation) *A* PE malformation *is data or layout of a PE file that violates conventions or the PE/COFF specification.*

Accidental malformations occur, e. g., if the malware writer does not know the PE/COFF specification well enough to perform file modifications in compliance with it. It might also be more convenient to modify files without adjusting all values that the PE/COFF specification actually requires. An example is a virus that enlarges the last section of the host file and copies itself into it. The adjusted section size might violate alignment restrictions by the PE/COFF specification due to bugs in the code or out of convenience. Some malformations are also done to hide information in a PE file, e. g., marking a host file as infected in a reserved field to prevent a virus from infecting the host twice.

Table 3.2: Field Malformations

Field Malformation	Examples
non-zero fields or flags that should be zero	usually reserved or deprecated fields, e. g. `Win32VersionValue`
mandatory fields or flags that are not set	zero `ImageBase`, zero `VirtualSize` for a section
fields that violate maximum or minimum value restrictions	`FileAlignment` lower than 512
fields that violate alignment restrictions	unaligned section sizes
fields or flags that violate other constraints	`FileAlignment` that is not a power of two
contradictory characteristics	`IMAGE_FILE_DLL` not set for a DLL, `IMAGE_FILE_32BIT_MACHINE` set for a PE32+
addresses or sizes that are too large or too small for the given file	section size too large, virtual entry point

Whether it is intentional or not, violation of the format specification is unexpected and potentially breaks or deceives parsers in any of these cases.

Sheehan et al state that 68 per cent of all image files have malformations (see [SHRS07, slide 7]). Because *PortEx* specializes in PE malware, one goal of *PortEx* is to parse malformed PE files correctly and to recognise malformations.

3.5.1 Field Malformations

Definition 35 (field malformation) *A* field malformation *is a field in the PE File Header or in a special section that has an invalid value according to the PE/COFF specification, or a value that is treated differently by the Windows loader than the PE/COFF specification suggests.*

Field malformations and examples are listed in Table 3.2. The following sections describe some examples for field malformations.

Too Large SizeOfRawData

The PE/COFF specification specifies the `SizeOfRawData` field in the section headers as the size of the section on disk rounded up to the file alignment (see

[Mic13, p. 25], [Pie02a]). According to Albertini [Alb13] the loader replaces the `SizeOfRawData` with the `VirtualSize` if the `SizeOfRawData` is larger than the `VirtualSize`. That means setting the `SizeOfRawData` to a larger value than the `VirtualSize` has the potential to confuse analysis tools. Some tools are not able to determine the physical section size correctly, and break if reading the section based on `SizeOfRawData` exceeds the file size. One example is *pype32*[1].

Zero Entry Point

The `AddressOfEntryPoint` in the Optional Header determines the RVA for the entry point of a PE file. The PE/COFF specification states that the entry point is optional for DLLs [Mic13, p. 17]. It doesn't say anything about EXE files, but since EXE files are applications, which run on their own, the entry point is necessary and conventional to define the start of execution.

If the `AddressOfEntryPoint` of an EXE file is zero, the execution of the file will start at the image base, executing the MS-DOS signature 'MZ' as '`dec ebp/pop edx`' (see [Alb13]). Parsers might classify an EXE with zero entry point as corrupt.

Zero or Too Large Image Base

The `ImageBase` field defines the preferred loading address. Windows XP allows the field to be zero and locates the file to 0x 1 00 00 (see [Alb13]). If the sum of `ImageBase` and `SizeOfImage` is larger than or equal to 0x 80 00 00 00, the file is also rebased to 0x 1 00 00 (see [Alb13]).

Both behaviours are not described in the PE/COFF specification. Emulators might declare a file as corrupt and refuse to load it if its image base is zero or above 0x 80 00 00 00.

Trailing Dots in Imports

Windows XP ignores trailing dots in DLL names in the import section (see [Alb12, slide 76]).

Example 7 *If the name* 'kernel32.dll. . . ' *is given as DLL name in the Import Section, the name* 'kernel32.dll' *is used instead by the Windows loader.*

Detection heuristics, which use, among others, the import names to determine suspicious patterns, may fail to recognise imports with trailing dots.

[1] `https://github.com/crackinglandia/pype32` (last access Wednesday 22nd October, 2014)

Win32VersionValue

Windows operating systems use a datastructure to hold internal process information. This data structure is called process environment block (PEB)[2].

The `Win32VersionValue` is a field in the Optional Header. The PE/COFF specification declares it as reserved and states it 'must be zero' [Mic13, p. 19].

If the field is set anyway, the loader overwrites version information in the PEB after loading the PE file (see [Alb12, slide 82]). Malware writers use this behaviour to break emulators that rely on the version information given in the PE File Header.

ImageFileDLL

The file characterics, which are defined in the COFF File Header, have a flag called `IMAGE_FILE_DLL`. The PE/COFF specification describes this flag as follows:

'The image file is a dynamic-link library (DLL). Such files are considered executable files for almost all purposes, although they cannot be directly run.' [Mic13, p. 15]

In contrast to the PE/COFF specification the `IMAGE_FILE_DLL` flag is not necessary for a DLL to work (see [Alb12, p. 85] and [Alb13]). The DLL will still be able to export symbols if the flag is not set. Tools that rely on the flag to determine whether a file is a DLL or EXE will handle the file as EXE. Subsequently they might fail to work because the entry point might be invalid for an EXE.

3.5.2 Structural Malformations

Definition 36 (structural malformation) Structural malformations *are PE structures—i. e. headers, sections, data structures of special sections, and tables—at unusual locations, with unusual odering, recursive calls; or PE structures that occur in an unusual amount.*

Table 3.3 lists possible structural malformations of a PE file and their examples. Some examples are explained in the following sections.

Too Many Sections

According to the PE/COFF specification the number of sections is limited to 96 [Mic13, p. 12]. While Windows XP refuses to execute a PE with more sections,

[2] `http://msdn.microsoft.com/en-us/library/windows/desktop/aa813706(v=vs.85).aspx` (last access Monday 20th October, 2014)

Table 3.3: Structural Malformations

Structural Malformation	Examples
structures at unusual locations	Section Table or PE File Header in overlay
unusual ordering of structures	shuffled sections, Section Table after the sections
truncated structures	Section Table truncated by the end of the file
unusual number of structures	no sections, no Data Directory, too many imports
fractionated structures	fractionated imports
duplicated structures	dual headers, duplicated sections
collapsed structures	collapsed MS-DOS Header, collapsed Optional Header
structural loops	resource loop
dummy structures	dummy import entry

Windows Vista, 7, and 8 run it regardlessly (see [Lis10, slide 11] and [Alb13]). The number of sections is defined in the COFF File Header as a 16-bit value. So the maximum number of sections is 0x FF FF (65 535) sections. Some tools fail to allocate enough memory upon loading that many sections and crash subsequently. An example is IDA v5.3[3].

Fractionated Data

The PE/COFF specification gives the impression that the structures that belong to one special section are always entirely contained in one PE section as defined by the Section Table. That is because they are labeled *special sections* [Mic13, p. 65] and always called by their conventional section names as given in the Section Table, e. g., the Import Directory Table and all the structures the table points to, are referred to as *.idata Section* (see [Mic13, p. 77]).

The author found malicious PE files that spread the special section contents over several PE sections, e. g., a sample of the file infecting virus W32.Sality[4] places two imports in a different section than the other imports. Another example is illustrated in Figure 3.6, where imports and resource structures are fractionated.

[3] `https://www.hex-rays.com/products/ida/` (last access Wednesday 12th November, 2014)

[4] malware sample #05e261d74d06dd8d35583614def3f22e

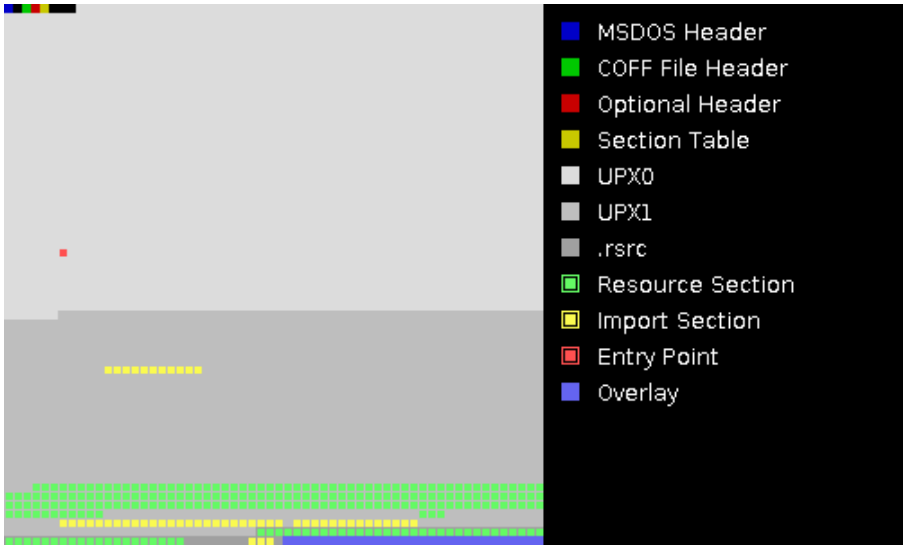

Figure 3.6: Fractionated imports (yellow) and resources (green), output by PortEx, malware sample #7dfcbb865a4a5637efd97a2d021eb4b3

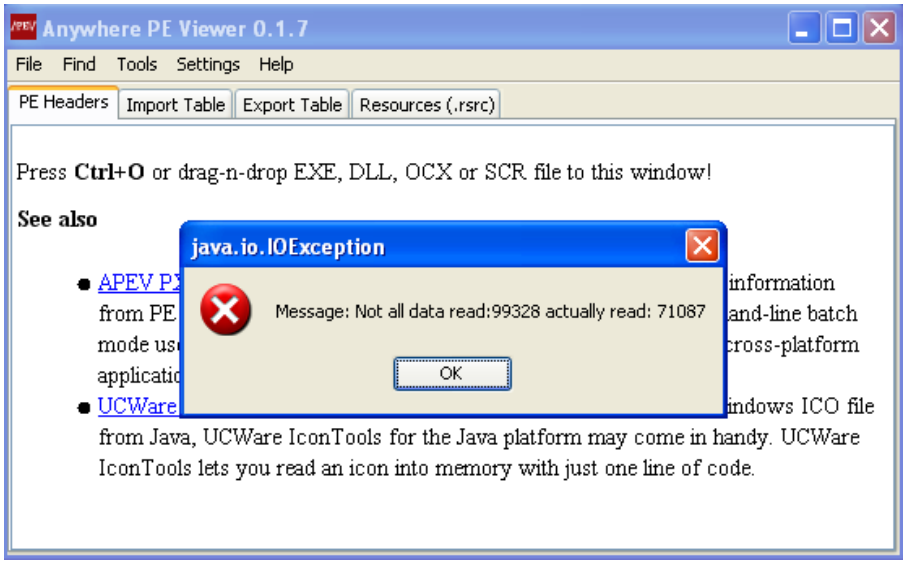

Figure 3.7: Anywhere PE Viewer fails to parse PE files with fractionated data.

Static PE parsers are not able to read fractionated data properly if they load only one section to parse a special section or fail to calculate the correct file offset for the fractionated parts. An example is *Anywhere PE Viewer 0.1.7* as demonstrated in figure 3.7.

Although malware authors may intentionally use this malformation to hide data and evade static heuristic analysis, they may also introduce the malformation unwittingly. A virus like W32.Sality might just add its own imports right in the section where it also places its own body out of convenience.

Writeable PE File Header

The PE File Header usually has only *execute* and *read* attributes enabled. There are two possibilities to make it writeable.

The first possibility is setting the file into *low-alignment mode* (see [Rev11, p. 6]). The purpose of low-alignment mode is to reduce the memory footprint of drivers. It causes the physical addresses to match their virtual counterparts and no padding is applied in memory, thus, the memory-mapped file will be smaller. Low-alignment mode is triggered by setting the `FileAlignment` and the `SectionAlignment` to the same value, which must be greater than zero and smaller or equal to 0x 2 00 (see [Rev11, p. 6]). Low-alignment mode makes the PE File Header writeable (see [Rev11, p. 6]) and forces the whole file to be loaded into memory.

The second possibility is placing the PE File Header in a section and enabling the *write* attribute in the section header (see [Rev11, p. 6]). The `PointerToRawData` in a section header defines the physical start of a section. If this field is set to zero, the section will be invalid (see [Alb13]). But if the `PointerToRawData` is non-zero and smaller than 0x 2 00, which is the standard value for the file alignment, the loader will round down the physical section start to zero (see [Alb13]). This behaviour makes it possible to put the PE File Header into a section.

This malformation is the basis for some other malformations, e. g., a sectionless PE file.

No Sections

Executable code of a PE file is usually placed in one or several sections. A sectionless PE file is able to execute code by placing instructions within the PE File Header. This is demonstrated by Sotirov in his *Tiny PE* project (see [Sot]).

To make this work, the file needs to be in low-alignment mode as described in *Writeable PE File Header*. More detailed instructions are in [Rev11, p. 12, 13].

A PE file without sections may be partly unreadable by reverse engineering tools or break tools that expect the file to have at least one section. One example is

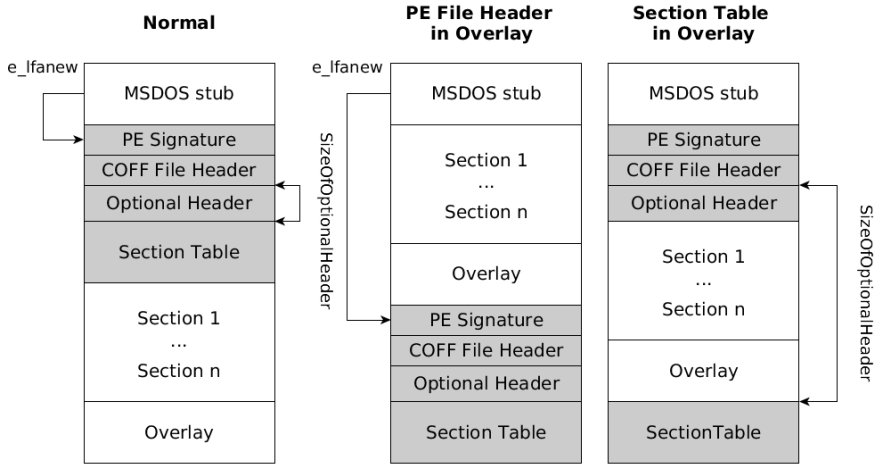

Figure 3.8: The PE File Header (grey) and the Section Table can be placed in overlay (referring to [VP11, slides 13, 14])

the hex editor *Hiew v8.03*[5], which does not recognise a sectionless file as PE file. The commercial tool *PE Explorer v1.99*[6] refuses to open a sectionless file.

PE File Header in Overlay

After the MS-DOS Stub usually follows the PE File Header. The first component of the PE File Header is the PE signature. As explained in section 3.2 the address to the beginning of the PE signature is located in the `e_lfanew` field within the MS-DOS Stub. The address marks the beginning of the PE File Header. It is a 32-bit value. Given that the file is small enough, the `e_lfanew` value can be changed to point to the overlay of the file as illustrated in Figure 3.8 (cf. [VP11, slide 13]). As a result the PE File Header will be read from overlay.

The overlay is never mapped into memory, so the PE File Header will not be present in memory if it resides in overlay. The Windows loader reads most fields of the PE File Header from disk and executes the file anyway. Tools that read the PE File Header from memory will not be able to find it (see [VP11, slide 13]).

Section Table in Overlay

The previous malformation is modified by moving only the Section Table to overlay. The Optional Header has a variable size. The offset from the beginning of the Optional Header and its size determine the beginning of the Section Table. The size of the Optional Header is defined in the field `SizeOfOptionalHeaders`

[5] `http://www.hiew.ru/` (last access Wednesday 12th November, 2014)
[6] `http://www.heaventools.com/overview.htm` (last access Wednesday 12th November, 2014)

of the COFF File Header. It is a 16-bit value, so the maximum value for the size is 65 535 bytes. If the end of the file is smaller than the offset of the Optional Header plus its size, the Section Table can be moved to the very end of the file. The rightmost picture in figure 3.8 illustrates the malformation. This malformation is described in [VP11, slide 14].

As a result of this modification the Section Table will not be mapped to memory. A tool that parses the memory content will not be able to find the Section Table and might classify the file as corrupt. Pericin demonstrates this in his talk at the BlackHat Conference with the debugger OllyDbg (see [VP11, min. 14:45, slide 14]).

Resource Loop

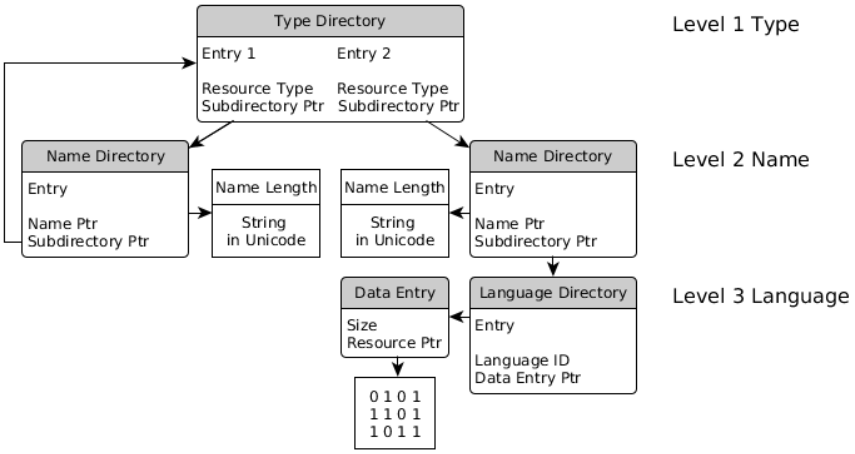

Figure 3.9: Resource tree with a loop

If a node of the resource tree has a back-pointing child-reference, the tree will contain a loop as illustrated in Figure 3.9. PE parsers run into an endless loop and eventually run out of memory if they do not have a limit on the parsed nodes, depth of the tree, or loop detection. PE parsers might also declare the resource section as corrupt and fail to show any resources. An example is *Resource Hacker* in version 3.6.0.92, which fails to read any resources of a PE file with a resource loop.

Collapsed Optional Header

The `SizeOfOptionalHeader` cannot only be enlarged as done for the previous malformation, but also be shrinked, e. g., to four bytes. A PE file example with this malformation is tinype [Sot]. Windows reads the contents of the Optional Header beyond the given size, which is the reason that such a file is still able to run properly.

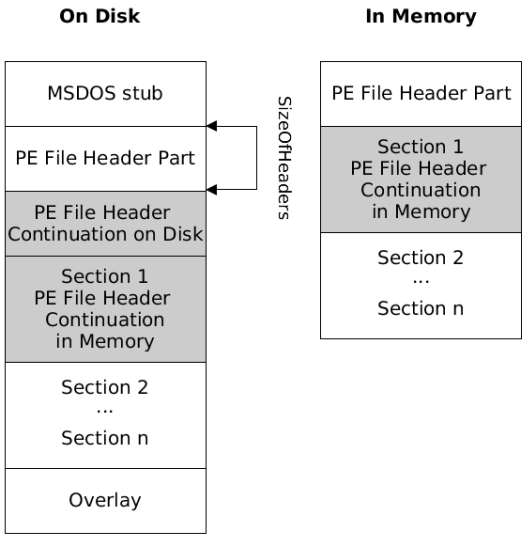

Figure 3.10: Dual PE File Header (referring to [VP11, slides 13, 14])

As a result of the shrinked size the Section Table overlaps with the Optional Header because the `SizeOfOptionalHeader` determines the relative offset of the Section Table (see *Section Table in Overlay*).

Some tools do not parse the contents of the Optional Header beyond the given size, including the data directory. Consequently they are not able to parse imports, exports or other special sections that might be part of the file.

Dual PE File Header

The `SizeOfHeaders` field in the Optional Header defines the 'combined size of an MS-DOS Stub, PE File Header, and section headers' [Mic13, p. 19]. The PE/COFF specification withholds that the `SizeOfHeaders` also determines the VA of the first section implicitly (see [VP11, slide 15]). The first section is located right after the PE File Header in memory based on the `SizeOfHeaders` fields.

If the `SizeOfHeaders` value is reduced, only a part of the original PE File Header will be loaded, and the contents of the first section make up the remaining PE File Header in memory (see [Rev11, p. 5]). That means there are two different PE File Header continuations: The PE File Header on disk is read during the loading process, whereas the PE File Header in memory is read by the loader afterwards upon request (see [Rev11, p. 5]).

One part of the PE File Header is the Data Directory that defines where imports, exports, resources, and other special section contents are located (see section 3.3). These contents are read after the file was loaded into memory. That means the

header continuation in the first section is relevant for loading these contents. Reverse engineering tools that read the contents on disk, will show other imports, exports, or resources than the ones that are actually used while the file is run. Figure 3.10 illustrates the Dual PE File Header malformation. The malformation is described in [Rev11, p. 5].

3.6 Summary

There is a gap between the PE format that the PE/COFF specification describes and the PE files that are actually allowed to run. The PE/COFF specification uses misleading field names and descriptions (e. g., 'SizeOfOptionalHeader' and 'special section'), is more restrictive than the loader, and does not state how corner cases are handled and which corrections the loader performs (see *Too Large SizeOfRawData*, page 44). Furthermore, the behaviour of the loader varies in different Windows versions (see *Too Many Sections*, page 46). Every new version of Windows possibly introduces formerly unknown malformations. This gap and the complexity of the PE format make it considerably hard for developers of PE format parsing tools to handle corner cases properly. The result are tools and antivirus products that are vulnerable to PE malformations.

The author attempts to implement a PE analysis library that is robust against known malformations. The following chapter introduces the requirements, technical details, design, and implementation details of the library *PortEx*. This includes malformation detection and robustness—the solution for the problems that the malformations create.

Chapter 4

Static Analysis Library

This chapter defines the target audience and requirements for the static analysis library *PortEx*, and describes the technology and design decisions based on these requirements. The last section explains the implementation and functionality of *PortEx*' features.

4.1 Target Audience and Requirements

The target audience of *PortEx* includes the following groups:

1. developers of reverse engineering and malware analysis tools
2. malware analysts

Software developers can use the library to build robust tools for malware analysts and reverse engineers. Malware analysts can use the library to automate tasks that are not yet covered by available tools.

The requirements of *PortEx* are based on the goals described in section 1.3. The following enumeration is ordered by importance, starting with the most important requirement:

1. PE format parsing
2. PE malformation robustness
3. PE format anomaly detection
4. maximised potential target audience
5. ease of use

6. platform independence
7. backward compatibility of future versions
8. PE file visualisation
9. overlay detection
10. entropy calculation
11. hash value calculation
12. string extraction
13. packer detection and identification
14. recognition of embedded files

PE format parsing is the main feature of *PortEx*; malformation robustness and anomaly detection are the main motivation for its implementation. Both are part of section 4.4.

An *easy-to-use* API and *platform independence* will attract more people to deploy *PortEx* for tool development. Both are actually subgoals of *maximised potential target audience*. Platform independence is also important because malware analysts often use a different operating system for the host system than for the VM as it makes an infection of the host system more unlikely (see subsection 2.5.3). Static malware analysis can be performed without a VM because the malware is not executed; as such it is more convenient to have platform independent tools for static analysis. *Backward compatibility of future versions* makes transition to an updated version of *PortEx* easier. The sections 4.2 and 4.3 describe how these requirements are met.

The remaining requirements are features for further investigation of PE files. Their implementation is covered in section 4.4.

4.2 Technologies

This section lists and justifies the technologies that are used for the implementation of *PortEx*. This includes the involved programming languages, project management tools, and technologies that help to avoid bugs and maintain high code quality.

4.2.1 Programming Languages and Target Platform

Two requirements for *PortEx* are platform independence and maximisation of the potential target audience (see section 4.1). That means the target platform

should be independent from the operating system, and the more popular and widely used the target language is, the more people will probably use the library.

The *TIOBE Index* is 'an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors'[1]. The ratings from 2002 to 2014 show Java and C variantly having the first and second highest rating (see figure 4.1).

PYPL—the PopularitY of Programming Language Index—shows similar results with Java being the most popular language from 2004 to 2014. PYPL 'is created by analyzing how often language tutorials are searched on Google : the more a specific language tutorial is searched, the more popular the language is assumed to be.'[2]

The author decided to use Java for the implementation of *PortEx* based on the popularity ratings and the platform independence of the Java Virtual Machine (JVM), which is the default runtime environment for Java. So both goals are met.

At the time of writing, the author was not able to find any actively maintained Java library for PE file analysis. Libraries like *PECOFF4J* and *jpexe* have not been updated for the last four years or longer. The commercial library *Anywhere PE Viewer* has been discontinued. This makes the choice for developing a Java library even more relevant because there are no alternatives for Java developers.

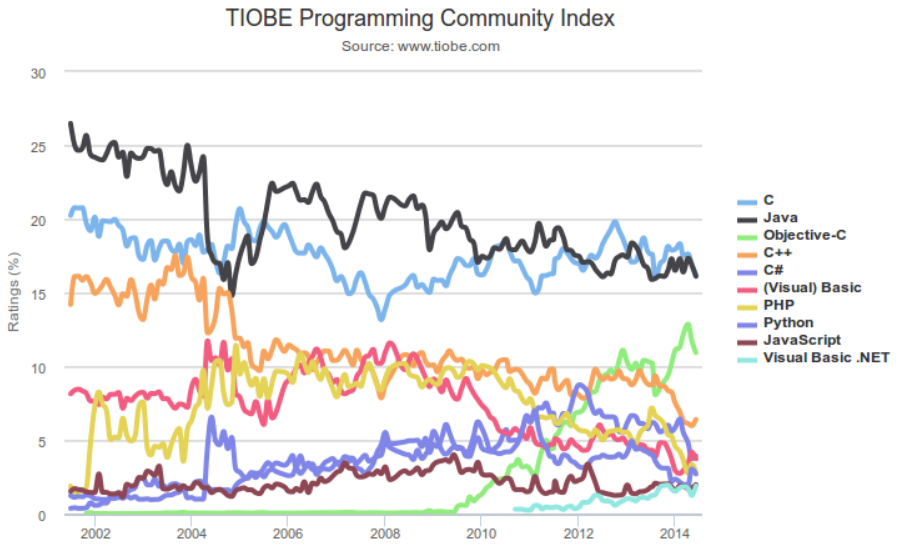

Figure 4.1: TIOBE Index 2002–2014

[1] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` (last access Thursday 23rd October, 2014)

[2] `https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language` (last access Thursday 23rd October, 2014)

PortEx uses Scala in addition to Java. Scala is a functional-imperative hybrid. It compiles to Java bytecode and was designed for seamless interoperability with Java[3]. Scala simplifies programming of tasks in *PortEx* that are better suited to be solved by functional programming. *PortEx* users need at least Java 1.7.

4.2.2 Build Tools and Code Quality Checks

During development, *PortEx* and its HTML documentation are build with Simple Build Tool (SBT)[4], which is a build tool for Scala projects.

Maven is a project management software and build tool for Java projects. Since the target programming language is Java, as defined in the previous section, *PortEx* provides support for Maven[5] (see '*README.md*'[6]). This includes the possibility to integrate *PortEx* into Maven projects as well as the option to build *PortEx* itself with Maven instead of SBT.

Metrics, static codestyle and bug analysis, precondition checks, and unit testing ensure the quality of *PortEx*' code. *Findbugs*[7] performs static code analysis for bad practices and actual bugs in the code. It analyses Java bytecode of any version from 1.0 to 1.8. The Eclipse *Checkstyle*[8] plugin is a static analysis tool that checks the code for compliance with a self-set coding standard. *PortEx* has *TestNG* unit tests for all public classes and methods. The *EclEmma*[9] plugin for Eclipse checks the code coverage.

4.3 API Design

The API design influences two goals of *PortEx* (see *Target Audience and Requirements*, section 4.1):

- ease of use
- backward compatibility of future versions

A *good API design* is hereby defined as a design that meets the two aforementioned goals. Bloch, who designed features of the Java language, defines principles for good API design (see [Blo09]).

This section gives an overview on the main structure of *PortEx*. It lays the foundation of understanding the details about *PortEx*' API design decisions. The

[3] `http://www.scala-lang.org/` (last access Wednesday 12th November, 2014)
[4] `http://www.scala-sbt.org/` (last access Wednesday 12th November, 2014)
[5] `https://maven.apache.org/` (last access Wednesday 12th November, 2014)
[6] `https://github.com/katjahahn/PortEx` (last access Wednesday 12th November, 2014)
[7] `http://findbugs.sourceforge.net/` (last access Wednesday 12th November, 2014)
[8] `http://eclipse-cs.sourceforge.net/` (last access Wednesday 12th November, 2014)
[9] `http://eclemma.org/` (last access Wednesday 12th November, 2014)

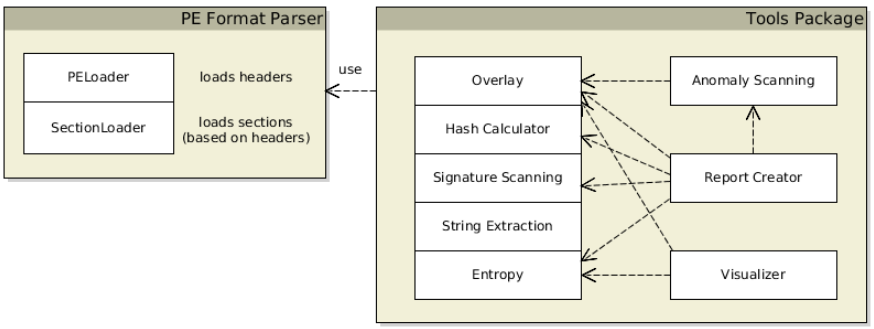

Figure 4.2: Main structure of PortEx and dependencies of the modules

section continues with explanations about Bloch's API design principles, how *PortEx* employs them, and how the design principles are checked—if checking is possible.

4.3.1 Main Structure

PortEx consists of two main parts: The first is the PE format parser; the second is the tools package, whose parts build upon the parser. Figure 4.2 illustrates the two parts and their main modules.

PE Format Parser

The PE format parser is responsible for parsing the contents of a PE file and providing the extracted header and section data to the API user in programmatical form.

The PE format parser has two loader classes: the `PELoader` for the header data—i. e. MSDOS Header, COFF File Header, Optional Header, and Section Table—, and the `SectionLoader` for the sections and special sections. The header data is loaded at once because its size is small enough (large sizes due to malformations are cut down). The sections, however, may be several gigabytes in size, thus, they are only loaded on behalf of the library user.

The `PELoader` collects all header data in a `PEData` object and returns it to the library user. This data object is the basis for loading sections and special sections with the `SectionLoader`. Special sections are, e. g., Import Section, Export Section, and Resource Section.

Example 8 *The following sample codes show how a user loads the header data and sections with* PortEx. *They are basic steps to get any parsed information about a PE file.*

```
1 File file = new File("WinRar.exe");
2 PEData headerData = PELoader.loadPE(file);
```

The header data is necessary to load the sections.

```
SectionLoader loader = new SectionLoader(headerData);
//load import section
ImportSection idata = loader.loadImportSection();
//load general purpose section by section number
final int sectionNumber = 1;
PESection section = loader.loadSection(sectionNumber);
```

Some special sections provide their collected data in two different abstraction levels. These are the Import Section, Export Section, and Resource Section. The lower abstraction level offers access to the underlying data structures, e. g., the addresses of each element in the structure. The higher abstraction level allows the user to get special section information, e. g., imported symbols from the Import Section, without knowledge of the underlying data structure.

Example 9 *This example demonstrates the two abstraction levels for the Resource Section. The Resource Section is build up as a tree. Each path from the root node to a leaf represents one resource (see section 3.3).* PortEx *allows the user to retrieve a list of* `Resource` *objects.* PortEx *traverses the tree and collects the information in these objects. This information includes language, name, and type ID, and the location of the actual resource bytes.*

```
// load the Resource Section
ResourceSection rsrc = new SectionLoader(headerData).loadResourceSection();
// retrieve a list of Resource objects
List<Resource> resources = rsrc.getResources();
// print resource information
for (Resource resource : resources) {
  System.out.println(resource.getInfo());
}
```

A typical output of the previous code is in the following listing. It shows the start address of the resource bytes, language ID, name ID, and the type ID.

```
address: 0x1f4a0, size: 0x2dc, language -> ID: 1033, name -> ID: 1, type -> ID:
    RT_VERSION
address: 0x1f77c, size: 0x15a, language -> ID: 1033, name -> ID: 1, type -> ID:
    RT_MANIFEST
```

On the lower abstraction level more detailed information about the resource tree is accessible by retrieving a resource tree object.

```
// get the root of the resource tree
ResourceDirectory tree = rsrc.getResourceTree();
// access the header information, in this example MAJOR_VERSION
ResourceDirectory tree = rsrc.getResourceTree();
Map<ResourceDirectoryKey, StandardField> header = tree.getHeader();
long majorVersion = header.get(ResourceDirectoryKey.MAJOR_VERSION).getValue();
// get a list of directory entries
List<ResourceDirectoryEntry> entries = tree.getEntries();
```

A more detailed illustration of the architecture of the PE format parser is in figure 4.3.

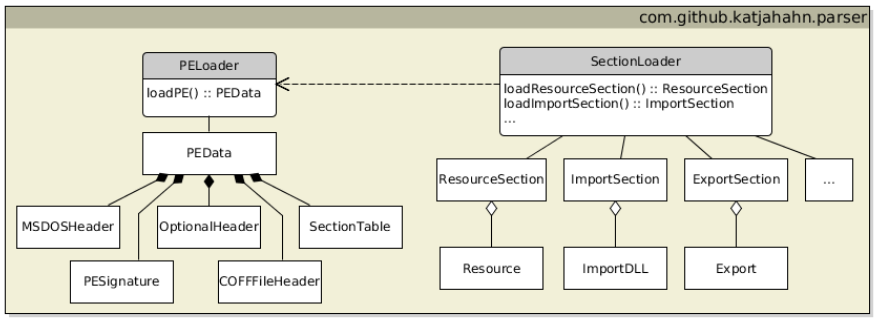

Figure 4.3: Structure of PortEx' PE format parser

Tools Package

The tools package is the second part of *PortEx*. It contains various tools that are useful for malware analysis. These tools rely on the PE format parser to perform their tasks.

The purpose of the tools is extraction of strings, signature scanning, anomaly detection, overlay detection, entropy calculation, hash value calculation of files and sections, PE visualisation, and report creation. Some of these tools are dependend on other tools:

- the `AnomalyScanner` uses overlay detection to determine overlay-related anomalies
- the `ReportCreator` collects section entropies, hashes, anomalies, overlay information, and string scanning results to return formatted reports about a file
- the `PEVisualizer` uses entropy calculation, and overlay detection to visualise PE files

The `ReportCreator` and `PEVisualizer` operate as information collectors and presenters. The library user does not need to know about these dependencies because they do not affect the usage of the tools. All tools are presented in detail in section *Features* starting from subsection 4.4.2.

4.3.2 General Design Principles

General design principles are not specific to classes or methods.

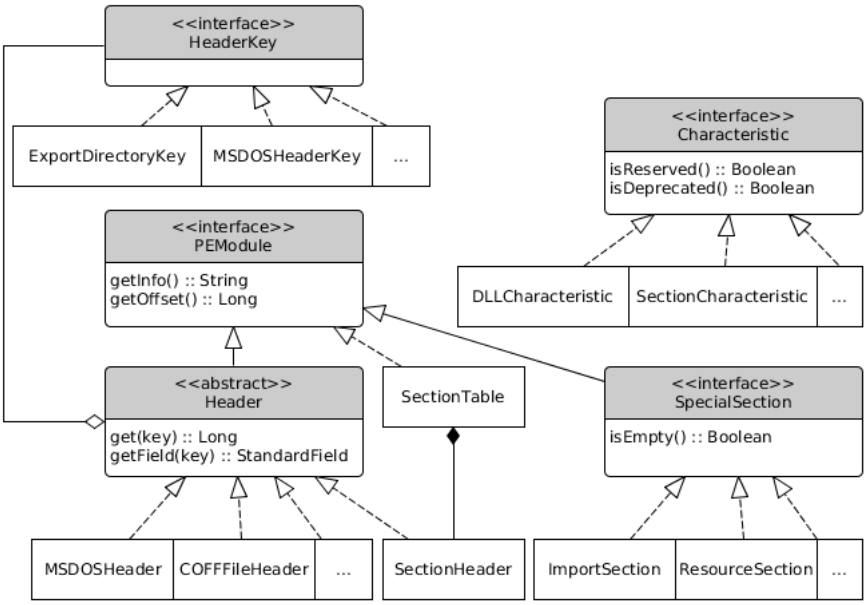

Figure 4.4: PE parser interfaces of PortEx

As Small as Possible

Once a library is in use, elements of its API cannot be removed any more without breaking backwards compatibility. Therefore, any class, method, or parameter should only become part of the public API if necessary. *PortEx* uses the `@Beta` annotation to denote public methods or classes that are subject to change.

Bloch states that 'the conceptual weight [is] more important than bulk' [Blo09, slide 14]. In his talk he explains that the conceptual weight is the number of concepts a person has to learn in order to use the API; and the bulk is the number of methods, classes, and parameters (see [Blo09, slide 14]).

According to Bloch, the most important way to minimise the number of concepts is re-use of interfaces (see [Blo09, slide 14]). That way the user only has to learn how to interact with the interface instead of learning how to interact with every implementation of that interface.

Widely implemented interfaces of the *PortEx* API are illustrated in figure 4.4. Every structure of a PE file is a `PEModule`, and can return a description and an offset. Such structures are sections, special sections, headers, and tables. The `Header` interface covers all headers of a PE File, including PE Headers and Headers in special sections. Each header consists of `HeaderKey` and `StandardField` pairs. `Characteristics` are part of several headers, e. g., the `SectionCharacteristics` are defined by flags in each `SectionHeader`. These five interfaces in figure 4.4 are implemented by 32 classes and provide access to all header data and file offsets.

Minimise Accessibility

Classes and their members have to be 'as private as possible' [Blo09, slide 16]. This adheres to the principle of *information hiding*. It protects parts of the program from modification and exposes only the classes and members that are important for the API user. It also *minimises coupling*, which makes modules indepently understandable, buildable, and testable (see [Blo09, slide 16]). *PortEx* has 484 private members and 373 public members in version 1.0-beta1.1.

Bloch also states that '[p]ublic classes should have no public fields (with the exception of constants)' [Blo09, slide 16]. This requirement is fully met by *PortEx*.

Names Matter

Bloch says the names of methods, variables and classes should be *consistent*, and *self-explanatory*. *Consistency* means that throughout the API the same name should always mean the same thing and the same thing should always have the same name.

Example 10 *It would be a bad idea according to Bloch to have* delete *and* remove *in the API. If* delete *and* remove *are different, it is not clear what the difference is; if both refer to the same, only one name should be used to avoid confusion (see [Blo09, slide 17])*

Self-explanatory code avoids abbreviations and should 'read like prose' [Blo09, slide 17]. *PortEx* uses the exact terms that the PE/COFF specification uses. In addition, the following rules are applied:

- *PortEx* adheres to the naming conventions defined by Sun Microsystems [Sun99].
- Every method that returns an `Optional` starts with 'maybe'.
- Every subclass of `Header` ends with 'Header'.
- Every subclass of `SpecialSection` ends with 'Section'.
- The interfaces `Header`, `PEModule`, and `SpecialSection` ensure the same method names for the same tasks in headers and sections.
- Abbreviations are only used if they are common (e. g. 'PE').
- `newInstance` always indicates a static factory method that creates and returns a new object every time it is called. `getInstance` on the other hand may return the same object on different calls (e. g. singleton pattern).
- The prefix 'load' for method names is used if the file on disk has to be read to perform the action.

Documentation

The documentation of an API is the user's resource to install, learn, and use the API. It includes:

1. information about packages, classes, interfaces, and class members
2. instructions for installation, compilation, building
3. tutorials
4. example codes
5. license information

Without code documentation the user of the API either has to guess the purpose of a class, method, or parameter; or has to read the implementation. In the latter case the implementation will become the specification, thus, must not be changed any more (cf. [Blo09, slide 19]). For this reason it is necessary to document everything. The code documentation of *PortEx* is publicly available[10].

Installation and build instructions are part of the *README.md*[11] and the *PortEx* project page[12].

The GitHub Wiki of *PortEx*[13] provides tutorials and usage examples. There is a general wiki page for the PE File Headers, and there is a wiki page for every special section and tool.

License information is part of the *README.md*, of the *PortEx* project page, and of the *LICENSE*[14] file in the root folder of the project.

4.3.3 Class Design

The following design principles are related to classes and inheritance.

Minimise Mutability

Immutable objects are simple, thread-safe, and reusable, therefore, should be preferred over mutable objects (see [Blo09, slide 24]). If there is a reason to

[10] `http://katjahahn.github.io/PortEx/javadocs/` (last access Thursday 23rd October, 2014)
[11] `https://github.com/katjahahn/PortEx/blob/master/README.md`
[12] `http://katjahahn.github.io/PortEx/` (last access Thursday 23rd October, 2014)
[13] `https://github.com/katjahahn/PortEx/wiki` (last access Thursday 23rd October, 2014)
[14] `https://github.com/katjahahn/PortEx/blob/master/LICENSE` (last access Thursday 23rd October, 2014)

use mutable objects, their state-space must be kept small and well-defined (see [Blo09, slide 24]).

Scala provides and encourages the use of immutable collections and constants. Mutable Scala collections are only used locally and not passed or returned. As the target audience are Java programmers, the Scala collections are not used as return type for the API methods. They are converted into Java collections instead and returned as such.

Java has no built-in keyword to force immutability. It only has the `final` modifier for variables, which makes it impossible to assign a new instance to the variable. However, the contents of final objects or arrays can still be changed. *PortEx* employs the following strategies to minimise mutability in the Java part of the code:

- *PortEx* uses immutable collections by Google Guava.
- *PortEx* only passes or returns copies of mutable collections and objects. This avoids the problem that the caller might be able to change the state of the class by modifying the collection or object. This is verified by Findbugs.
- *PortEx* uses no public fields with the exception of constants. This is verified by Checkstyle.

Subclasses Only Where it Makes Sense

Every public subclass must have an is-a relationship to its superclass. Public classes that inherit other classes just for the ease of implementation are prohibited (see [Blo09, slide 25]).

Example 11 *A bad example according to Bloch is* 'Properties extends Hashtable' *(see [Blo09, slide 25]) because semantically it is not the case that every Properties object is a Hashtable. A good example for subclassing is* 'Set extends Collection' *(see [Blo09, slide 25]).*

The public classes in *PortEx* adhere to this principle. The following subclasses are part of the public API of *PortEx* and all of them have an is-a relationship:

- `FileFormatException` extends `IOException`
- `VirtualLocation`, and `PhysicalLocation` extend `Location`
- `COFFFileHeader`, `OptionalHeader`, `MSDOSHeader`, and `SectionHeader` extend `Header`
- `SpecialSection` extends `PEModule`

4.3.4 Method Design

Method design refers to design principles that are related to method definitions, method parameters and semantics.

Reduce Boilerplate Code for the Client

Boilerplate code is code that is found in several places of a program with no or slight variations. An API can force a user to write boilerplate code if it leaves actions to the client that the library should actually do. Boilerplate code is error-prone, hard to read, and annoying (cf. [Blo09, slide 28]).

Basic functions of *PortEx* are listed in the requirements section. They include PE file parsing and getting information from the modules of the tools package. The following examples demonstrate that the basic functions of *PortEx* are accessible with two to three lines of code, of which no step is superfluous. More code samples are in section 4.4.

Example 12 *The following code uses the report creator to print all available information about a PE file to standard output.*

```
1 File file = new File("sample.exe");
2 ReportCreator.newInstance(file).printReport();
```

The *PortEx* tools take a file object and call the parser themselves, thus, they do not force the client to load the header and section data. Every tool has predefined settings if possible, so they are operable with a minimum of parameters. One example is the PE visualiser, which has seven settings that affect the appearance of the image, but can operate after passing the file object as minimum requirement.

Example 13 *The following listing shows the minimum code to create a visualisation of a PE file with* PortEx.

```
1 File file = new File("sample.exe");
2 Visualizer visualizer = new VisualizerBuilder().build();
3 BufferedImage image = visualizer.createImage(file);
```

Access to all Data Available in String Form

Every information that is available as description string must also be available in programmatic form (see [Blo09, slide 31]). The API user must not be forced to parse strings to get information. This is not only cumbersome for the user, the string format will also become part of the API and cannot be modified without potentially breaking the clients' code.

PortEx ensures the programmatic accessibility for all data in string form. `Anomaly` instances in *PortEx*, e. g., have not only a description string, but

also a type, subtype, and the structure or field they are related to. *PortEx* users might decide to put their own description for the anomaly and they are able to do so without parsing the predefined description string. See also example 14, which shows the same principle for the imports of a file.

Example 14 *The imports of a PE file are be printed with the following code.*

```
1 PEData data = PELoader.loadPE(file);
2 ReportCreator reporter = new ReportCreator(data);
3 System.out.println(reporter.importsReport());
```

The `ReportCreator` *constructs and returns a string based on the import information. This string is printed to standard output. An excerpt of the resulting output looks like this:*

```
1 ADVAPI32.DLL
2 rva: 90292 (0x160b4), name: RegCloseKey, hint: 0
3 rva: 90296 (0x160b8), name: RegCreateKeyExA, hint: 0
4 rva: 90300 (0x160bc), name: RegOpenKeyExA, hint: 0
5 rva: 90304 (0x160c0), name: RegQueryValueExA, hint: 0
6 rva: 90308 (0x160c4), name: RegSetValueExA, hint: 0
```

The following code reconstructs the import description string above by accessing every unit of data programmatically:

```
1  PEData data = PELoader.loadPE(file);
2  // Loading the import section
3  SectionLoader loader = new SectionLoader(data);
4  ImportSection idata = loader.loadImportSection();
5  // List of imports
6  List<ImportDLL> imports = idata.getImports();
7  for (ImportDLL dll : imports) {
8      System.out.println(dll.getName());
9      for (NameImport nameImport : dll.getNameImports()) {
10         System.out.print("rva: " + nameImport.getRVA());
11         System.out.print(", name: " + nameImport.getName());
12         System.out.println(", hint: " + nameImport.getHint());
13     }
14     for (OrdinalImport ordImport : dll.getOrdinalImports()) {
15         System.out.println("ordinal: " + ordImport.getOrdinal());
16     }
17     System.out.println();
18 }
```

Avoid Long Parameter Lists

Bloch suggests not to have more than three parameters in a parameter list (see [Blo09, slide 35]). Otherwise the usage of the API becomes difficult and error-prone (see [Blo09, slide 35]). If several parameters of the same type are involved and the API user confounds the arguments, the program will still compile. The argument transposition will only be noticeable at runtime.

Long parameter lists are avoided by breaking up the method or introducing helper classes that hold parameters. A prominent example for a helper class in *PortEx* is the `PEData` class, which holds all headers of a PE file. Another example is

the builder pattern, which prevents long parameter lists in constructors. *PortEx* uses the builder pattern for the visualiser (see subsection 4.4.6).

Long parameter lists are not part of the public interface of *PortEx*. Checkstyle enforces a maximum of three parameters.

Avoid Return Values That Demand Exceptional Processing

Return values that demand exceptional processing must be avoided. An example for a problematic return value is the null reference. It has no actual type, its semantics are not clear, the client is not forced to check for `null`, which might result in a `NullPointerException`, whereas null-checks make the code cumbersome. Another example for possibly problematic return values are magic numbers, e. g., a -1 to indicate a missing value.

Example 15 *The following code returns the header of the resource tree:*

```
1 header = loader.loadResourceSection().getResourceTree().getHeader();
```

If any of the methods above returns a null reference, the code will throw a `NullPointerException` at runtime. The only way to avoid this from the client's perspective is to check for null references as in the following listing.

```
1 ResourceSection rsrc = loader.loadResourceSection();
2 if(rsrc != null) {
3     ResourceDirectory tree = rsrc.getResourceTree();
4     if(tree != null) {
5         header = tree.getHeader();
6     }
7 }
```

PortEx *avoids `null` as return value, so this chain of null-checking if-statements is not necessary. The methods in this example throw exceptions for wrong usage and return empty objects for missing data.*

The following list provides alternatives to exceptional return values:

- Methods with potentially missing return values may use the `Optional` class by Java 8 or Google Guava. The client is forced to check for the missing value, making the code less error-prone. The method's type already indicates the possibility of a missing value.
- The method may return an empty object. An empty object is, e. g., an empty string or an empty list. It may also be a special subclass that indicates emptyness, e. g., a `NullEntry` to indicate the last lookup table entry in the Import Section. Methods of empty objects can still be called without provoking a `NullPointerException`.
- Methods can throw an exception if something went wrong. The client's code will fail as soon as the error occurs and the client has the chance to handle the exception where it is appropriate. Subsequent operations that require the missing value are not performed if an exception is thrown.

No public method of *PortEx* returns `null`. This is verified by FindBugs. *PortEx* makes use of exceptions, empty objects, and Google Guava's `Optional` if a return value might be missing (*PortEx* uses Java 7, thus, Java 8 Optional is not available). Java assertions check for the absence of returned `null` values during development and testing. In version 1.0-beta1.1 there are 77 assertions in *PortEx*, 17 public methods return `Optional`, no methods return `null`.

4.4 Features

The requirements of *PortEx* that have not been addressed so far are the ones related to features of the API:

- PE format parsing
- PE malformation robustness
- PE format anomaly detection
- PE file visualisation
- overlay detection
- entropy calculation
- hash value calculation
- string extraction
- packer detection and identification
- recognition of embedded files

This section describes how these features are implemented, mentions the buildup of the underlying classes and their usage.

4.4.1 Robust Parsing

The robustness of a PE format parser is measured by the ability to parse malformed PE files *correctly*. A file is parsed *correctly* if the information extracted by the parser matches the information that the operating system uses upon execution of the file. That means if there is, e. g., a duplicated PE File Header, the header information that the Windows loader will use is the correct one.

PortEx is a static analysis tool, thus, it parses the file's content from disk. Some malformations are targeted at tools that parse the file from memory, e. g., a Section Table in overlay will not be present in memory. These malformations are not taken into account because they do not affect static analysis tools.

Malformations that possibly distort static parsers are relevant for the robustness of *PortEx*.

This section explains how the PE parser of *PortEx* deals with malformations. The term *naïve parser* hereby refers to a PE parser that is vulnerable to all static-parser-affecting malformations.

Physical Section Range

One important task of a PE format parser is to calculate the actual physical start and physical size of sections as they are read on disk. These values differ from the start and the size of sections in memory (i. e. virtual address and virtual size). The physical size of a section is herewith called *readsize*.

Several malformations are related to the *readsize* and physical start of sections and cause malfunctions in tools with wrong *readsize* calculation. These malformations are:

- zero `VirtualSize`, zero `SizeOfRawData`
- violation of file alignment constraints for `SizeOfRawData`, `VirtualSize`, or `PointerToRawData`
- the physical end of the section is outside the file
- non-default `FileAlignment` used

Listing 4.1 provides an algorithm to calculate the *readsize* of a section. The listing is in pseudocode.

Listing 4.1: Calculating the physical size of a section (based on [15])

```
1  // calculates and returns the readsize of a section
2  method getReadSize():
3    alignedPointerToRawData = rounded down PointerToRawData to multiple of 512
4    alignedVirtualSize = rounded up VirtualSize to multiple of 4 KB
5    alignedSizeOfRawData = rounded up SizeOfRawData to multiple of 4 KB
6    readsize = fileAligned(PointerToRawData + SizeOfRawData)
7        - alignedPointerToRawData
8    readsize = min(readsize, alignedSizeOfRawData)
9    if VirtualSize != 0 then
10      readsize = min(readsize, alignedVirtualSize)
11   // section end outside the file
12   if readsize + alignedPointerToRawData > filesize then
13      readsize = filesize - alignedPointerToRawData
14   // section start outside the file --> nothing is read
15   if alignedPointerToRawData > filesize then
16      readsize = 0
17   return readsize
18
19 //rounds up to multiple of FileAlignment
20 method fileAligned(value):
21   rest = value % FileAlignment
22   result = value
23   if rest != 0 then
24       result = value - rest + FileAlignment
25   return result
```

The algorithm displays the corrections that are done by the Windows loader if certain fields violate alignment restrictions.

The physical start of the section is the aligned `PointerToRawData`. The value 512 to align the `PointerToRawData` is hardcoded and independent of the actual `FileAlignment` value in the header.

The `VirtualSize` and `SizeOfRawData` are rounded up to a multiple of 4 KB, which is the default `SectionAlignment` value.

If the `SizeOfRawData` exceeds the `VirtualSize`, only the `VirtualSize` is used to calculate the `readsize` of the section (see 'SizeOfRawData' in [Alb13]).

Simulation of the Loading Process

The location of special sections like the Import Section is defined in the data directory of the Optional Header (see section 3.3). Each entry of the data directory consists of a size field and an address field that defines the virtual start of the special section. The naïve parser reads special sections by loading exactly the bytes given by the data directory. Special sections, whose data is outside this range, cause an error or are only partially parsed. An example is the fractionated data malformation, where connected structures are placed in different sections (see section 3.5.2). Static PE parsers additionally face the problem that they have to convert the in-memory addresses and offsets to physical ones. If two structures are placed in different sections and one structure has a relative offset to the other structure, the in-memory offset will not necessarily match the one on disk. As demonstrated in section 3.4, the sections may have different locations and sizes in memory than they have on disk, and their order might also differ (shuffled sections). Robust parsing has to take these cases into account.

PortEx' solution to this problem is to simulate the behaviour of the Windows loader while the loader maps the PE file into memory. The object that represents the memory mapped PE loads the content on request. Thus, it is possible to map and parse large PE files without causing memory problems. Listing 4.2 provides an algorithm in pseudocode to create the section mappings of a PE file. A mapping in this code is a pair of one physical address range and one virtual address range. All mappings of one file make up the memory-mapped PE file.

Listing 4.2: Algorithm to create section mappings for a PE file (in pseudocode)

```
1 method getSectionMappings():
2   mappings = new List()
3   foreach sectionHeader in sectionTable do
4     if isValidSection(sectionHeader) then
5       readsize = sectionLoader.getReadSize(sectionHeader)
6       // calculate the physical range
7       physStart = sectionHeader.getAlignedPointerToRawData()
8       physEnd = physStart + readsize
9       physRange = new PhysicalRange(physStart, physEnd)
```

[15] `http://reverseengineering.stackexchange.com/questions/4324/reliable-algorithm-to-extract-overlay-of-a-pe` (last access Thursday 23rd October, 2014)

```
      // calculate the virtual counterparts for the physical range
      virtStart = sectionHeader.getAlignedVirtualAddress()
      virtEnd = virtStart + readsize
      virtRange = new VirtualRange(virtStart, virtEnd)
      // add mapping to list
      mappings.add(new Mapping(virtRange, physRange))
  return mappings
```

The aligned `PointerToRawData` is the `PointerToRawData` value rounded down to a multiple of 512. The aligned `VirtualAddress` of a section is the `VirtualAddress` value rounded up to a multiple of 4 KB.

Reading a byte from such a memory-mapped PE file requires to find the mapping that contains byte's virtual address. The mapping translates the virtual address to a physical one. The virtual space is initially filled with zeroes, so if there is no mapping that contains a given virtual address, a zero byte is returned (see listing 4.3). This provides robustness for malformations that place structures or fields (partially) outside the range of the file. The naïve parser crashes if it faces these malformations because it attempts to read after the end of the file.

Listing 4.3: Reading a byte from the simulated memory mapping of a PE file (in pseudocode)

```
class MemoryMappedPE:

  method getByte(virtualAddress):
    foreach mapping in mappings do
      if mapping.virtRange.contains(virtualAddress) then
        // mapping found, return byte
        return mapping.getByte(virtualAddress)
    // there is no mapping
    // return initial value for virtual space
    return 0

class Mapping:

  field virtRange
  field physRange

  method getByte(virtualAddress):
    // relative offset from the start of the virtual range
    relativeOffset = virtualAddress - virtRange.start
    // absolute file offset to start reading from
    fileOffset = physRange.start + relativeOffset
    // read byte from file
    file.readByte(fileOffset)
```

Simulating the mapping process of headers and sections provides robustness for the following malformations:

- fractionated data
- shuffled sections
- shuffled data, e. g., shuffled resource tree nodes
- dual PE File Header; the header is read as it would be read from memory
- virtually overlapping sections

- virtual fields and structures, e. g., a virtual section table
- truncated fields and structures, i. e. fields or structures that are truncated by the end of the file

Resource Loop Detection

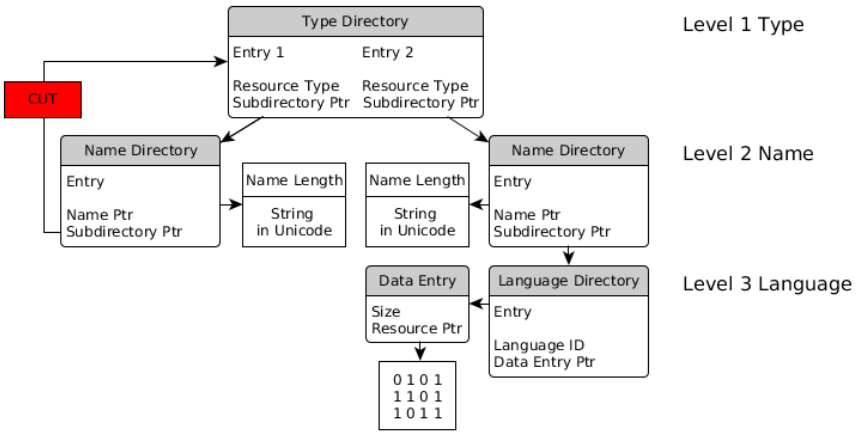

Figure 4.5: A loop in a resource tree is cut by the PortEx parser

PortEx has a loop detection to avoid problems with loops in the resource tree. *PortEx* saves the physical addresses of the nodes that have already been parsed. If the parser recognises that it is about to parse the same node again, it will cut the tree at this point (the cut is done internally, the file is not modified). Figure 4.5 shows were the cut is done in a sample tree.

Dealing With Oversized and Overmuch Structures

The naïve parser reads all bytes of one section, special section, or one resource at once. Malware authors exploit this, e. g., by setting a very large size value in a data directory entry, so that the naïve parser runs out of memory. Such structures are *oversized*.

Malware authors set large values for the number of structures, e. g., relocations, to exhaust the naïve parser while it attempts to read all of them. Such structures are *overmuch*. Related malformations to oversized and overmuch structures are listed below.

- too large size given in a data directory entry
- too long strings, e. g., export names
- too large sections, resources, or `SizeOfOptionalHeader`

- too many sections, imports, exports, relocations, or resources

To deal with overmuch structures *PortEx* has upper limits for the number of imports, exports, relocations, and resources. The section limit by the PE/COFF specification is ignored because the loader of Windows Vista and above also ignores it. *PortEx* is still able to read the maximum number of sections.

PortEx deals with oversized structures by reading only small chunks of bytes as they are requested. *PortEx*' objects for, e. g., resources and sections do not save any bytes, but their address range.

Sizes in the data directory must be ignored, except for the security directory (see [Alb13]). The `NumberOfRvaAndSizes` value denotes the number of data directory entries. It must be rounded down to 16 if it is bigger (see [Alb13]).

The name `SizeOfOptionalHeader` and its definition in the PE/COFF specification (see [Mic13, p. 12]) are misleading because the field does not determine the number of bytes that are necessary to parse the Optional Header. It only defines the start of the Section Table relative to the start of the Optional Header. The naïve parser loads all bytes of the Optional Header based on the `SizeOfOptionalHeader`, as such, it runs out of memory if the value is too large. *PortEx* sets the number of bytes to parse the Optional Header as follows:

```
1 // MAX_SIZE == 240
2 int size = OptionalHeader.MAX_SIZE;
3 // if offset and size exceed the end of the file, cut size
4 if (size + offset > file.length()) {
5     size = (int) (file.length() - offset);
6 }
```

The `MAX_SIZE` is the size in bytes that the Optional Header has for a PE32+ with the maximum of 16 data directory entries.

Dealing With Collapsed and Missing Structures

Collapsed and missing structures include, among others, the following malformations:

- collapsed MSDOS Header, Optional Header, or Import Table
- no Section Table and no sections
- no data directory entries

Collapsed or missing structures cause the naïve parser to ignore information because it is unable to parse these structures or classifies them as corrupt. In case of the collapsed Optional Header the malformation is based on misleading information about the `SizeOfOptionalHeader` by the PE/COFF specification (see section 3.5.2). The loader for .NET PE files ignores the number of data

directory entries as given by the `NumberOfRvaAndSizes`. Thus, the number must also be ignored by PE parsers.

Missing and collapsed structures also cause problems because there is a gap of knowledge about the actual behaviour of the loader. The naïve parser sets artificial limits for parsing PE files based on the PE/COFF specification. The solution is to ignore the restrictions that are imposed by the PE/COFF specification and allow structures to be absent or to overlap each other.

4.4.2 Entropy Calculation

Shannon defines the entropy H of a set of probabilities $p_1, \ldots, p_n$ as follows (see [Sha48, p. 19]):

$$H = -\sum_{i=1}^{n} p_i \log p_i$$

Billouin explains the meaning of Shannon's entropy as 'a measure of the lack of detailed information [...]. The greater is the information, the smaller will be the entropy' [Bri04, p. 193]. That means the entropy of randomly generated, encrypted, or compressed files is higher than of other files. Since malware packers use encryption and compression, the entropy of sections or overlay is an indicator for packer usage and a possible feature for packer heuristics. In an infected file the entropy also indicates the location of an encrypted virus body.

PortEx calculates the entropy of PE sections using the following algorithm (representation in pseudocode).

Listing 4.4: Entropy calculation of a byte sequence

```
1 P = set of probabilities for each byte to occur in the byte stream
2 H = 0.0
3 base = 256
4 foreach p in P do
5   if (p != 0)
6     H = H - p * (log(p) / log(base))
7 return H
```

PortEx calculates P by counting the occurrences of every byte value in the byte stream. The relative frequency of the bytes is used as probability estimate.

The base of the logarithm in Shannon's formula is here the number of possible values a byte can have, which is 256. This way the resulting entropy H is a number in the interval $I = [0, 1]$. Excluding the case $(p = 0)$ is necessary because $(\log 0)$ is undefined.

Some analysis software like *VirusTotal* use base 2 for the logarithm instead of base 256. The resulting entropy is in the interval $J = [0, 8]$ instead of $I = [0, 1]$.

Figure 4.6 illustrates the structure and dependencies of *PortEx*' entropy tool. The `ShannonEntropy` class calculates the entropy of byte arrays, file entropy,

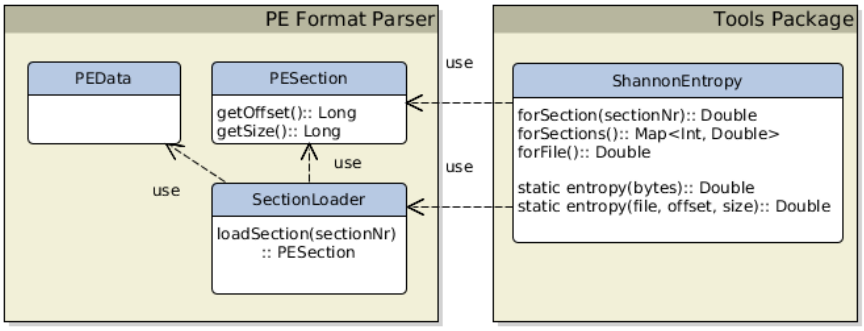

Figure 4.6: Entropy tool and dependencies

and section entropies. It uses the parser package to obtain the physical location of the sections.

Example 16 *A typical usage of the tool is shown in the following listing. The code prints the entropy for every section.*

```
PEData data = PELoader.loadPE(file);
int nrOfSections = data.getCOFFFileHeader().getNumberOfSections();
ShannonEntropy entropy = new ShannonEntropy(data);
for(int i = 1; i < nrOfSections; i++) {
    double sectionEntropy = entropy.forSection(i);
    System.out.println("Entropy for Section " + i + ": " + sectionEntropy);
}
```

An example output of the code above indicates that the content of the first section is compressed or encrypted.

```
Entropy for Section 1: 0.8860567048281903
Entropy for Section 2: 0.4341001902327080
Entropy for Section 3: 0.4696436282541145
```

4.4.3 Signature Scanning

The signature scanning module of *PortEx* is used to detect packers, compilers, and embedded files. Their detection is useful to determine the next steps for analysing the malware. If a packer was found, the malware analysts needs to unpack the file. Identification of the packer enables the malware analyst to use packer-specific unpacking tools if they are available. Knowledge about the compiler helps to determine suitable decompilers. The malware analyst may also want to extract embedded files for further analysis.

PortEx scans for signatures that have the *PEiD* signature format. *PEiD*[16] is a compiler and packer identifier for Windows. It has a graphical user interface

[16] `http://woodmann.com/BobSoft/Pages/Programs/PEiD` (last access Thursday 23rd October, 2014)

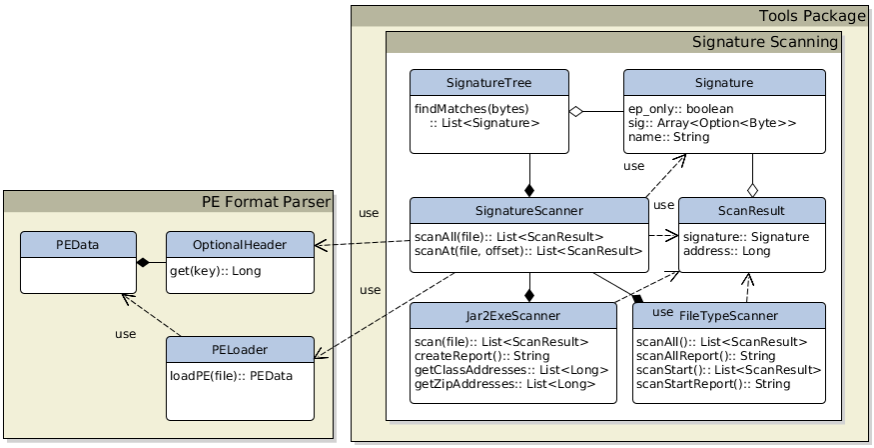

Figure 4.7: Signature scanning package structure and dependencies

and an external signature database that contains 1832 entries. User-customised signature databases with more entries are available on the Internet.

A *PEiD* database entry consists of three parts: the name, the signature, and the ep-only flag. The signature is saved as a sequence of hexadecimal values and allows the use of wildcards (see subsection 2.4.1) denoted with a '?' sign. If the ep-only flag is set, *PEiD* and *PortEx* employ entry-point scanning (see section 2.4.2), otherwise the whole file is searched for the signature. The following listing shows a sample database entry for a *PEiD* signature with the name *AcidCrypt*:

```
1 [AcidCrypt]
2 signature = BE ?? ?? ?? ?? 02 38 40 4E 75 FA 8B C2 8A 18 32 DF C0 CB
3 ep_only = true
```

The advantage of using *PEiD* signatures is the availability of databases on the Internet and the possibility to reuse the own customised signature database for *PortEx*.

The main class of *PortEx*' signature detection package is the `SignatureScanner`. The `SignatureScanner` uses a prefix tree to search for the signatures in a file. The prefix tree is the `SignatureTree` in figure 4.7. A byte sequence of a file is matched against all signatures that are saved in the prefix tree. The worst case runtime-complexity to lookup one signature in a prefix tree is $O(n)$, where n is the length of the signature. *PortEx* returns the matched signatures as `ScanResult`, which consists of a `Signature` and the file offset where it was found at.

The same signature scanning module is used to search for embedded files and jar-to-exe wrapper signatures. The author created a customised signature database for both tasks. The `FileTypeScanner` uses a customised signature list with 468 different signatures to scan for embedded files in the PE file. It is still in development and as such marked as beta-feature in *PortEx*. The author

reverse-engineered packed files by Launch4j, Jar2Exe, JSmooth, and Exe4J to extract signatures for the database of `Jar2ExeScanner`. The database also has packer independent signatures that indicate embedded JAR files or Java bytecode files (file extension *.class*), or calls to *java.exe* or *javaw.exe*.

Example 17 *A sample output of the jar-to-exe wrapper detection is given in the next listing. The scanned file was created with the tool* Launch4j. PortEx *shows matching signatures and the file offset for an embedded JAR.*

```
Signatures found:
  * Jar manifest (strong indication for embedded jar)
  * Launch4j signature
  * PZIP Magic Number (weak indication for embedded zip)
  * Call to java.exe (strong indication for java wrapper)
  * Call to javaw.exe (strong indication for java wrapper)

ZIP/Jar offsets: 0x5c00
```

The following code was used to create the output:

```
Jar2ExeScanner scanner = new Jar2ExeScanner(file);
System.out.println(scanner.createReport());
```

PortEx *can dump the embedded JAR file for further investigation:*

```
Jar2ExeScanner scanner = new Jar2ExeScanner(file); ;
for(Long address : scanner.getZipAddresses()) {
    scanner.dumpAt(address, new File("dump.out"));
}
```

4.4.4 Anomaly Detection

Not only malformations are relevant for malware analysis, but any unusual properties of a file, even if they are permitted by the specification. These properties set the file apart from others, as such they might be used as part of a malware signature or as input for heuristic analysis (cf. [SHRS07, slides 32,35] and see [Szo05, pp. 426–430]). They can also be indicators for malware defence techniques that are not malformation-related. Because of that, *PortEx* collects all kinds of unusual file format properties—so called *anomalies*.

Definition 37 (PE anomaly) PE anomalies *include PE malformations and any unusual or non-default properties of PE files.*

Figure 4.8 illustrates the anomaly package structure of *PortEx*. There is an `AnomalyScanner` realisation for every header and special section that is responsible to find anomalies concerning these structures. The `PEAnomalyScanner` implements all of these scanners as Scala traits, it collects their output and acts as interface for the library user. An `Anomaly` is composed of a description, key, type, and subtype. The key defines the structure or field that is affected by the anomaly. Types and subtypes are explained hereafter.

PortEx differentiates five anomaly types, which are programmatically defined by the enum `AnomalyType`:

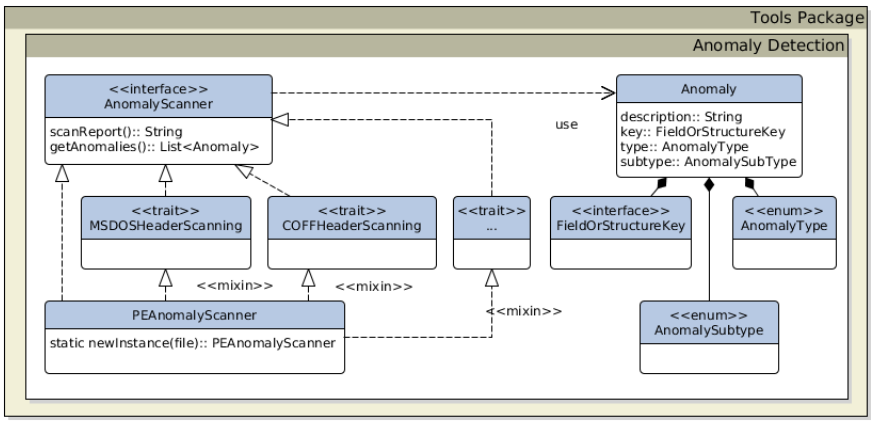

Figure 4.8: Anomaly detection package structure

1. *Non-default anomalies* describe valid properties (according to the PE/-COFF specification) that do not match the default setting or are unusual in their nature. *Non-default anomalies* are no malformations.

2. The *deprecated-value malformation* is a field malformation that concerns the usage of out-dated fields or flags.

3. The *reserved-value malformation* is a field malformation. It relates to fields and flags, whose usage is prohibited, but might be valid in future versions of the PE/COFF specification.

4. *Wrong-value malformations* are field malformations. The concerned fields or flags are invalid for other reasons than being deprecated or reserved, e. g., violation of file alignment restrictions.

5. The *structural malformation* is the last anomaly type. It has been defined in subsection 3.5.2.

In addition to these five anomaly types, there are 77 subtypes that *PortEx* uses. Each subtype specifies a concrete anomaly. An overview of the anomalies that are recognised by *PortEx* is in appendix D. A full list of anomaly subtypes and their corresponding anomaly type can be printed with the following code:

```
1 for(AnomalySubType subtype : AnomalySubType.values()) {
2     System.out.println(subtype + ": " + subtype.getSuperType());
3 }
```

Example 18 demonstrates the usage of *PortEx*' anomaly detection tool.

Example 18 *An anomaly list can be retrieved by* PortEx *using the following code.*

```
1 File file = new File("filepath");
2 PEAnomalyScanner scanner = PEAnomalyScanner.newInstance(file);
3 List<Anomaly> anomalies = scanner.getAnomalies();
```

The following listing shows a sample output:

```
1  Scanned File: VirusShare_8e28f3f765c013eb9eec29c28189a00d
2  * Optional Header: Size of Headers should be 512, but is 4096
3  * Section Header 1 with name UPX0: POINTER_TO_RAW_DATA must be 0 for sections with
       only uninitialised data, but is: 1024
4  * Section Header 1 with name UPX0: SIZE_OF_RAW_DATA is 0
5  * Section name is unusual: UPX0
6  * Section name is unusual: UPX1
7  * Section 1 with name UPX0 (range: 1024--1024) physically overlaps with section
       UPX1 with number 2 (range: 1024--223232)
8  * Entry point is in writeable section 2 with name UPX1
9  * Section Header 3 with name .rsrc has unusual characteristics, that should not be
       there: Write
10 * Resources are fractionated!
```

4.4.5 Overlay Detection

The overlay is used to save data and files, e. g., some packers append the target to the stub, thus, write it to the overlay of the stub. Some compilers and EXE converters use the overlay to save additional information such as code. An example is the Jar-to-EXE wrapper Launch4J[17]. Accurate overlay detection helps to identify embedded files or other information written to the file, and is also necessary to detect certain anomalies.

Calculating the correct *readsize* of a section is crucial to detect overlay (see listing 4.1). Unless there is a sectionless PE file, the offset to the overlay is equal to the largest physical endpoint of the sections. The section headers are ordered by the section's virtual location (see [Mic13, p. 24]). The virtual order of sections does not necessarily equal the physical order (see [Alb13]), so the physical endpoint has to be calculated for every section to determine the largest one. Sectionless PE files must be in low-alignment mode, thus, do not have any overlay because the whole file is part of the image (see section 3.5.2). This case is handled in line 13.

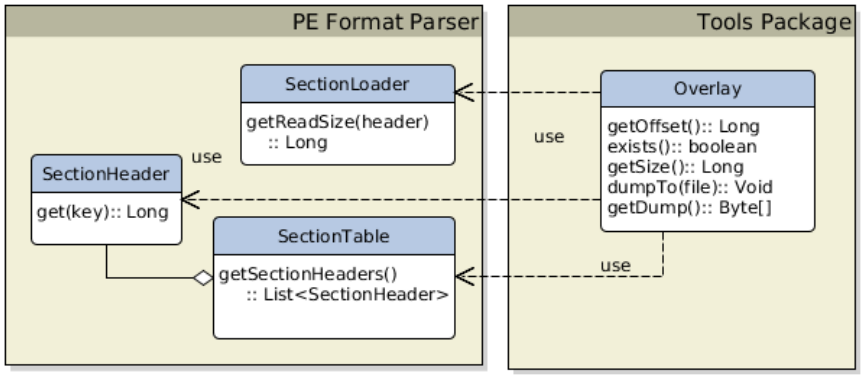

Figure 4.9: Overlay tool structure and dependencies

[17] `http://launch4j.sourceforge.net/` (last access Thursday 23rd October, 2014)

Listing 4.5 shows an algorithm in pseudocode to determine the overlay's offset and size. Albertini states 'if a section starts at [unaligned] offset 0, it's invalid' [Alb13]. These sections are ignored (line 8). Sections with zero readsize are never read from disk, so they are also ignored (line 8).

Listing 4.5: Calculating the overlay

```
1  def getOverlayOffset():
2      offset = 0
3      foreach sectionHeader in sectiontable do
4          sectionStart = sectionHeader.getAlignedPointerToRawData()
5          readsize = sectionHeader.getReadSize()
6          sectionEnd = readsize + sectionStart
7          // ignore invalid and zero-sized sections
8          if readsize == 0 or sectionHeader.getPointerToRawData() == 0 then
9              continue
10         // save largest section end as overlay offset
11         if offset < sectionEnd then
12             offset = sectionEnd
13     if offset == 0 then
14         offset = filesize
15     return offset
16
17
18 def getOverlaySize():
19     return filesize - getOverlayOffset()
20
21 def overlayExists():
22     return getOverlaySize() != 0
```

The dependencies of the overlay tool are illustrated in figure 4.9. The `Overlay` instance uses the Section Table to retrieve the section headers and their physical start addresses. The section loader calculates the *readsize* of each section, which is used to determine the physical end address.

4.4.6 Visualisation of PE files

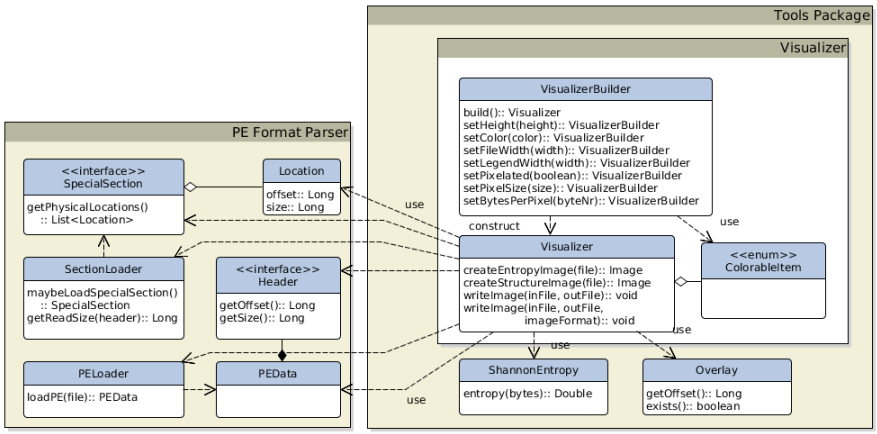

Figure 4.10: Visualiser package structure and dependencies

The visualiser of *PortEx* creates an image that represents the structure of a PE file. It is customiseable, e. g., by the dimensions of the image and the number of bytes a square represents. The visualiser is suitable for getting a quick overview of the currently analysed file

Every object that represents a structure of a PE file in *PortEx* carries its file offset and size. Structures that are composed of several smaller structures collect their physical locations on request. That enables the PE visualiser to draft an image of the file's structure.

The visualiser uses the builder pattern. There are six customisation settings and an additional option to set each color separately. The builder pattern avoids the need for visualiser constructors that have all parameter combinations. Sample usage of the builder and the visualiser is in Listing 4.6.

Listing 4.6: PE Visualiser usage example

```
1  // use the builder to create a visualiser instance
2  Visualizer visualizer = new VisualizerBuilder()
3            .setPixelated(true)
4            .setHeight(800)
5            .setColor(ColorableItem.SECTION_TABLE, Color.BLUE)
6            .build();
7  // create an image that shows the structure of sample.exe
8  File peFile = new File("sample.dll");
9  File outputFile = new File("visualized.png");
10 visualizer.writeImage(peFile, outputFile);
```

The structure and dependencies of the visualiser package are displayed in figure 4.10.

In addition to the file's structure, the visualiser is able to create an image of the local entropies. Figure 4.11 shows an example output with a representation of the local entropy on the left side and the PE file structure on the right side for a *W32.Sality* infected file[18].

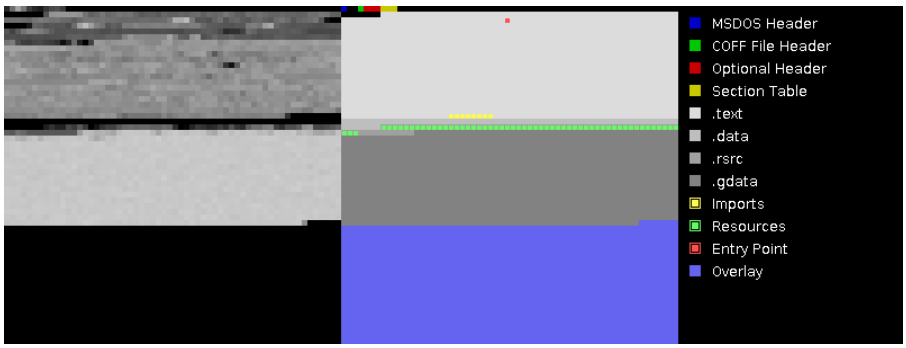

Figure 4.11: PE visualiser, example output for a PE file infected with W32.Sality; the left side shows the local entropy, the right side the structure of the file.

W32.Sality is a polymorphic, entry-point obscuring file infector. The encrypted body of the virus inserts itself into the last section, which is the *.gdata* section

[18] malware sample #191b28bb42ad40340e48926f53359ff5

for the sample file. The virus body appears bright on the entropy image because the encrypted part has a higher entropy than the rest of the file.

4.4.7 Report Creation

The report creator is a convenience tool to get a formatted textual description about a PE file. The report creator collects information from the PE parser, entropy calculator, hash calculator, overlay detector, and anomaly scanner and returns a report string. Example code for report creation is in the following listing. A complete example output is in appendix C.

Listing 4.7: ReportCreator usage example

```
1 // instantiate report creator
2 File file = new File("sample.exe");
3 ReportCreator reporter = ReportCreator.newInstance(file);
4 // print all available information
5 reporter.printReport();
6 // alternatively get a report string for certain contents
7 String coffHeaderReport = reporter.coffHeaderReport();
8 String peidReport = reporter.peidReport();
9 String anomalyReport = reporter.anomalyReport();
```

The report creator can be used to generate the output of command line tools, for testing purposes, or to write text file reports to disk.

4.4.8 Hash Calculation

Malware analysts use hashes to identify and search for malware (see section 2.3.1). *PortEx* calculates hash values for files, sections, and byte arrays. The `Hasher` object takes a `MessageDigest`, which defines the algorithm to calculate the hash values. That means all available `MessageDigest` instances in Java can be used, including MD5 and SHA-256, but also implementations by the library user. The following listing shows how to create hash values for files and sections using *PortEx*.

Listing 4.8: Hash calculator example

```
1  // load PE header data
2  Hasher hasher = Hasher.newInstance(new File("sample.exe"));
3
4  // prepare message digest
5  MessageDigest md5 = MessageDigest.getInstance("MD5");
6
7  // create and print MD5 hash for the file
8  byte[] hash = hasher.fileHash(md5);
9  System.out.println("file hash: " + ByteArrayUtil.byteToHex(hash, ""));
10
11 // create and print SHA-256 hash for the first section
12 int sectionNumber = 1;
13 hash = hasher.sectionHash(sectionNumber, md5);
14 System.out.println("section hash: " + ByteArrayUtil.byteToHex(hash, ""))
```

4.4.9 String Extraction

String extraction can reveal information like email addresses, passwords, file names, embedded code, dialog strings, and process names. *PortEx* has a class called `StringExtractor` that extracts ASCII and Unicode strings from files. A code example is in the following listing.

Listing 4.9: String extractor example

```
1 int minLength = 4;
2 List<String> strings = StringExtractor.readStrings(file, minLength);
```

An excerpt of the output for malware #baf3492d31705048e62d6f282a1ede8d shows an absolute file path that includes the user name of the malware author:

```
1 C:\Users\Jontes station\Desktop\Hack Stuff\Minecraft Force Op Hack (2013)
2 \Minecraft Force Op Hack (2013)\obj\x86\Debug\Minecraft Force Op Hack (2013).pdb
```

4.5 Summary

The static PE analysis library *PortEx* is a platform independent solution for malware analysts and software developers of reverse-engineering tools. Its buildup considers the API design principles by Bloch to provide easy usage and backward compatibility. It supports integration to Maven and SBT projects.

The feature section described usage, structure, and implementation of eight malware analysis features and provided parsing robustness solutions for at least 30 different PE malformations. It is yet left to evaluate the malformation robustness of *PortEx* and how it competes with similar products. This is part of chapter 5.

Chapter 5

Evaluation

The *Evaluation* chapter analyses the quality of the library *PortEx*. Section 5.1 compares *PortEx*' features with the features of three other PE analysis tools and libraries. Robustness tests for *PortEx* and the other PE analysis products are performed in section 5.2. The last section presents statistics about PE malware, clean PE files, and their anomalies. It suggests boosters and stoppers for heuristic analysis based on the statistical results.

The following two sections have been removed for the public version. They compare features and robustness of *PortEx* and three other libaries. The reasons for removal are:

- The comparison is already out-of-date, because all of the libraries have been updated in the meantime.
- I reported the bugs that I found by the robustness tests to the authors, a lot of them have already been fixed. It would be of no use for anyone to see a list of old bugs.
- A robustness comparison is not valid if done by one of the authors (me). Robustness is based on knowledge about malformations. I can only test, what I know, thus, I am in no position to perform a valid comparison.
- I am biased.

To sum it up: I do not see any use of an outdated and biased comparison. It will not help anyone to pick the right library, nor will it help the authors, because they already know everything they have to know.

5.1 Feature Comparison

Removed for public version.

5.2 Malformation Robustness Tests

Removed for public version.

5.3 Statistics by PortEx

This section presents an analysis of malicious and clean PE files. It provides general statistical information and determines the suitability of PE anomalies as heuristic boosters or stoppers.

5.3.1 Test Files

Statistical information is gathered from two categories of test sets: malicious files and clean files.

Malicious Test Files: the BAD and the WORSE set

The malicious test files are 131 072 files that where uploaded to VirusShare[1] on April 2014 as torrent 128. VirusShare is a private malware repository that grants access to malware researchers via an invitation. The files are of any file format. The set containing all files from torrent 128 is hereby called WORSE. The author used *PortEx* to extract the subset of PE files from the WORSE test set. The subset has 103 275 samples and is herewith called BAD.

Clean Test Files: the GOOD set

The set of clean test files consists of 49 814 PE samples. The files are taken from fresh installs of Windows 7 64-bit, Windows XP 64-bit, Windows Vista 64-bit, and Windows 8 using the tool *CleanPECollector*[2], which is based on *PortEx*. The author makes the assumption that a fresh operating system installation only contains clean files. The set of clean PE test files is herewith called GOOD.

Control Sets: the $\mathbf{GOOD_{CS}}$ and the $\mathbf{BAD_{CS}}$ set

The GOOD and the BAD set have a corresponding control set each. The control sets do not contain any files from GOOD or BAD, they are used to test the transferability of certain results. The $\mathrm{BAD_{CS}}$ set is the control set for BAD. It consists of 111 043 malicious PE files from torrent 131 of *VirusShare* (uploaded on May 2014). The $\mathrm{GOOD_{CS}}$ set is the control set for GOOD. It consists of

[1] http://virusshare.com/
[2] https://github.com/katjahahn/CleanPECollector

12 305 clean PE files, which where collected the same way as the files from the GOOD set from a Windows XP 32-bit and a Windows 7 32-bit machine.

5.3.2 Booster Score

The statistical data collected from the test sets includes file properties to determine which of them are suitable for heuristic analysis. The author calculates a score for each property that represents the applicability as booster or stopper for heuristic analysis. This score of a file property p is called *booster score of* p, short $BScore(p)$. The booster score is in the interval $I = [-10, 10]$, with 10 representing the best suitability of a property as booster and -10 the best suitability as stopper. A booster score of 0 means the property cannot be used as booster or stopper because the property is equally frequent in clean and malicious files. The *booster score* of property p is defined as follows.

$$BScore(p) = \frac{p_{\text{bad}}}{p_{\text{bad}} + p_{\text{good}}} * 20 - 10$$

p_{bad} is the relative frequency of files in the BAD set that have file property p. p_{good} is the relative frequency of files in the GOOD set that have file property p.

The following definitions set a threshold for the booster score to define a file property as *suitable booster* or *suitable stopper*.

Definition 38 (suitable booster) *A file property p is a* suitable booster *for heuristic analysis iff $BScore(p) \geq 5.0$.*

Definition 39 (suitable stopper) *A file property p is a* suitable stopper *for heuristic analysis iff $BScore(p) \leq -5.0$.*

5.3.3 Conditional Probabilities

The author calculates conditional probabilities as a second measure to determine how well properties can be used as boosters or stoppers. The conditional probability P(B|C) is defined as follows:

B is the event of a file being in the BAD set. C is the event of a file having property p. G is the event of a file being in the GOOD set. If a file has property p, P(B|C) is the probability of this file being malicious and P(G|C) is the probability of this file being clean. Let G, B and C be events and G be the complementary event of B. The conditional probability P(B|C) is calcuated by the following formula (see [SK06, p. 2]):

$$P(B|C) = \frac{P(C|B)P(B)}{P(C)} = \frac{P(C|B)P(B)}{P(C|B)P(B) + P(C|G)P(G)}$$

The probabilities P(B) and P(G) are *non-informative priors* because the base rate of malicious and clean files is unknown. The overall test set BAD ∪ GOOD does not reflect the base rates in reality because the sets were collected independently. The events B and G are mutually exclusive and collectively exhaustive, so the priors P(B) and P(G) are set to 0.5 based on the *principle of indifference* (see [Key21, pp. 44–70]). That means if there is no information about a file, we define the events G and B for this file as equally probable. A probability P(B|C) of 0.5 is equivalent to a booster score of 0.0.

5.3.4 Malware File Types

Table 5.1 shows the percentage and number of PE files and non-PE files in the WORSE set.

Table 5.1: Malware filetypes of the WORSE set

Filetype	Absolute Number	Percentage
PE	103275	78.79 %
No PE	27797	21.21 %

The statistical data of the filetypes confirms that the majority of malware has the PE format because 78.79 per cent of the WORSE set are PE files (see Table 5.1).

Table 5.2: PE malware filetypes of the BAD set

Filetype	Absolute Number	Percentage
PE32	103253	99.98 %
PE32+	22	0.02 %

Table 5.2 shows the number of PE32 and PE32+ files in the BAD set. The target platform of PE32+ files is restricted to 64-bit platforms, whereas PE32 files can run on 32 and 64-bit systems. That means a file that uses the PE32+ format is not able to run on as many systems as the same file that uses the PE32 format. Malware authors, who try to infect as many systems as possible, will also strive for compatibility with most systems. So it comes of no surprise that malicious PE32+ files are rare.

5.3.5 Anomaly Prevalence

The author collected information about the prevalence of anomalies in the BAD set and the GOOD set. The booster score and the conditional probability P(B|C) for each anomaly type are calculated. The results are shown in table 5.3.

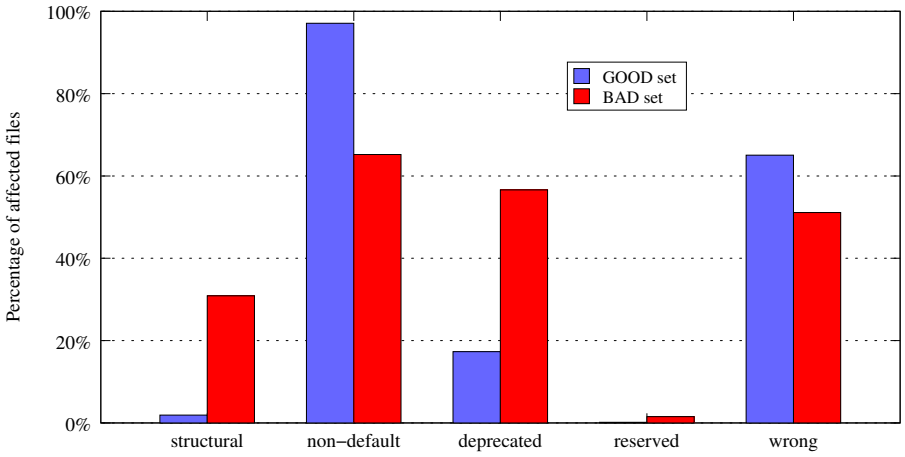

Figure 5.1: PE files with at least one anomaly of specified type

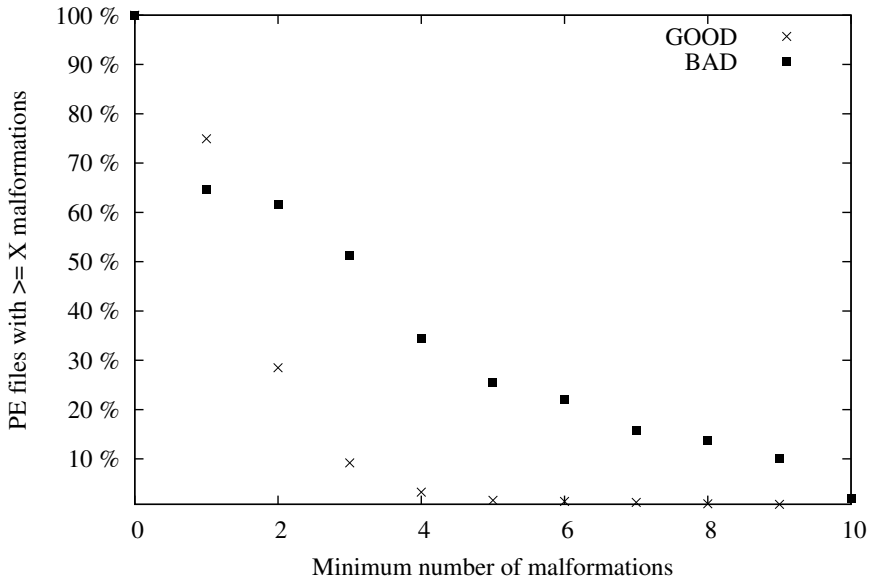

Figure 5.2: Percentage of PE files with more than or exactly X malformations

Table 5.3: Anomaly prevalence

	GOOD	**BAD**	**BScore**	**P(B|C)**
Percentage of files with at least one anomaly of type				
structural	1.90 %	30.88 %	8.84	94.20 %
non-default value	97.08 %	65.22 %	-1.96	40.18 %
deprecated value	17.33 %	56.60 %	5.31	76.55 %
reserved value	0.17 %	1.52 %	7.99	89.94 %
wrong value	65.05 %	51.12 %	-1.20	44.00 %
Average number of anomalies per file				
total anomalies	3.3903	6.1922		
total malformations	1.2770	3.0800		
structural	0.1226	0.6030		
non-default value	2.1133	3.1122		
deprecated value	0.3800	1.4662		
reserved value	0.0031	0.0172		
wrong value	0.7713	0.9936		

Structural malformations are indicative for malware. If a file has at least one structural malformation, there is a probability of 94.20 per cent that the file is malicious. Usage of reserved values is rare for BAD and GOOD files, but the percentage of BAD files using them is higher. Deprecated values are also more prevalent in malware than in clean files. Both deprecated and reserved values are a possibility to store infection markers, which might explain why these anomalies are more common in malware than in clean files. Deprecated and reserved values are suitable boosters with booster scores of 5.31 and 7.99.

There is only one anomaly type that exists in more GOOD files than BAD files, which is the non-default value anomaly. 97.08 per cent of all GOOD files have at least one non-default value compared to 65.22 per cent of all BAD files. However, the non-default value anomaly is still more frequent in BAD if counted per file: BAD files have on average 3.1122 non-default anomalies, GOOD files have 2.1133.

Malformations are all anomaly types except for the non-default anomaly. Table 5.4 and figure 5.2 show that there is a higher percentage of GOOD files with at least one malformation (74.92 per cent) than BAD files (64.73 per cent) with at least one malformation.

Table 5.4: Percentage of files with at least X malformations

X	GOOD	BAD	BScore	P(B|C)
1	74.93 %	64.73 %	-0.7303	46.35 %
2	28.48 %	61.57 %	3.6746	68.37 %
3	9.19 %	51.22 %	6.9574	84.87 %
4	3.25 %	34.37 %	8.2722	91.36 %
5	1.59 %	25.54 %	8.8279	94.14 %
6	1.56 %	21.96 %	8.6735	93.37 %
7	1.36 %	15.79 %	8.4140	92.07 %
8	1.18 %	13.78 %	8.4225	92.11 %
9	0.90 %	10.03 %	8.3532	91.77 %
10	0.78 %	1.90 %	4.1791	70.90 %

This result seems counterintuitive as malware writers include malformations to deceive or break analysis tools. However, we can derive from table 5.4 that 46.45 per cent of all GOOD files have exactly one malformation. BAD files with malformations have usually more of them than GOOD files, e. g., 51.22 per cent of all BAD files have more than two malformations, but only 9.19 per cent of all GOOD files.

The results in table 5.3 support this: BAD files have 3.0800 malformations on average, GOOD files have 1.2770. The occurrence of more than two malformations in a file is a suitable booster (see table 5.4).

The author determined for every anomaly subtype the number of files in GOOD and BAD that have an anomaly of this subtype. Results with less than 500 involved files are ignored. Table 5.5 shows percentages, booster score, and conditional probabilities for every anomaly subtype. The rows are sorted by the booster score, so the first and last entries are the most relevant boosters and stoppers for heuristic analysis. The results are rounded to two decimal figures.

Table 5.5: Prevalence of anomaly subtypes

Anomaly subtype	GOOD	BAD	BScore	P(B|C)
Percentage of files with at least one anomaly of subtype				
collapsed MS-DOS Header	0.00 %	0.81 %	10.00	100.00 %
SizeOfImage not aligned	0.00 %	2.34 %	9.98	99.91 %

Continued on next page

Table 5.5 – *Continued from previous page*

Anomaly	**GOOD**	**BAD**	**BScore**	**P(B|C)**
too large SizeOfRawData	0.01 %	2.04 %	9.90	99.51 %
uninit. data constraints violation	0.14 %	12.91 %	9.79	98.95 %
entry point in last section	0.06 %	5.06 %	9.75	98.75 %
invalid data directory	0.03 %	1.46 %	9.65	98.25 %
reserved data directory	0.03 %	1.31 %	9.49	97.46 %
fractionated data	0.47 %	14.46 %	9.37	96.83 %
SizeOfRawData not aligned	0.09 %	2.62 %	9.32	96.59 %
PtrOfLineNr set (deprectated)	0.13 %	3.67 %	9.30	96.51 %
NrOfLineNr set (deprecated)	0.12 %	2.89 %	9.19	95.93 %
writeable only section	0.52 %	11.38 %	9.13	95.63 %
sections phys. overlapping	1.30 %	26.95 %	9.08	95.38 %
PtrToReloc set (deprecated)	0.06 %	1.09 %	8.98	94.91 %
phys. duplicated section	0.03 %	0.47 %	8.64	93.21 %
entry point in writeable section	3.12 %	26.12 %	7.86	89.32 %
SizeOfRawData zero	5.76 %	46.62 %	7.80	89.01 %
SizeOfHeaders non-default	2.52 %	15.91 %	7.26	86.32 %
SizeOfHeaders not aligned	0.20 %	1.13 %	6.95	84.75 %
writeable and executeable section	6.63 %	26.27 %	5.97	79.85 %
unusual section name	14.23 %	51.07 %	5.64	78.21 %
deprecated file characteristics	17.24 %	55.10 %	5.23	76.17 %
unusual section characteristics	21.11 %	42.72 %	3.39	66.93 %
control symb. in section name	1.57 %	3.15 %	3.35	66.76 %
non-default file alignment	8.37 %	8.10 %	-0.16	49.19 %
NrOfSymbols set (deprecated)	1.12 %	0.79 %	-1.73	41.33 %
section virtually overlapping	0.69 %	0.38 %	-2.89	35.54 %
low-alignment mode	0.84 %	0.45 %	-3.00	35.01 %
PtrToSymbTable set (deprecated)	1.73 %	0.91 %	-3.14	34.32 %
file alignment too small	0.84 %	0.29 %	-4.94	25.31 %

Continued on next page

Table 5.5 – *Continued from previous page*

Anomaly	GOOD	BAD	BScore	P(B|C)
section virtually duplicated	0.82 %	0.23 %	-5.57	22.13 %
non-default ImageBase	93.87 %	7.45 %	-8.53	7.35 %
virtual entry point	27.63 %	0.15 %	-9.89	0.54 %
too large ImageBase	41.94 %	0.02 %	-9.99	0.05 %

Table 5.5 has 22 suitable boosters and four suitable stoppers by definitions 38 and 39. At least five of them are already known as malware indicators.

1. The entry point in the last section is an indicator for a virus infection (see [Szo05, p. 427]).
2. Sections with zero `SizeOfRawData` are often a result of packer usage [DN12, p. 478].
3. A non-aligned `SizeOfImage` value is described as virus indicator (see [Szo05, p. 427]).
4. Sections containing virus code are often marked as writeable and executable, or as writeable only (see [Szo05, p. 427]). This is suspicious because code sections usually do not need a writeable attribute (see [Szo05, p. 427]).
5. Unusual section names are flagged as suspicious by *PEStudio*.

The other suitable boosters and stoppers in table 5.5 might provide new properties for heuristic analysis. Usage of reserved fields, reserved data directory entries, deprectated fields, or deprecated flags can be an infection marker for viruses or used as part of a decryption key by packed files or encrypted viruses. Violation of constraints for unitialized data may occur if the author of a packer or file infecting virus does not update these values properly. Fractionated data can be a result of a virus infection (see section 3.5.2). Non-aligned `SizeOfRawData` fields might stem from file infecting viruses that enlarge a section or add a section to copy themselves into it, but fail to align the field properly.

5.3.6 File Scoring Based on Booster Scores

Heuristic analysis is done in two steps as described in subsection 2.4.6. The first step is data gathering: The file is parsed to find properties, which are boosters or stoppers. The second step is data analysis, e. g., by assigning weights to boosters and stoppers and calculating the sum; if the sum is larger than a certain treshold, the file is labelled as malicious. This section evaluates the

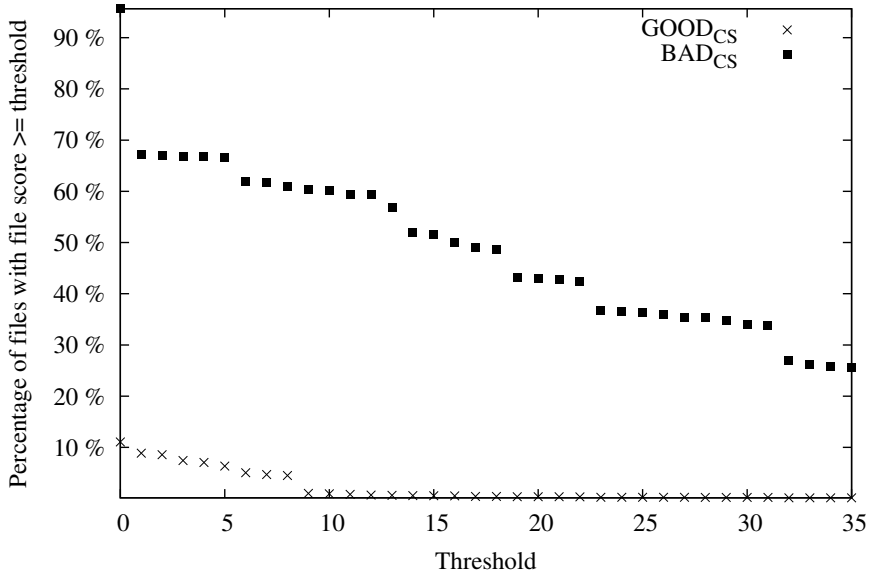

Figure 5.3: Percentage of files whose file score is greater than or equal to a given threshold using the unselective scanner

suitability of the booster score as a property's weight for the data analysis step. All anomaly-based booster scores of table 5.5 are used.

Let p be the set of all properties p_i of file f where $1 \leq i \leq |p|$ and $i \in N$. The *file score* of f is calculated by the following formula:

$$FileScore(p) = \sum_{i=1}^{|p|} BScore(p_i)$$

By using file anomalies as properties we determine the *file score* for all files in the control sets GOOD_{CS} and BAD_{CS}. The control sets are used because they are not the basis for the statistical data about anomaly prevalence. Figure 5.3 shows the percentage of files that have a larger file score than a certain threshold.

The treshold is used to determine heuristically if a file is malicious. The graph in figure 5.3 shows the percentage of detected files for different thresholds. A heuristic scanner that has a threshold of 15 detects 51.73 per cent of all malicious files and has a false positive rate of 0.55 per cent (see table 5.6).

Such a heuristic scanner is not sufficient on its own. For actual antivirus products it is crucial to have false positive rates that are near zero. These products combine several methods to achieve these results, including white lists of clean files. Such optimisations are ignored for the purpose of this test.

Antivirus scanners also need better detection rates than the heuristic scanner in figure 5.3. This scanner classifies all files into malicious or clean regardless if enough data was available for the given file. The scanner is *unselective*. An

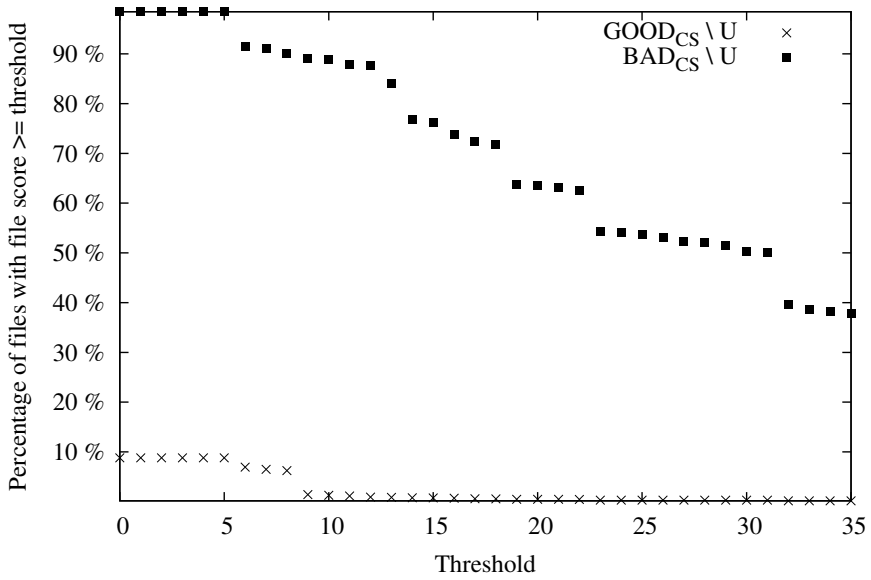

Figure 5.4: Percentage of files whose file score is greater than or equal to a given threshold using the selective scanner

actual antivirus scanner uses several detection techniques in combination. If a file cannot be classified by one detection technique, the antivirus scanner will resort to other techniques. That means it is of interest to filter files that are suitable for anomaly-based heuristic analysis, and evaluate the detection rate for the filtered files.

Let C be the set of files that have an absolute file score greater than or equal to 5.0 and let set U be the complement of set C, i. e. $U = \overline{C}$. The heuristic scanner that accepts only files from set C is herewith called *selective scanner*. The heuristic scanner that classifies all files is herewith called *unselective scanner*.

The detection rates and false positive rates for $\mathrm{GOOD_{CS}} \setminus U$ and $\mathrm{BAD_{CS}} \setminus U$ are illustrated in figure 5.4. The results for the selective and the unselective scanner are in table 5.6. There is an improvement of the detection rate for the *selective scanner* compared to the *unselective scanner*, e. g., the detection rate based on threshold 15 increased from 51.63 per cent to 76.26 per cent (see table 5.6).

The costs for the improved detection rate is the percentage of files that cannot be classified by the selective scanner. The selective scanner rejects 27.29 per cent of the $\mathrm{GOOD_{CS}}$ and 32.30 per cent of the $\mathrm{BAD_{CS}}$ set. The implementation of more anomaly subtypes and other file properties might decrease the rejection rate.

Table 5.6: Percentage of files labelled as malicious based on file scoring thresholds

Treshold	$\mathbf{GOOD_{CS}}$	$\mathbf{BAD_{CS}}$	$\mathbf{GOOD_{CS}}$ \U	$\mathbf{BAD_{CS}}$ \U
0	11.06 %	95.66 %	8.81 %	98.47 %
1	8.87 %	67.14 %	8.81 %	98.47 %
2	8.57 %	67.07 %	8.81 %	98.47 %
3	7.42 %	66.84 %	8.81 %	98.47 %
4	7.05 %	66.71 %	8.81 %	98.47 %
5	6.36 %	66.66 %	8.81 %	98.47 %
6	5.05 %	61.91 %	6.95 %	91.45 %
7	4.69 %	61.65 %	6.49 %	91.08 %
8	4.50 %	60.95 %	6.25 %	90.04 %
9	1.02 %	60.29 %	1.42 %	89.06 %
10	0.92 %	60.10 %	1.27 %	88.78 %
11	0.83 %	59.46 %	1.13 %	87.83 %
12	0.67 %	59.28 %	0.91 %	87.57 %
13	0.63 %	56.85 %	0.85 %	83.98 %
14	0.57 %	52.04 %	0.79 %	76.87 %
15	0.55 %	51.63 %	0.76 %	76.26 %
16	0.51 %	49.93 %	0.69 %	73.76 %
17	0.42 %	49.01 %	0.58 %	72.40 %
18	0.41 %	48.60 %	0.57 %	71.79 %
19	0.36 %	43.20 %	0.48 %	63.82 %
20	0.36 %	43.04 %	0.48 %	63.57 %
21	0.35 %	42.79 %	0.47 %	63.22 %
22	0.34 %	42.37 %	0.45 %	62.59 %
23	0.25 %	36.81 %	0.30 %	54.38 %
24	0.24 %	36.64 %	0.30 %	54.12 %
25	0.24 %	36.30 %	0.30 %	53.62 %
26	0.24 %	35.95 %	0.30 %	53.11 %

Continued on next page

Table 5.6 – *Continued from previous page*

Treshold	**GOOD$_{CS}$**	**BAD$_{CS}$**	**GOOD$_{CS}$ \U**	**BAD$_{CS}$ \U**
27	0.24 %	35.41 %	0.30 %	52.31 %
28	0.24 %	35.29 %	0.30 %	52.12 %
29	0.24 %	34.85 %	0.30 %	51.49 %
30	0.24 %	34.00 %	0.30 %	50.23 %
31	0.24 %	33.90 %	0.30 %	50.08 %
32	0.15 %	26.87 %	0.17 %	39.68 %
33	0.15 %	26.12 %	0.17 %	38.59 %
34	0.15 %	25.82 %	0.17 %	38.14 %
35	0.15 %	25.59 %	0.17 %	37.80 %

The implementation of the heuristic scanners in this test is simple. It does not have any optimisations, e. g. the booster scores of all anomaly subtypes in a file are summed up regardless if some anomaly subtypes are just special cases of others (e. g., a duplicated section is a special case of an overlapping section). The purpose of this test is to show the booster score as an appropriate measurement for the usefulness of file properties in heuristic analysis. The test is successful in this regard, and no optimisations of the scanners are required to achieve this. The test uses the control file sets, that means the booster score is applicable to other file sets than GOOD and BAD.

The file scoring module of *PortEx* does not have a threshold for detection. The module is meant as indicator of a file's maliciousness, but not for malware detection.

5.3.7 Entropy Statistics

From each test set, BAD and GOOD, 10 000 test files are used to collect statistics about the section entropies. The entire test sets are too large to read all file entropies in a reasonable time. The files are chosen arbitrarily. Table 5.7 shows the results.

Section entropies above 0.75 is an indicator for encryption or compression, which is, among others, applied by viruses and packers. Entropies below 0.25 indicate repetitive values and presence of code caves. The results in table 5.7 confirm that the existence of at least one section with a very high entropy is a suitable booster. BAD files have on average approximately one more section with a high entropy than GOOD files, whereas GOOD files have approximately one more section with neutral entropy. Based on the booster score and the conditional

Table 5.7: Section entropy statistics

Entropy	**GOOD**	**BAD**	**BScore**	**P(B|C)**
Percentage of files with at least one section with specified entropy H				
$H > 0.90$ (very high)	2.31 %	47.21 %	9.0670	95.34 %
$H > 0.75$ (high)	45.74 %	94.60 %	3.4815	67.41 %
$0.25 \leq H \leq 0.75$ (neutral)	98.61 %	88.14 %	-0.5378	47.20 %
$H < 0.25$ (low)	56.07 %	55.21 %	-0.7728	49.61 %
$H < 0.10$ (very low)	39.19 %	52.50 %	1.4516	57.26 %
Average number of sections per file with specified entropy H				
$H > 0.90$ (very high)	0.0238	0.5998		
$H > 0.75$ (high)	0.6194	1.6288		
$0.25 \leq H \leq 0.75$ (neutral)	3.0998	2.1871		
$H < 0.25$ (low)	1.0066	1.0352		
$H < 0.10$ (very low)	0.5933	1.0520		

probability it is adviceable to use a section entropy greater than 0.9 as heuristic booster.

5.3.8 Summary

The collected statistical information confirms the high relevance of PE malware analysis because 79 per cent of the WORSE set have the PE format. Malicious PE32+ files are rare, they make up only 0.02 per cent of the BAD set. Malformation statistics show that malicious files have on average more malformations than clean files; but more GOOD than BAD files have at least one malformation. The latter is surprising and needs further investigation to find the reasons.

The booster scores for anomalies depend on the statistical information of the sets BAD and GOOD, but were successfully used as weights for heuristic analysis for the control sets. That means the statistical results are applicable for other file sets and the booster score is a useful measure for file properties. Conditional probabilites are an alternative measure.

Although malformations impose problems for PE parsers, they are advantageous for heuristic analysis. The number of malformations, their type, and their subtype can be suitable properties.

Chapter 6

Conclusion

This master thesis is set out to find and implement solutions for PE format-related malware defence. Malformations affect tools that parse the PE format, making it possible for malware to evade detection by antivirus scanners and to prolong its analysis. That offers the malware more time to spread and harm.

The following section addresses the findings of the present thesis. Implications are stated in section 6.2. Section 6.3 lists the limitations imposed by the scope of the present thesis and regarding the methodology of robustness tests and statistical evaluation. Future plans for the PE library *PortEx* and prospects for software developers and malware researchers are covered in section 6.4.

6.1 Findings

The PE/COFF specification is a misleading document that does not state corner cases, presents fields as mandatory, which are never used by the loader, uses confusing names, and contains descriptions that do not match the actual purpose of certain structures or fields. Proper parsing of the PE format is not possible based on the PE/COFF specification alone. It requires knowledge about malformations, knowledge about the behaviour of the Windows loader for all available Windows versions, and access to malformed files for testing.

The static analysis library *PortEx* is a product of the present thesis. It is proven to be robust against known PE malformations. *PortEx* is hardened against 269 malformed proof-of-concept files by Ange Albertini and a malware collection of 103 275 files from VirusShare.

The hardening process reveals common problems that occur when malformed files are parsed, and a malformation that is not listed by Ange Albertini [Alb13] nor Pericin and Vuksan [Rev11]: fractionated data. Test files that contain this malformation are listed in appendix B. The present thesis describes robustness

solutions for static parsers for these cases. The library *PortEx* serves as example implementation and basis to build robust PE analysis tools.

The comparison of *PortEx* with three other PE analysis products shows that *PortEx* is equally feature-rich and can keep up with the available software. Robustness tests reveal problems in three popular PE analysis libraries and tools. All of them are vulnerable to PE malformations. The issues have been reported to the respective authors.

Statistical data extracted by *PortEx* reveals the most common PE anomalies in malicious and clean PE files. Their usefulness for heuristic analysis is estimated by conditional probabilities and a self-created booster score. 26 anomalies and five additional properties have been identified as suitable heuristic boosters and stoppers this way. Additional tests demonstrate the booster score as a useful weight for heuristic analysis.

6.2 Implications

The present thesis introduces a new classification for *PE malformations* and *anomalies*, and proposes a distinction for both terms that sees PE malformations as a subset of PE anomalies. Vuksan's and Pericin's definition for the term *malformation* only accepts deliberate PE file modifications. The present thesis expands this definition to include also accidental modifications because their effect on PE parsers is the same and the intention might be unknown.

PortEx is at present (November 2014) the only up-to-date Java based PE library known to the author. It serves as a robust and platform independent back-end for PE analysis tools, but hopefully also as a role model and sample implementation for other authors of static PE parsers.

The list of test files in appendix B supplements the proof-of-concept files by Ange Albertini with more malformed samples from VirusShare. Software developers are free to use these additional samples to harden their PE parsers.

The anomaly detection module of *PortEx* serves malware analysts as an additional detection tool for malformation-related problems during analysis. *PEStudio* already covers a wide range of anomalies, but they are different from the anomalies detected by *PortEx*.

Statistical results about anomaly prevalence in malicious and clean files enrich the antivirus community with PE format-based patterns for heuristic analysis. The present thesis proposes an additional measure for the usefulness of file properties for heuristic analysis.

6.3 Limitations

The present thesis focalises on basic static analysis. Any anomalies or algorithms that are instruction-based or require the file to be executed are left out.

The GOOD set that is used to extract statistical information about clean files only contains files that are pre-installed on Windows operating systems. The extraction of files from fresh Windows installations makes it likely that the files are not malicious. But the GOOD set does not contain any files that are downloaded by the user after the operating system installation. The statistical findings might differ if files other than those provided by Microsoft are included in the test set.

The robustness tests for PE parsers in section 5.2 are biased because they are only based on malformations known to the author. This problem cannot be avoided unless a third party performs such tests.

6.4 Prospects

It is planned to extend *PortEx* with more parser features: thread local storage (TLS) parsing, certificate parsing, and load config parsing. The number of detectable anomalies shall be increased in future versions. An option to edit PE files may enable users to remove malformations with *PortEx* so they will not affect other analysis tools.s

Malware analysis tools often have their own PE parsers implemented. Since PE malformations can cause crashes, buffer overflows, and other problems, and because malformation robustness is difficult to achieve (see section 3.6), it is adviceable to use PE libraries that have been hardened against PE malformations. The robustness tests in section 5.2 reveal *pefile* and *PortEx* to be robust for all or most tested malformations.

The author suggests to put further research into anomalies of other file formats which are commonly used by malware, e. g., PDF. As the present thesis focalises on static PE parsing, solutions for dynamic PE parsing robustness have yet to be found.

Further statistical analysis may include section names, certain strings in the file, overlay prevalence and size, the entropy of headers and overlay, prevalence of caves, import names, frequent anomaly or import combinations, section sizes, physical or virtual gaps between sections.

Antivirus companies have repositories of clean files, which they use to evaluate the quality of their detection software and to ensure that the false positive rate is near zero. These repositories contain more than the pre-installed files from the operating system. It is suggested to repeat statistical analysis of anomalies with these test sets and adjust the booster scores accordingly.

6.5 Summary

The PE format is complex and robust PE parsing must consider the behaviour of all 32- and 64-bit Windows operating systems. It is unpredictable how many malformations are still unknown, which malformations will be possible with new Windows releases, and how they will affect analysis and antivirus software. Research in finding and documenting malformations must proceed as long as PE files are used. The present thesis contributes by raising awareness on possible consequences, describing solutions for robust parsing, providing a free and robust analysis library for public use, and turning anomalies from a disadvantage into an advantage for malware detection by using them as heuristic boosters or stoppers.

Bibliography

[Alb12] Ange Albertini. Binary art; Byte-ing the PE that fails you. https://code.google.com/p/corkami/wiki/hashdays2012 (last access on Oct. 2014), November 2012.

[Alb13] Ange Albertini. PE; The Portable Executable Format on Windows. https://code.google.com/p/corkami/wiki/PE (last access on Oct. 2014), October 2013.

[Ayc06] John Aycock. *Computer Viruses and Malware*. Springer, 2006.

[Bla11] BlackHat Briefings. *BlackHat USA 2011, Las Vegas*, 2011. https://www.blackhat.com/ (last access on Oct. 2014).

[Blo09] Joshua Bloch. How to Design a Good API and Why it Matters. http://lcsd05.cs.tamu.edu/slides/keynote.pdf (last access on Oct. 2014), 2009.

[BO11] Randal E. Bryant and David R. O'Hallaron. *Computer Systems, A Programmer's Perspective*. Prentice Hall, 2 edition, 2011.

[Bon94] Vesselin Bontchev. Possible Virus Attacks Against Integrity Programs and How to Prevent Them. http://repo.hackerzvoice.net/depot_madchat/vxdevl/vdat/epposatt.htm (last access on Oct. 2014), 1994.

[Bri04] Léon Brillouin. *Science and Information Theory*. Courier Dover Publications, 2004.

[Coh84] Fred Cohen. Computer Viruses - Theory and Experiments; Introduction and Abstract. http://web.eecs.umich.edu/~aprakash/eecs588/handouts/cohen-viruses.html (last access on Oct. 2014), 1984.

[DN12] Dhruwajita Devi and Sukumar Nandi. PE File Features in Detection of Packed Executables. In *International Journal of Computer Theory and Engineering*, volume 4, pages 476–478. TODO, June 2012.

[HL] David Harley and Andrew Lee. Heuristic Analysis—Detecting Unknown Viruses. http://www.eset.com/us/resources/white-papers/Heuristic_Analysis.pdf (last access on Oct. 2014).

[JS12] Suman Jana and Vitaly Shmatikov. Abusing File Processing in Malware Detectors for Fun and Pro fit. https://www.cs.utexas.edu/~shmat/shmat_oak12av.pdf (last access on Oct. 2014), 2012.

[Kat13] Randy Kath. The Portable Executable File Format from Top to Bottom. http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile2.html (last access on Oct. 2014), 2013.

[Key21] John Maynard Keynes. *A Treatise On Probability*. Macmillan And Co., Limited, 1921.

[Lis10] Alexander Liskin. PE: specification vs. loader. http://www.kaspersky.com/images/alexander_liskin_-_pe_specification_vs_pe_loader.pdf (last access on Oct. 2014), 2010.

[Mic02] Microsoft Corporation. *MSDN Magazine*, 2002. http://msdn.microsoft.com/en-us/magazine/default.aspx (last access on Oct. 2014).

[Mic07] Microsoft Corporation. What is a DLL? https://support.microsoft.com/kb/815065/EN-US (last access on Oct. 2014), December 2007.

[Mic13] Microsoft Corporation. *Microsoft PE and COFF specification*, February 2013. revision 8.3.

[Mic14] Microsoft Corporation. Naming malware. http://www.microsoft.com/security/portal/mmpc/shared/malwarenaming.aspx (last access on Oct. 2014), 2014.

[MMF10] Samir Mody, Igor Muttik, and Peter Ferrie. Standards and policies on packer use. In *Virus Bulletin Conference 2010*, September 2010. https://www.virusbtn.com/index (last access on Oct. 2014).

[OMR] Markus F.X.J. Oberhumer, László Molnár, and John F. Reiser. UPX. http://upx.sourceforge.net/ (last access on Oct. 2014).

[Pan14] PandaLabs. Quarterly Report; January-March 2014. http://press.pandasecurity.com/wp-content/uploads/2014/05/Quaterly-PandaLabs-Report_Q1.pdf (last access on Oct. 2014), 2014.

[Pie02a] Matt Pietrek. An In-Depth Look into the Win32 Portable Executable File Format. In *MSDN Magazine* [Mic02]. http://msdn.microsoft.com/en-us/magazine/default.aspx (last access on Oct. 2014).

[Pie02b] Matt Pietrek. An In-Depth Look into the Win32 Portable Executable File Format, Part 2. In *MSDN Magazine* [Mic02]. http://msdn.microsoft.com/en-us/magazine/cc301808.aspx (last access on Oct. 2014).

[Rad94] Yisrael Radai. Integrity Checking for Anti-Viral Purposes; Theory and Practice. `https://www.virusbtn.com/files/old_papers/YisraelRadai-Integrity.pdf` (last access on Oct. 2014), 1994.

[Rev11] ReversingLabs Corporation. Undocumented PECOFF. In *BlackHat USA 2011, Las Vegas* [Bla11]. `http://www.reversinglabs.com/sites/default/files/pictures/PECOFF_BlackHat-USA-11-Whitepaper.pdf` (last access on Oct. 2014).

[Sec08] Secure Systems. Revealing Packed Malware, 2008.

[SH12] Michael Sikorski and Andrew Honig. *Practical Malware Analysis*. No Starch Press, Inc., 2012.

[Sha48] C. E. Shannon. A Mathematical Theory of Communication. In *The Bell System Technical Journal*, volume 27, pages 379–423. American Telephone and Telegraph Co., July 1948.

[SHRS07] Casey Sheehan, Nick Hnatiw, Tom Robinson, and Nick Suan. Pimp My PE: Parsing Malicious and Malformed Executables. In *Virus Bulletin Conference 2007*. Sunbelt Software, 2007. `https://www.virusbtn.com/index` (last access on Oct. 2014).

[SK06] Chilin Shih and Greg Kochanski. Bayes' Theorem. `http://kochanski.org/gpk/teaching/0401Oxford/Bayes.pdf` (last access on Oct. 2014), September 2006.

[Sot] Alexander Sotirov. Tiny PE. `http://www.phreedom.org/research/tinype` (last access on Oct. 2014).

[Sun99] Sun Microsystems. Code Conventions for the Java ™ Programming Language. `http://www.oracle.com/technetwork/java/codeconvtoc-136057.html` (last access on Oct. 2014), April 1999.

[Sym14] Symantec Corporation. Trojan.Dropper. `http://www.symantec.com/security_response/writeup.jsp?docid=2002-082718-3007-99` (last access on Oct. 2014), 1995–2014.

[Szo05] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley Professional, February 2005.

[VP11] Mario Vuksan and Tomislav Pericin. Constant Insecurity: Things You Didn't Know About (PECOFF) Portable Executable Format. In *BlackHat USA 2011, Las Vegas* [Bla11]. `https://www.blackhat.com/` (last access on Oct. 2014).

List of Figures

List of Tables

List of Listings

List of Acronyms

DLL	dynamic-link library
EXE	executable file
IAT	Import Address Table
JVM	Java Virtual Machine
NE	New Executable
PE	Portable Executable
PEB	process environment block
PE/COFF specification	Microsoft Portable Executable and Common Object File Format Specification
PE32+	Portable Executable with a 64-bit address space
PE32	Portable Executable with a 32-bit address space
RAT	remote administration tool
RVA	relative virtual address
SBT	Simple Build Tool
TLS	thread local storage
VA	virtual address
VM	virtual machine

Appendices

Appendix A

Anomaly Detection Test Files and Results

These are the test files used for anomaly tests in **??**.

Test Files for Anomaly Tests

Name	**Testfile**
TinyPE	tiny.import.133/tiny.exe [Sot]
Corkamix	corkami/corkamix.exe [Alb13]
max_secW7.exe	corkami/max_secW7.exe [Alb13]
imagebase_null.exe	corkami/imagebase_null.exe [Alb13]
Win32.Sality	#c0405fc5e28278bfeb23610fd8e3e671
Trojan.Spy.Zeus	#b448b67aef297c16af6964d14950d61e
W32.Simile	#ed1d33ce9ed4be9a8c2f8077b95b9c30
W32.IrcBot	#86be9017e24a8fa3a21bf8f5a073afde
W32.Hybris.Worm	#f37aba2360cb62551ceb7bad098b83a1
W32.FavilageJ	#8da90f9255f575036ece38d90cac4e6a

Table A.1: Anomaly detection comparison

Testfile	**PortEx**	**pefile**	**PEStudio**	**pev**
Number of detected anomalies				
TinyPE	15	1	25	error
Corkamix	10	3	14	0
max_secW7.exe	204	0	29	3
imagebase_null.exe	13	0	15	7
W32.Sality	12	3	19	6
Zeus Trojan	5	1	16	4
Simile	6	0	19	2
W32.IrcBot	16	2	10	error

Appendix B

Test Files for Robustness Comparison

These are the test files used for the robustness comparison in section 5.2. Filename and reference are given for clean files. Malicous test files are from VirusShare[1] and defined by their hash value.

Test Files for Robustness Tests

Malformation	**Testfile**
No data directory	tiny.168/tiny.exe [Sot]
Collapsed Optional Header	tiny.128/tiny.exe [Sot]
196 sections	corkami/max_secW7.exe [Alb13]
65535 sections	corkami/65535sects.exe [Alb13]
Virtual Section Table	corkami/virtsectblXP.exe [Alb13]
Fractionated imports	#05e261d74d06dd8d35583614def3f22e
Collapsed IAT	tiny.import.133/tiny.exe [Sot]
Virtual first import descriptor	corkami/imports_virtdesc.exe [Alb13]
Resource loop	corkami/resource_loop.exe [Alb13]
Resources shuffled	corkami/resource_shuffled.exe [Alb13]
Fractionated resources	#7dfcbb865a4a5637efd97a2d021eb4b3
Nothing DLL	corkami/nothing.dll [Alb13]

Continued on next page

[1] http://virusshare.com/

Table B.1 – *Continued from previous page*

Malformation	**Testfile**
Exports shuffled	corkami/exports_order.exe [Alb13]
Virtual relocations	corkami/virtrelocXP.exe [Alb13]
Sectionless PE	corkami/sectionless.exe [Alb13]
All sections invalid	#d4a3a413257e49d81962e3d7ec0944eb

Appendix C

Report Example

Listing C.1: Text report by PortEx

```
Report For whole_pe_section.exe
*******************************

file size 0x2000
full path /home/deque/portextestfiles/unusualfiles/corkami/whole_pe_section.exe


MSDOS Header
************

description                             value          file offset
-------------------------------------------------------------------------
signature word                          0x5a4d         0x0
last page size                          0x90           0x2
file pages                              0x3            0x4
relocation items                        0x0            0x6
header paragraphs                       0x4            0x8
minimum number of paragraphs allocated  0x0            0xa
maximum number of paragraphs allocated  0xffff         0xc
initial SS value                        0x0            0xe
initial SP value                        0xb8           0x10
complemented checksum                   0x0            0x12
initial IP value                        0x0            0x14
pre-relocated initial CS value          0x0            0x16
relocation table offset                 0x40           0x18
overlay number                          0x0            0x1a
Reserved word 0x1c                      0x0            0x1c
Reserved word 0x1e                      0x0            0x1e
Reserved word 0x20                      0x0            0x20
Reserved word 0x22                      0x0            0x22
OEM identifier                          0x0            0x24
OEM information                         0x0            0x26
Reserved word 0x28                      0x0            0x28
Reserved word 0x2a                      0x0            0x2a
Reserved word 0x2c                      0x0            0x2c
Reserved word 0x2f                      0x0            0x2e
Reserved word 0x30                      0x0            0x30
Reserved word 0x32                      0x0            0x32
Reserved word 0x34                      0x0            0x34
Reserved word 0x36                      0x0            0x36
Reserved word 0x38                      0x0            0x38
Reserved word 0x3a                      0x0            0x3a
PE signature offset                     0x80           0x3c


```

```
COFF File Header
****************

time date stamp  Jan 1, 1970 1:00:00 AM
machine type     Intel 386 or later processors and compatible processors
characteristics  * Image only, Windows CE, and Windows NT and later.
                 * Machine is based on a 32-bit-word architecture.
                 * Image only.
                 * COFF line numbers have been removed. DEPRECATED
                 * COFF symbol table entries for local symbols have been removed.
                     DEPRECATED

description                          value          file offset
-------------------------------------------------------------------
machine type                         0x14c          0x84
number of sections                   0x2            0x86
time date stamp                      0x0            0x88
pointer to symbol table (deprecated) 0x0            0x8c
number of symbols (deprecated)       0x0            0x90
size of optional header              0xe0           0x94
characteristics                      0x10f          0x96


Optional Header
***************

standard field                       value            file offset
-----------------------------------------------------------------------
magic number                         0x10b            0x98
major linker version                 0x0              0x9a
minor linker version                 0x0              0x9b
size of code                         0x3b             0x9c
size of initialized data             0x3ff04e         0xa0
size of unitialized data             0x0              0xa4
address of entry point               0x1000           0xa8
address of base of code              0x1000           0xac
address of base of data              0x2000           0xb0

windows field                        value            file offset
-----------------------------------------------------------------------
image base                           0x400000         0xb4
section alignment in bytes           0x1000           0xb8
file alignment in bytes              0x1000           0xbc
major operating system version       0x4              0xc0
minor operating system version       0x0              0xc2
major image version                  0x0              0xc4
minor image version                  0x0              0xc6
major subsystem version              0x4              0xc8
minor subsystem version              0x0              0xca
win32 version value (reserved)       0x0              0xcc
size of image in bytes               0x4000           0xd0
size of headers                      0x1000           0xd4
checksum                             0x0              0xd8
subsystem                            0x2              0xdc
dll characteristics                  0x0              0xde
size of stack reserve                0x100000         0xe0
size of stack commit                 0x1000           0xe4
size of heap reserve                 0x100000         0xe8
size of heap commit                  0x1000           0xec
loader flags (reserved)              0x0              0xf0
number of rva and sizes              0x10             0xf4

data directory          virtual address  size             file offset
------------------------------------------------------------------------
import table            0x105e           0x80             0x100
```

```
Section Table
*************
                         1. .text        2. whole
-------------------------------------------------------
Entropy                  0.04            0.05
Pointer To Raw Data      0x1000          0x1
-> aligned (act. start)                  0x0
Size Of Raw Data         0x1000          0x1fff
-> actual read size                      0x2000
Physical End             0x2000          0x2000
Virtual Address          0x1000          0x2000
Virtual Size             0x1000          0x2000
Pointer To Relocations   0x0             0x0
Number Of Relocations    0x0             0x0
Pointer To Line Numbers  0x0             0x0
Number Of Line Numbers   0x0             0x0
Code                     x               x
Initialized Data         x               x
Uninitialized Data       x
Execute                  x               x
Write                    x               x


Imports
*******

kernel32.dll
rva: 4250 (0x109a), name: ExitProcess, hint: 0

user32.dll
rva: 4258 (0x10a2), name: MessageBoxA, hint: 0


Anomalies
*********

* Deprecated Characteristic in COFF File Header: IMAGE_FILE_LINE_NUMS_STRIPPED
* Deprecated Characteristic in COFF File Header: IMAGE_FILE_LOCAL_SYMS_STRIPPED
* Optional Header: Default File Alignment is 512, but actual value is 4096
* Section Header 2 with name whole: SIZE_OF_RAW_DATA (8191) must be a multiple of
    File Alignment (4096)
* Section Header 2 with name whole: POINTER_TO_RAW_DATA (1) must be a multiple of
    File Alignment (4096)
* Section name is unusual: whole
* Physically shuffled sections: section 1 has range 4096--8192, section 2 has
    range 0--8192
* Section 1 with name .text (range: 4096--8192) physically overlaps with section
    whole with number 2 (range: 0--8192)
* Entry point is in writeable section 1 with name .text
* Section Header 1 with name .text has unusual characteristics, that shouldn't be
    there: Initialized Data, Uninitialized Data, Write


Hashes
******

MD5:    e2cd26e0c4296ab7ff11a5c0df4a41a4
SHA256: cd124ee7648ac00fdd5f82e015146c79075591cf8d6f810134de77bc5dc3d1c3

Section      Type      Hash Value
-------------------------------------------------------------------------------
1. .text     MD5       94cde06f47a710774c075900e290caf6
             SHA256
                 8740f0b568752f308d193206c9016c7549c6fccc0fae5c1b3fb3c4445ff6c729
2. whole     MD5       e2cd26e0c4296ab7ff11a5c0df4a41a4
             SHA256
                 cd124ee7648ac00fdd5f82e015146c79075591cf8d6f810134de77bc5dc3d1c3
```

Appendix D

Anomalies Recognised by PortEx

Anomalies Recognised by PortEx

Structure	Anomalies
MSDOS Stub	• collapsed MSDOS Header
COFF File Header	• PE File Header in overlay (see page 50) • `SizeOfOptionalHeader`: too large, too small, collapsed Optional Header • `NumberOfSections`: too many (see page 46), sectionless (see page 49) • deprecated: `NumberOfSymbols`, `PointerToSymbolTable`, file characteristics • reserved file characteristics
Optional Header	• `ImageBase`: check for too large, zero or non-default, must be multiple of 64 K • `SizeOfImage`: must be multiple of `SectionAlignment` • unusual number of data directory entries or no data directory • reserved or invalid data directory entry • `SizeOfHeaders`: min value (see *Dual PE File Header*, page 52), exact value, must be multiple of `FileAlignment`

Continued on next page

Table D.1 – *Continued from previous page*

Structure	**Anomalies**
	• reserved or deprecated: DLL characteristics, `Win32Version` (see page 46), loader flags • `FileAlignment`: must be power of two, between 512 and 65 536; check for non-default value • `SectionAlignment`: must be greater than `FileAlignment` • low alignment mode (see [Alb13]) • `AddressOfEntryPoint`: must be greater than or equal to `SizeOfHeaders`, must be non-zero in EXE file (see page 45), virtual entry point detection, entry point in last section suspicious
Section Table	• unusual section names, control characters in section names • Section Table in overlay (see page 50) • `SizeOfRawData` larger than file size permits (see page 44) • section characteristics: extended reloc, reserved, deprecated, unusual for purpose of section • entry point in writeable section • physically or virtually overlapping, duplicated, and shuffled sections • ascending `VirtualAddress` values of sections • deprecated fields: `PointerOfLineNumbers`, `NumberOfLineNumbers` • zero values: `VirtualSize`, `SizeOfRawData`, object only characteristics, `PointerToReloc`, `NumberOfReloc` • uninitialised data constraints • `FileAlignment` constraints for: `SizeOfRawData`, `PointerToRawData`

Statement of Authorship

I declare on oath that I completed this work on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this, nor a similar work, has been published or presented to an examination committee.

Leipzig, _____ _____

Signature