BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

EXTRACTION OF STATIC FEATURES FROM BINARY APPLICATIONS FOR MALWARE ANALYSIS

EXTRAKCIA STATICKÝCH RYSOV Z BINÁRNYCH APLIKACII ZA ÚČELOM ANALÝZY MALWARU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **JAKUB PRUŽINEC**
AUTOR PRÁCE

SUPERVISOR — **Doc. Dr. Ing. DUŠAN KOLÁŘ**
VEDOUCÍ PRÁCE

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2018/2019

Bachelor's Thesis Specification

22056

Student: **Pružinec Jakub**
Programme: Information Technology
Title: **Extraction of Static Features from Binary Applications for Malware Analysis**
Category: Security
Assignment:

1. Study binary-file analysis. Focus on analysis of executable files.
2. Get acquainted with tool Fileinfo and study analyses/heuristics implemented in it.
3. Study systems developed by Avast that use output of Fileinfo for malware analysis (e.g. RetDec, Clusty).
4. Analyze which features are used in each system and evaluate limitations of the current solution.
5. Design analyses and heuristics that will provide new static features to individual systems (or improve the existing ones) for the purpose of malware-analysis enhancement in tool Fileinfo.
6. Implement the algorithms designed in the previous step after a discussion with the supervisor and consultant.
7. Thoroughly verify the implemented solution by creating a suite of tests, including real tests and real malware samples.
8. Evaluate your work and discuss future development possibilities.

Recommended literature:

- Sikorski, Michael, and Andrew Honig. Practical Malware Analysis: A Hands-On Guide to Dissecting Malicious Software. San Francisco: No Starch Press, 2012. Print.
- RetDec Project documentation. https://github.com/avast-tl/retdec/
- Additional resources recommended by supervisor and consultant.

Requirements for the first semester:

- The first five points from the assignment and part of the sixth point.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor: **Kolář Dušan, doc. Dr. Ing.**
Consultant: Milkovič Marek, Ing., Avast
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2018
Submission deadline: May 15, 2019
Approval date: October 24, 2018

Abstract

Forms of malware are changing and evolving on daily basis, therefore it is necessary to continuously create, update, and improve methods for malware analysis. One of possible approaches to fighting malware is to classify it based on certain static characteristics. This thesis deals with design and extraction of these features from binary executables. Goal of this work is to enrich a static feature extraction tool by extracting new features and verifying their effectiveness in malware classification. The tool is developed in cooperation with Avast Software, where it is used in a clustering system.

Abstrakt

Podoby škodlivého software sa deň čo deň menia a vyvíjajú. Vzniká tak nutnosť jednostaj tvoriť, aktualizovať a zlepšovať metódy na analýzu škodlivého software. Jedným z možných prístupov ako bojovať proti škodlivému software je klasifikovať ho na základe určitých statických charakteristík. Táto práca sa zaoberá návrhom a extrakciou týchto čŕt z binárnych spustiteľných súborov. Cieľom tejto práce je obohatiť nástroj na extrakciu statických rysov o extrakciu nových rysov a overenie ich účinnosti pri klasifikácii škodlivého software. Nástroj je vyvíjaný v spolupráci so spoločnosťou Avast, kde sa používa v systéme zhlukovej analýze.

Keywords

Reverse engineering, malware, static analysis, Avast

Klíčová slova

Reverzné inžinierstvo, malware, statická analýza, Avast

Reference

PRUŽINEC, Jakub. *Extraction of Static Features from Binary Applications for Malware Analysis.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Dr. Ing. Dušan Kolář

Rozšířený abstrakt

Práca sa zaoberá statickou analýzou škodlivého software, malware. Cieľom práce je navrhnúť rysy binárnych spustiteľných súborov použiteľných pri statickej analýze malware. Navrhnuté rysy sú ďalej implementované v rámci nástroja FileInfo firmy Avast určeného na statickú extrakciu rysov. Výstupy z nástroja FileInfo sú používané v spätnom prekladači RetDec and internom zhlukovom systéme firmy Avast, ktorý sa volá Clusty.

Navrhnuté a implementované rysy sú ďalej testované sadou regresných testov s reálnymi vzorkami malware. Rysy vhodné na zhlukovú analýzu sú integrované do nástroja Clusty a sú taktiež testované na skutočných vzorkách škodlivého software.

Práca kladie dôraz na analýzu binárnych súborov formátu Portable Executable (PE) platformy Windows. Mechanizmus importovania externých symbolov .NET frameworku je analyzovaný. Tabuľka TypeRef je zrekonštruovaná a jej rysy sú produkované. Takto extrahovaná tabuľka poskytuje informácie o externých triedach a to ich meno, menný priestor a pôvod. Rodičovský a synovský vsťah vnorených tried je vyjadrený odkazom jednej položky tabuľky TypeRef na druhú. Pri rekonštrukcii sa vnorené triedy spajajú tak, že synovské instancie položiek TypeRef obsahujú ukazateľ na otcovské instancie. Cykly v orientovanom grafe odkazov položiek TypeRef sú detekované a rozpojené. Zrekonštruované meno synovskej položky je spojené s menami otcovských položiek. MD5 hash tabuľky TypeRef je integrovaný do produkčnej verzie nástroja Clusty a dosahuje výnimočne dobré výsledky. Zo vzoriek malware, ktoré sú odchytené firmou Avast v priebehu jedného mesiaca vznikajú zhluky, ktoré obsahujú viac a 2.3 miliónov škodlivých súborov s presnosťou 99%.

Visual Basic aplikácie sú podrobené dôkladnej statickej analýze a ich metadáta sú extrahované. Metadáta sú uložené v rozsiahlej hierarchii štruktúr, ktoré sú spracovávané metodou top-down parsing. Parsovanie je robustné a dáta, ktoré boli poškodené v jednej štruktúre, ale sú duplicitne obsiahnuté v inej štruktúre sú extrahované. Metadáta obsahujú informácie o projekte, ako je napríklad pôvodný názov spustiteľného súboru, systémova cesta k projektu a používané jazyky identifikované číslom LCID. Po extrakcii je číslo LCID preložené do znakovej reprezentácie jazyka a dialektu. Použitie P-code bajtkódu namiesto natívnych inštrukcii je detekované. Visual Basic metadáta obsahujú aj štruktúry popisujúce triedy definované v rámci projektu. Tieto štruktúry sa preložia do podoby takzvanej tabuľky Objektov, kde každá položka odpovedá jednej definovanej Visual Basic triede. Trieda obsahuje meno a názvy metód, ktoré daná trieda implementuje. Z tabuľky objektov sa produkú kryptografické hashe. Okrem tabuľky Objektov obsahuje hierarchia metadát aj štruktúry popisujúce externé funkcie, ktoré sa importujú v dobe behu programu. Tie sú zase reprezentované tabuľkou External. Položky tabuľky External obsahujú názvy externých funcii a názvy modulov, z ktorých funkie pochádzajú. Po rekonštrukcii tabuľky External su vypočítané jej kryptografické hashe. Globálne unikátne identifikátory projektu GUID a CLSID sú extrahované a manuálne testované na perzistentnosť po opakovanom preklade. Visual Basic aplikácia je preložená prvýkrát a jej unikátne identifikátory sú zaznamenané. Pri druhom preklade sa porovnajú aktuálne identifikátory so zaznamenanými. Ak sa identifikátor nezmení, je perzistentný. Identifikátory TypeLib CLSID a COM Object GUID sa prekladom nemenia a boli použité pri zhlukovaní. Viac ako 11.000 vzoriek malware je rozdelených do zhlukov na základe MD5 hashu tabuľky Objektov, MD5 hashu tabuľky External, COM Object GUID a TypeLib CLSID. Štatistiky o početnosti COM informácii viac ako 11.000 vzoriek Visual Basic aplikacii sú produkované.

Ďalším významným okruhom, ktorým sa táto práca zaoberá sú ikony zabudované do programov. Windows 10 dektop environment je manuálne analyzovaný, keďže voľba hlavnej ikony, teda ikony ktorá je zobrazená na ploche, nie je deterministická. Do spustiteľného

súboru je zabudovné množstvo ikon rôznych štandardných rozmerov a bitových hĺbok odlíšených farbou. Farba zobrazenej ikony prezrádza rozmery a bitovú hĺbku najprioritnejšej ikony. Táto ikona je zaznamenaná a odstránená zo súbru. Preces sa opakuje dokým sú v súbore ikony. Takto zostrojený zoznam priorít ikon je použitý na výber hlavnej ikony, ktorá je ďalej spracovávaná. Kryptografické hashe hlavnej ikony sú produkované. Kvôli častnej ale nepatrnej modifikácii ikon nie sú kryptografické hashe najvhodnejšie na zhlukovú analýzu. Z tohoto dôvodu je hlavná ikona prevedená do internej reprezentácie a vypočíta sa jej perceptuálny hash, teda hash ktorý sa miernou zmenou dát zmení len nepatrne. Prevod ikony do internej reprezentácie je realizovaný pomocou parsovania popisných štruktúr ikon, takzvaných Icon resourcov a parsovania samotných dát obrázku uložených vo formáťe DIB. Výpočet perceptuálneho hashu zvaného average hash zahŕňa operácie s obrázkom ako napríklad zmenšenie a prevod na čiernobielu. Ikony, ktorých average hashe majú Hammingovu vzdialenosť menšiu ako 3 sú považované za podobné. Dataset s 11.000 vzorkami programov s ikonami je rozdelený do zhlukov na základe kryptografického MD5 hashu a perceptuálneho average hashu hlavnej ikony. Ikony sú testované na striktnú zhodu average hashov napriek tomu, že metrika podobnosti ikon je definovaná benevolentnejšie. Ukazuje sa, že aj v prípade testu na exaktnú zhodu perceptuálneho hashu dáva average hash lepšie výsledky ako MD5 hash. Štatistiky o vlastnostiach ikon extrahovaných z hlavicky DIB viac ako 50.000 vzoriek malware sú produkované.

Práca obsahuje pomocné rysy určené primárne pre analytikov. Binárne súbory sú skenované a množstvo navrhnutých anomálii týkajúcich sa formátu PE je zaznamenaných. Anomálie zahŕňajú porušený formát sekcii, nezvyčajné alebo duplicitné mená sekcii, známe mená sekcii packerov, nezvyčajné lokácie vstupného bodu programu a anomálie týkajúce sa rozsahu a prekryvu sekcii, importov, exportov a resourcov. Ďalej sa práca zaoberá detekciou kompresie dát. Entropia sekcii formátov PE, ELF, Mach-O a COFF je vypočítaná za účelom detekcie užitia packeru. Entropia dát pridaných za koniec súboru je vypočítaná ako odhad ich informačnej hodnoty.

Neskôr je v práci popísaný spôsob získavania pomocných informácii, ktoré niektoré prekladače vkladajú do spustiteľných súborov ako pomôcku pre užívateľov a iné služby. Tieto informácie sú uložené v resource zvanom VersionInfo. VersionInfo je reprezentovaný v PE súbore ako množina vnorených štruktúr TLV definovaných typom, dĺžkou a hodnotou. Top-down parser VersionInfo dát extrahuje reťazce popisujúce produkt a zoznam jazykov podporovaných aplikáciou. Reťazce môžu obsahovať informácie o autorských právach, názve produktu a čase vzniku produktu. Položky v zozname podporovaných jazykov sa skladajú z identifikátoru jazyka LCID a identifikátoru kódovania IBM CPID. Oba identifikátory sú preložené do ich reťazcovej reprezentácie.

Napokon sa práca zaoberá extrakciou informácii o thread-local storage. Statické dáta lokálne pre dané vlákno sú uložené v .TLS sekcii. Tieto premenné su inicializované pomocou inicializačných rutín prezývaných TLS callbacks. Z adresára thread-local storage sú extrahované adresy callbackov, rozmedzie statických dát a iné pomocné informácie.

Navrhnuté rysy sú implementované v jazyku C++. Práca popisuje objektový návrh FileInfa ako aj novo navrhnuté objekty a algoritmy na realizáciu extrakcie uvedených rysov. FileInfo extrahuje všetky dostupné informácie o binárnom súbore a prezentuje ich formou čitateľného textu, alebo vo formáte JSON. Regresné testy FileInfa sú implementované v rámci sady regresných testov firmy Avast.

Práca bola predvedená na študentskej konferencii Excel@FIT.

Extraction of Static Features from Binary Applications for Malware Analysis

Declaration

I hereby declare that this bachelor's thesis was prepared as an original author's work under the supervision of Doc. Dr. Ing. Dušan Kolář. The supplementary information was provided by Ing. Marek Milkovič and Ing. Jakub Křoustek, PhD. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Jakub Pružinec
May 16, 2019

Acknowledgements

I would like to express special thanks to my supervisor Doc. Dr. Ing. Dušan Kolář and consultant Ing. Milkovič for willingness and help throughout the work. I would like to thank Ing. Jakub Křoustek for advice and support and Ing. Petr Zemek for assistance.

Contents

1 Introduction **3**

2 Static Malware Analysis **4**
2.1 Information Extraction 4
2.2 Information Relevant to Malware Analysis 5
2.3 Reverse Engineering Tools 6

3 PE File Format **7**
3.1 Data Directories and Sections 8
3.2 Icons 10
3.3 VersionInfo Resource 13
3.4 .NET 14
3.5 Visual Basic 15

4 Malware Analysis Tools Developed by Avast Software **18**
4.1 Clusty 18
4.2 RetDec 18
4.3 FileInfo 19

5 Design and Extraction of Malware Features **20**
5.1 .NET Features 20
5.2 Icon Features 21
5.3 Visual Basic Features 24
5.4 Detection Features 26

6 Implementation of Extraction of Designed Malware Features **29**
6.1 FileInfo Infrastructure 29
6.2 Feature Extraction 31
6.3 Source Codes, Compilation, and Execution 39

7 Testing and Results **40**
7.1 Regression Tests 40
7.2 Statistics 41
7.3 Integration, Deployment and Feature Efficiency Evaluation 42

8 Conclusion **44**

Bibliography **46**

A Format of Binary Data Structures **49**
A.1 Icon and IconGroup Resources 49
A.2 .NET Values 50
A.3 Visual Basic Structures 50
A.4 Thread-local Storage Directory 52

B Statistical Analysis of Malware Icons **53**

Chapter 1

Introduction

Overall userbase of information technologies is being diversified due to a rapid and therefore widespread adoption of information technologies and modern world digitalization. Use of information technologies among laymen is now more common than any time before. Despite it being a necessity, laymen often are not even remotely interested in protecting their data. Furthermore, value of their data is constantly growing and it is in great interest of certain people to manipulate it without permission. User's lack of awareness and their heavy use of information technologies are perfect conditions for spreading *malicious software*, *malware*.

The fight against malware has been around for a while now. Numerous methods to identify and remove malware have been developed. Today, as malware grows larger than any time before, manual analysis of every malware sample is impossible. Due to this, anti-malware companies are developing systems for automated malware clustering and classification as an absolute necessity to fight today's malware. These systems detect similarities between analyzed programs and group them together, so that whole groups can be classified as malicious. During this process various features are extracted from samples. Sample grouping is performed based on similarity of extracted features.

Goal of this work is to design features extractable from binary applications that are suitable for malware analysis. Special attention is paid to PE file format components and its extensions such as Visual Basic and .NET. All features extracted in this work are obtained statically, meaning that malware samples are not executed during extraction process. A feature extraction tool, FileInfo, is expanded by extraction of new features. FileInfo is used inside a complex malware clustering and classification system developed by Avast Software. Efficiency of clustering and classification based on most promising extracted features is tested.

Chapter 2 describes static analysis methods and tools in general. In Chapter 3, most relevant parts of PE file format are described. Tools developed by Avast Software are depicted in Chapter 4. Further, Chapter 5 contains all designed features. Implementation, testing, and results of designed features can be found in Chapters 6 and Chapter 7. Contributions of this work are summarized in Chapter 8.

Chapter 2

Static Malware Analysis

Static malware analysis is a technique of malware examination without program execution. Its major purpose is to identify malware and estimate its behavior before performing a dynamic analysis. This is usually done by dissecting malware fragments and investigating their semantics. Such fragments are various data structures, tables, machine code understood as a sequence of instructions, or other components of executable files. In this work static analysis is used to produce valuable features that can be applied in identification of malware and therefore its classification.

2.1 Information Extraction

A *parser* is a software component that processes input data and produces its internal representation. Such representation might be a *parse tree* or an *abstract syntax tree*. Input for parsers may differ. Most of parser inputs are programming languages, texts, or binary streams. Here, binary streams in the form of executable files are parsed. Parsers presented here are mostly designed to be robust and to extract as much information as possible. In the process of parsing executable files, data is extracted rather than parsed to parse trees or abstract syntax trees.

A lot of data in executable files is optional such as debug information or compilation relicts. Presence of this information is not needed for proper program execution, therefore it is often *stripped*. Binary stripping is a process of removing information unnecessary for execution. Debug information is valuable when inspecting and analyzing security of binaries, especially in decompilation. Debug information of commercial off-the-shelf binaries is often stripped for size reduction reasons. Vulnerable and malicious binaries are often intentionally stripped to resist security analysis [17].

Moreover, it is a common practice for malware authors to corrupt binaries in a way that they remain executable but violate their file format. Corruption is done in order to deceive or evade static analysis tools.

File format of an executable file has to be known in order to properly parse it. These formats are usually described in documentation. Sometimes, the file formats are proprietary or partially closed source. Structure of such files is unknown. This is where *reverse engineering* comes into play. Reverse engineering is a process of analyzing a system to identify its components, their meaning, and possibly their reconstruction to a higher level abstraction form [14].

2.2 Information Relevant to Malware Analysis

Not all information is relevant to malware analysts. Only information that determines or at least depicts behavior of an analyzed malware is valuable. It is no surprise that malware authors try to remove or minimize this information as much as possible. Some of the most popular information malware analysts look for is presented hereafter.

Strings are sequences of printable characters of a certain length. Strings can be represented in various encodings such as ASCII or UTF16 and their length can be either explicit or implied by a terminating symbol `'\0'`. String extraction tools usually scan whole files for sequence of printable characters of a length greater than a set threshold and dump them. It is up to the malware analyst to distinguish random garbage from valid strings. Compilers frequently add additional string information or leave string relicts in compiled binaries. Not only can strings provide human readable information about program behavior, they may be used for detection of used compilers as well.

Imported symbols are used to estimate malware intentions. External libraries often implement documented API functions, therefore their use can paint a relatively detailed picture about malware functionality. Imports are so valuable that it is reasonable to investigate other importing mechanisms besides standard system. Framework specific importing systems are an example of such mechanisms. Analogically, *Exported symbols* can help identify application a malware is targeting.

Sections divide program data (and code) into fragments based on their semantics. Sections may be named and have privileges, such as privilege for execution, reading, and writing. Malformation of sections is a common practice among malware authors in order to confuse analysis tools or achieve nonstandard functionality. Investigation of section names, privileges, and ranges is done to detect practices malware authors use.

Entry point is an address pointing to where execution of a program starts from the programmers point of view. Entry point itself is not that useful without context. Location of entry point outside of mapped sections or in a section with write permission may be an indication of malicious behavior. Writable and executable section is suspicious as is, but it is almost certain that program author does not wish his/her code to be examined if the entry point is located in such a section.

Embedded resources such as icons, bitmap images, cursors, strings, and many others are contained within the file for it to be self sufficient. These resources, more specifically images may be unique to applications of a certain "batch". Applications rarely change their icons or logos in new releases. Sometimes its reasonable to assume relation between programs based on use of same (or similar) resources.

Debug information is additional program data provided by the compiler for debugging tools to associate binary code fragments with parts of source codes. Debug symbols allow analysts to gain access to information such as identifiers of variables that are otherwise omitted during compilation. Malware programs hardly ever contain debug information, but in case they are provided, they speed up analysis process rapidly.

Project identifiers are globally unique identifiers defining a relation between executables and the original project. These identifiers are added by some IDE compilers such as Visual Studio. GUIDs and CLSIDs are project identifiers defined and used by Microsoft.

2.3 Reverse Engineering Tools

Malware analysis often includes reverse engineering of executable code. *Disassemblers* and *decompilers* might serve for this task.

Disassembler is a program designed to translate machine code to assembly language. Machine code is just a sequence of binary digits, practically unreadable by a human. Because of this, translation to assembly language is an inevitable part of executable code reverse engineering. Distinguishing code from data is an undecidable problem, thus, disassemblers implement more sophisticated mechanisms to detect code sequences of binary files of a given format. Disassemblers can be used on their own, but they often come as a part of more complex tools. Listing 2.1 shows how a simple "Hello World!" program can be disassembled.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000001139 <+0>: push rbp
   0x000000000000113a <+1>: mov rbp,rsp
   0x000000000000113d <+4>: lea rdi,[rip+0xec0] # 0x2004
   0x0000000000001144 <+11>: call 0x1030 <puts@plt>
   0x0000000000001149 <+16>: mov eax,0x0
   0x000000000000114e <+21>: pop rbp
   0x000000000000114f <+22>: ret
End of assembler dump.
```

Listing 2.1: Disassembled Hello World program

Another static analysis method is *decompilation*. Decompilation is a process of reconstructing the source code of binary executables. Decompilers can approximate original source code of compiled executables despite significant information loss during compilation process. Decompilation is achieved by parsing machine code to an intermediate language that is further turned into higher abstraction representation based on processed context. One such decompiler, *RetDec*, is described in more detail in Section 4.2. Decompilation of the "Hello World!" program can be seen in Listing 2.2

```
// From module: /home/kubo/Data/tmp/main.c
// Address range: 0x1139 - 0x1150
// Line range: 4 - 8
int main(int argc, char ** argv) {
    // 0x1139
    puts("Hello World!");
    return 0;
}
```

Listing 2.2: Hello World program decompiled by RetDec decompiler

Chapter 3

PE File Format

Microsoft Windows is the platform most targeted by malware to this day [2]. Most of malware targeting this platform comes in form of *Portable Executables.* Portable Executable is a file format of binary executables designed to run on Windows operating system.

Portable Executables come in different file types from which most notable are DLL and EXE file types. Both types share the same file format; difference between these two is solely a semantic one. *Dynamic Link Libraries* are meant to export functions or data that other programs can use. They usually run only within the context of other programs. On the other hand, *Executables* run in their own process instead of being loaded into an existing process of another program [16].

The majority of information presented in this chapter was adopted from official Microsoft PE documentation [1].

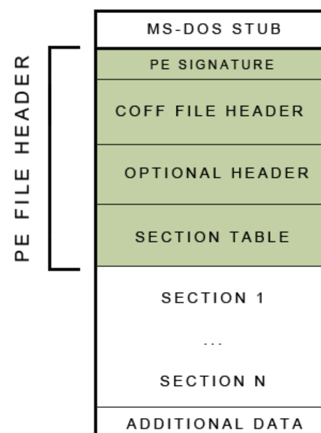

Figure 3.1: PE file format structure

Standard structure of the PE format is shown in Figure 3.1. Every PE consists of an MS-DOS stub, PE header, sections, and possibly additional data appended to file. *MS-DOS stub* is a valid application that runs under the historic MS-DOS operating system, previously developed by Microsoft. The only functionality this application has is to print `This program cannot be run in DOS mode.`

MS-DOS stub is followed by *PE header*. PE header is a structure containing data necessary for loading the program into memory and its proper execution. It consists of

a PE signature, COFF file header, Optional header and Section table. PE header offers a variety of information from which most notable is:

- PE signature
- program entry point
- number of sections
- data directories address

PE signature is a four byte sequence, `"PE\0\0"`, identifying the PE file format. *Program entry point* is an address of the first instruction to be executed. The rest of PE header information is described hereafter.

3.1 Data Directories and Sections

Semantics of data may differ and is often diverse. Therefore, programs are divided into logical groups: *sections*.

Section is a basic unit of code or data within the PE file. Sections allow the linker to link in code more selectively. Furthermore, sections bring the option to load file data to separate memory pages. Access privileges of these pages can be set based on semantics of their content. Not all sections need to be present. Relevance of a certain section is dependent on the application it is part of. Every section is defined by name, physical address, physical size, virtual addresses, and virtual size. There is not much restriction in section naming, however some sections have typical names like *.text* for a section containing code or *.rsrc* for a section containing application resources. As a result of naming freedom, malware authors, compilers and packers often violate these conventions. Due to this fact, section names are unreliable. Physical address and size define, where a given section starts and how large it is within a file. On the other hand, logical address, and size define section position relative to image base address[1] and the total size of section when loaded into process memory.

As mentioned earlier, section naming is unreliable and section positions may differ. Because of this, there needs to be some other mechanism to reliably determine the position of certain data. *Data directories* are present for this reason. Data directories contain information about file structure. They determine position of tables of imported and exported symbols, resources tables, and many others. Every data directory contains a virtual address of a table and its virtual size. Data directories overlap with sections.

3.1.1 Resource Section

PE binaries often contain embedded resources such as fonts, menus, icons, cursors, and many others. Location of resources is defined in a *Resource data directory*. Most often they are located in *Resource section*, usually called .rsrc section. Resources are structured in a multi-level sorted tree structure called *resource tree*. Despite this structure being able to incorporate 2^{31} possible levels, Windows, by convention, uses three [20]. Each level is used to describe a resource attribute. The first node determines resource type, the second determines name, and the third determines its language identifier. Every path from root to leaf node defines a resource [26].

Example in Figure 3.2 demonstrates the structure of a resource tree. The first level, *Type Directory*, has two entries in this example. Type directory entries define the type of

[1] Base address is a logical address pointing to beginning of loaded image.

a given resource. Such a type can be for example a cursor, icon, dialog box, string etc. Further, these entries point to *Name Directories*. In this example, the name directory has a single entry defining a resource name. Name is represented as a pointer to a sequence of Unicode symbols and its length. For resources of types where names are irrelevant, the name pointer is replaced by a unique number called *name id*. Additionally, data directory entries point to *Language Directories*. Language directory entries define the language of a given resource and points to *data entry*. Data entries point to raw binary data and define their length. Format of raw data depends on resource type.

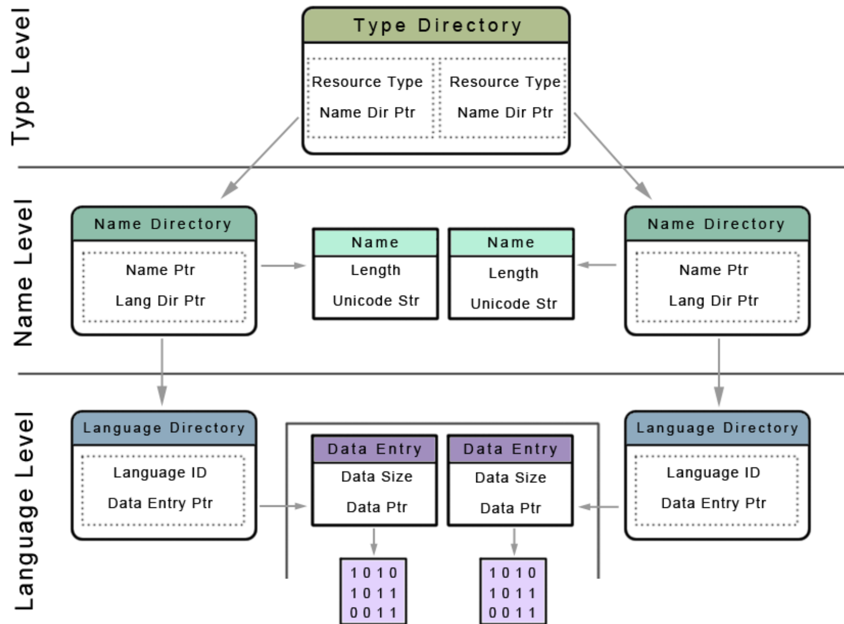

Figure 3.2: Resource tree

3.1.2 Thread-local Storage Section

Parallel execution concept introduces two major abstractions — *processes* and *threads*. Process level parallelism is achieved by maintaining process resources such as address spaces or opened file descriptors, separate. Scheduler, prevalently implemented in operating system kernel, manages execution contexts of processes. The scheduler periodically switches between currently executed and inactive processes, given that the processes cannot run simultaneously.

Thread level parallelism is a similar concept, except that threads are executed within processes and thus share most of their resources. Creation of threads is significantly more efficient than creation of processes, because only thread-local data and stacks are copied instead of the whole address space. Stacks are copied due to the necessity of storing return addresses during function calls and preventing interference of local variables.

Memory local to threads can be allocated in two ways in Windows. Dynamic allocation is done by invoking kernel with API calls, such as *TlsAlloc*, *TlsFree*, *TlsSetValue*, and *TlsGetValue*. Static allocation can be achieved by declaring static thread-local variable: `__declspec (thread) int myTlsVal = myTlsInit();`. As for regular static data, static

variables declared thread-local are contained in file, more specifically within *.tls section* referenced by *Thread-Local Storage directory*. Non-constant initialization of TLS variables (*myTlsInit*) is executed before a jump to declared entry point. As a consequence, malware sometimes implements its malicious behavior in one of thread-local data initialization routines called *TLS callbacks*. Intuitive investigation of entry point is useless in this case. What is more, many debuggers set breakpoints on entry point, therefore TLS callbacks may infect host machine before analyst has a chance to intervene.

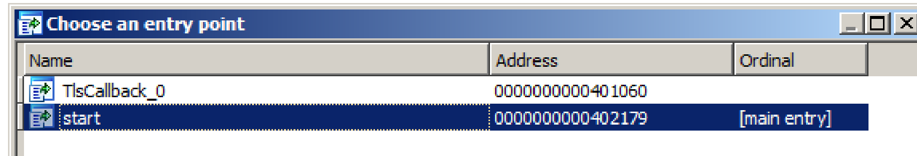

Figure 3.3: IDA Pro recognizes TLS callbacks as potential entry point

3.2 Icons

Malware authors often produce malware with icons to attract their victims. From 50.000 malware samples analyzed in this work almost 25.000 had an icon embedded as can be seen in Table 7.1. Icons, among other resources, are embedded directly in PE binaries in the Resource section. Extraction of icons is a lengthy and rigorous process.

To grasp the concept of icon embedding one has to see a bigger picture. Information presented here is only a small excerpt depicting the process. Whole topic is later discussed in more detail.

Embedding starts with creation of icons. Icons are stored in ICO file format. What is rather unconventional, is that a single ICO file can contain multiple icons. The icons themselves are stored in DIB image format. When embedding ICO files into a PE file, a group for each ICO file is created. These groups contain icons originating from same ICO file.

3.2.1 ICO File Format

ICO file format is an icon image file format developed by Microsoft. Single ICO file can contain multiple icons. Motivation for doing so is that icons can be properly scaled when having them in multiple versions in different dimension and color depths.

ICO file format is similar to the way PE files store their icons. The icon file begins with an *icon directory* followed by *icon directory entries* consecutively. Icon directory specifies the number of icon directory entries. Icon directory entries contain various icon properties such as dimensions, color depth, and offset of icon data within the file. The icons themselves can be either in DIB or PNG format [19].

3.2.2 Icon and IconGroup Resources

During compilation, icons in ICO files are spread across *icon resources*. These resources are grouped by *IconGroup resources* based on their file origin. IconGroups and Icons are separate entries in the resource tree and they are of different resource types.

IconGroup resources consist of *icon group directory* immediately followed by *icon group entries*. Icon group directory counts its entries. Entries themselves are referencing Icon resources, thus, making a relation between icons.

Format of IconGroup resources can be seen in Table A.1. *Icon width* and *icon height* are icon dimensions ranging from 0 through 255, where 0 is understood as value 256.

Bit count and *planes* describe color depth of icons. Color depth can be extracted from DIB header (described in later section) so both, bit count and planes are neglected. DIB header is described in Table A.2. *Color count* is supposed to be the number of colors in icons. Its value is expressed by the following formula:

$$colCnt = 1 \ll (bitCount + planes)$$

However, it is common practice to violate this formula and set the value to zero. Doing so can confuse Windows desktop environment and cause it to render suboptimal icon.

Icon name identifiers are used to reference icon resources. Name pointers contained within a name directory of icon resources do not point to a string as is showin in Figure 3.2. The pointers are rather used as a so called *name IDs*. Icon name identifier in an icon group entry matches with name ID of referenced icon resource. Example 3.4 demonstrates how icon resources can be grouped.

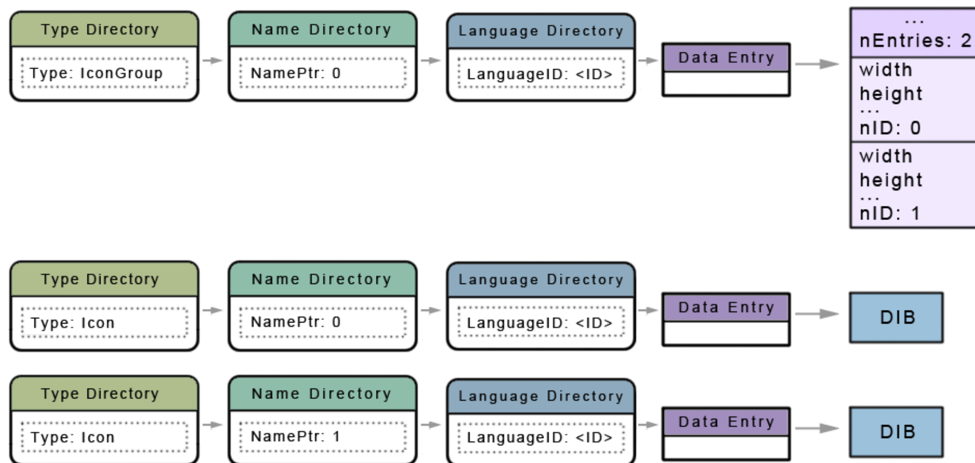

Figure 3.4: Grouped icon resources

3.2.3 DIB File Format

Since PNG format is very rarely used for icons, the only relevant format for analysis is DIB as shown in Table 7.1. *Device Independent Bitmap* is a raster image file format. It is very similar to a regular *bitmap*. In fact, the only significant difference between them is that DIB lacks bitmap header [3]. When loaded into memory, a bitmap becomes DIB. DIB format consists of three parts: *BITMAPINFOHEADER structure*, *color palette*, and *pixel array*.

BITMAPINFOHEADER structure, sometimes referred to as *DIB header*, contains information about image dimensions, color depth, compression methods, image size, and size of *color palette*. Currently supported color depths are 1, 4, 8, 16, 24, and 32 bits per pixel.

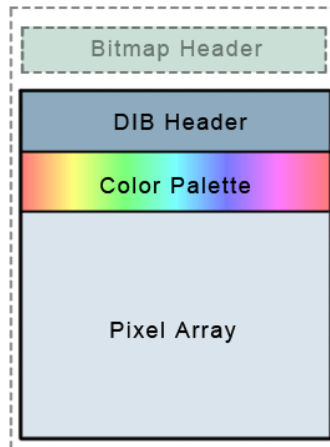

Figure 3.5: DIB format

Color palette is an array of four-byte RGBA32 pixels. Color palettes are primarily used to save memory. Single pixel for every color used in an image is stored once in the palette. All image pixels are replaced by indices to the palette. Color palette colors are formed by three color channel bytes[2] followed by a single byte alpha value. Size of color palette can be either 0 or 2^n, where n is the color depth of an image. Use of a color palette can be seen in an example in Figure 3.6.

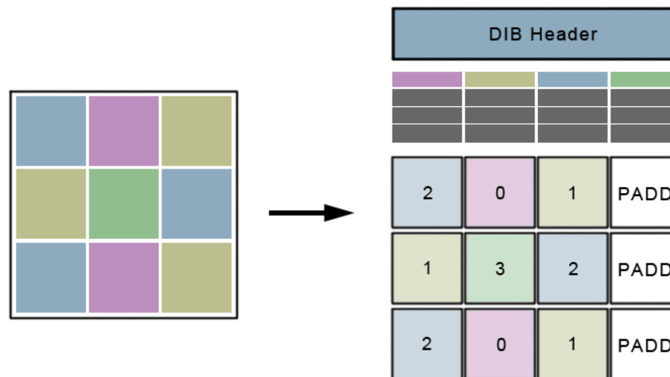

Figure 3.6: DIB icon example

Eventually, a pixel array is one dimensional array of pixels or color palette indices. Size of image rows is padded up to multiples of four bytes. Rows are stored in the pixel array consecutively. Size of a padded row can be expressed by the following formula:

$$rowsize = \left\lfloor \frac{bpp \cdot width + 31}{32} \right\rfloor \cdot 4$$

Size of pixels or indices is determined by color depth. Pixels with 1, 4, 8, or 16 bpp color depth are represented as color palette indices. Size of these indices is 1, 4, 8, or 16 bits. A pixel with 24 bpp color depth is represented as three color channel values[3]. Finally,

[2]Order of color channels is: blue, green, and red.
[3]Order of color channels is the same as in color palette

pixels with 32 bpp color depth are represented as three color channel values followed by an alpha value.

Example in Figure 3.6 demonstrates how an icon can be stored as a 4 bpp DIB. In this example, there is a 3x3 icon. Size of this color palette is $2^4 = 16$ pixels. Each pixel in a color palette is of 4 byte size. The color palette is followed by a pixel array. There are three rows in the pixel array and all of them are three pixels wide. Pixels themselves are represented as 4 bit color palette indices. Note that not every entry of a color palette needs to be referenced. Every row is padded to 4 byte alignment. In this case the size of padding in bits is:

$$\begin{aligned} p &= rowsize - bpp \cdot width \\ &= \left\lfloor \frac{bpp \cdot width + 31}{32} \right\rfloor \cdot 4 \cdot 8 - bpp \cdot width \\ &= 20 \end{aligned}$$

3.3 VersionInfo Resource

Compilers often add supplementary data unnecessary for proper execution of applications. Such information can be obtained from *VersionInfo resources*. Auxiliary VersionInfo resources are meant to provide additional product information for application users or other services.

These resources may contain various information, most notably a list of supported languages, and VersionInfo strings. According to Microsoft guidelines, the strings should specify *CompanyName*, *ProductVersion*, *LegalCopyright*, *OriginalFilename*, and other product related information [5]. Such information can be observed in application properties window as shown in Figure 3.7.

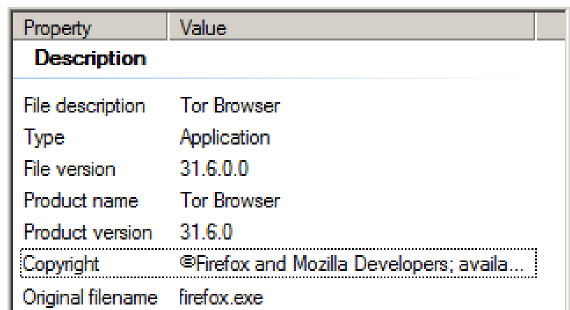

Figure 3.7: VersionInfo presented in application properties

Unfortunately, extraction of this information is not straightforward and requires parsing of multiple nested structures. Most of them are in type-length-value (TLV) format and contain string structure type identification encoded in UTF-16. The value member of the structures is either a single instance or an array of other TLV structures, further referred to as children. Hierarchy of VersionInfo structures is depicted in Figure 3.8.

VS_VERSIONINFO is the main structure and its children are an array of either one or zero *StringFileInfo* and one or zero *VarFileInfo* structures. Child of VarFileInfo structure is a structure called *Var*. Finally, child of Var structure is a list of supported languages represented as pairs of *language code identifiers* and *IBM code page numbers*. Language code identifier, abbreviated as LCID, is a standardized 16 bit value used for identification

of world languages and dialects [8]. IBM code page identifiers define string encoding, such as UTF-8, UTF-16, and others [4]. Children of StringFileInfo structures are of *StringTable* type. Identification string of StringTable is replaced with a string representation of the IBM code page identifier. This code page identifier defines encoding of its children. Children of StringFileInfo structures are *String* structures. Strings are TLV structures as well and they contain a key-value pair. Their identification string member is to be interpreted as the key and value member is the actual string value. In example in Figure 3.7 "Original filename"-"firefox.exe" and other key-value pairs can be observed.

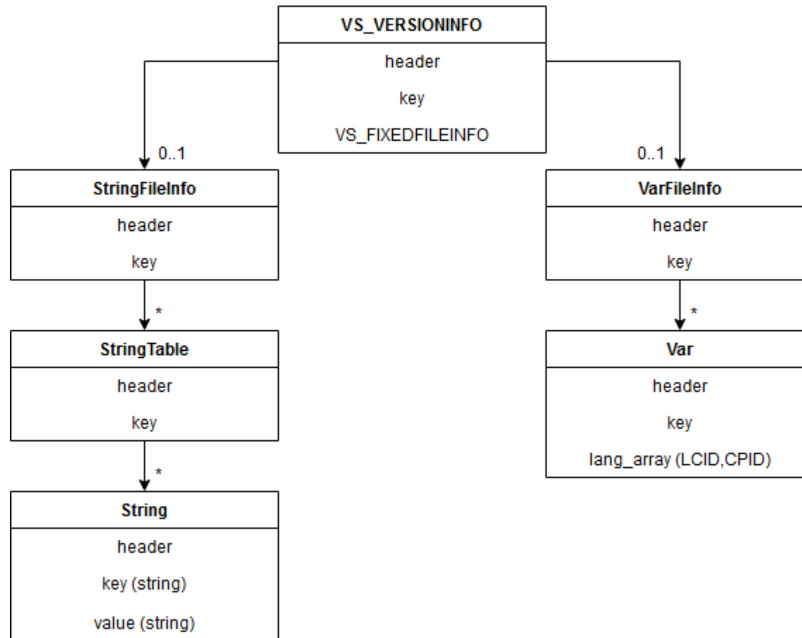

Figure 3.8: VersionInfo hierarchy

3.4 .NET

.NET framework is a popular framework developed by Microsoft in 2000. Programs written in .NET framework are compiled into a *Common Intermediate Language*. CIL is then stored in *assemblies*. Assemblies can be understood as chunks of code interpreted by a virtual machine called *Common Language Runtime*. Compiled .NET programs, CLIs stored in assemblies, come as a part of PE executables. Class imports are discussed in this section primarily.

3.4.1 Streams and MetaData Tables

A *stream* is a "section" which contains a specific kind of data. This data is referenced by so called *MetaData Tables*. The tables implement a multitude of .NET functionality such as defining, importing, and exporting .NET classes. There are several default streams present in .NET binaries.

#Strings stream is an array of ASCII strings terminated by a null byte. The strings in this stream are referenced by MetaData Tables. *#US* stream is an array of Unicode strings. The name stands for User Strings, and these strings are referenced directly by code

instructions. *#Blob* stream encodes .NET classes, methods, and their parameters. *#GUID* stream contains 128 bits long globally unique project identifiers called GUIDs. *#~* stream contains the MetaData Tables [25].

3.4.2 TypeRef Table

.NET framework is known for its reusable code libraries. Classes are compiled into libraries that are later imported by different programs. Knowing what classes are imported can heavily speed up analysis, since they can hint at malware intentions.

Imported classes are referenced by *TypeRef Table* entries, *TypeRefs*. TypeRef is defined by the name of an imported class, its namespace, and its origin.

Names and namespaces of imported classes are present in the form of a *TypeName* and *TypeNamespace* indices. These indices point to *string stream*. String stream is a set of null-terminated strings ordered consecutively.

Source is a bit more tricky as TypeRefs can originate from different sources. Sources are stored in the form of a *ResolutionScope index*. Two least significant bits of ResolutionScope indices are called *tags*. Rest of the bits form an index to a table determined by the tag described in Table A.3.

If a TypeRef originates in an external module, then a row in *ModuleRef Table* is referenced by the ResolutionScope index. Each row in a ModuleRef Table represents a reference to an external module. If an import is present in the current assembly then a row in *Module Table* is referenced. Module Table is a single row table representing the current assembly. For imports originating in external assemblies, the ResolutionScope index is referencing a row in *AssemblyRef Table*. For nested classes, the ResolutionScope index is pointing to a TypeRef Table. Descendant TypeRefs reference parent TypeRefs, thus, making a hierarchy of nested classes [25].

3.5 Visual Basic

In this section, a fraction of Visual Basic metadata present in PE files is presented. Further, use of this information for malware analysis is discussed. Most of the information presented hereafter has been adopted or derived from the work of Alex Ionescu [20].

Visual Basic is an event-driven object based programming language developed by Microsoft corporation back in 1991. Visual Basic was created for beginners to learn programming basics. Visual Basic has various language features, from which the most notable are basic support for object-oriented programming and pre-implemented GUI components. Despite claims that Visual Basic programs "benefit from security" [7], today they are almost exclusively created by malware authors.

From a malware analyst's perspective, programs written in Visual Basic contain tons of useful information usable for malware classification. Aside from regular metadata extractable from PE executables, Visual Basic programs contain additional metadata and compilation relicts. A peek into Visual Basic internals is needed for extraction of this metadata. Unfortunately, the Visual Basic file format is officially undocumented however it was partially reversed by reverse engineers throughout the years.

3.5.1 Visual Basic Execution

Visual Basic programs can be compiled to *native code* or so called *P-Code*. Native code, as the name suggests, runs natively on the processor. Contrary, P-Code is an intermediate language interpreted by *MSVBVMxx.DLL* virtual machine [28]. Execution of a Visual Basic program starts with a call to *ThunRTMain* function as shown in Figure 3.9. This function takes a single parameter: pointer to a start of hierarchically structured metadata beginning with *VB Header*.

```
004010C3     00              DB 00
004010C4   $ 68 5C124000     PUSH Malware.0040125C             Entry Point: Push VBHeader *
004010C9   . E8 EEFFFFFF     CALL <JMP.&MSVBVM60.#100>         CallThunRTMain
004010CE   . 0000            ADD BYTE PTR DS:[EAX],AL
004010D0   . 0000            ADD BYTE PTR DS:[EAX],AL
004010D2   > 0000            ADD BYTE PTR DS:[EAX],AL
004010D4   . 3000            XOR BYTE PTR DS:[EAX],AL
004010D6   . 0000            ADD BYTE PTR DS:[EAX],AL
```

Figure 3.9: Visual Basic executable entry point

In this work, some metadata regarding execution are extracted. Neither deep inspection of native code, nor P-Code is done.

3.5.2 Visual Basic Metadata

In this section, Visual Basic metadata hierarchy is presented. Most relevant metadata structures are later described in more detail. Addresses mentioned in the following text are virtual. Offsets described throughout this section are relative to the beginning of a structure they are a part of. Note that only an excerpt of Visual Basic format is presented here. More information regarding Visual Basic file format can be found in work of Alex Ionescu [20] and Andrea Geddon [15].

Visual Basic executables contain information about objects, imported functions, and much more. Figure 3.10 depicts a Visual Basic metadata hierarchy. Structures referenced by addresses are linked with arrows. Relevant structures are highlighted.

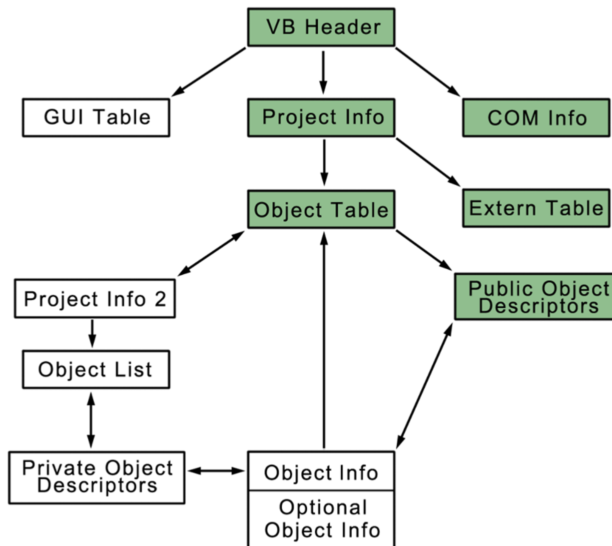

Figure 3.10: Visual Basic metadata structures

VB Header is referenced by the first parameter of *ThunRTMain* function. It references Project Info Header and COM Register Data. Further, VB Header contains information about project, language, "real" entry point, and other data. Format of VB header can be found in Table A.4.

Project Info is referenced by VB Header and references External Table and Object Table. It contains information code, project path, imports, and others. Its format is shown in Table A.5. If *native code address* member is zero, P-Code executable is implied. Otherwise native code is implied.

External Table is referenced by Project Info. Names of imported functions are stored here. There are two types of Visual Basic imported functions, internal and external. Internals are present in the virtual machine module and Externals are imported from external modules. External Table is an array of entries containing import type and address. If an entry is of External type than this address points to an *External Entry Data* structure. External Entry Data contains address of name of imported module and function [15]. External functions are imported dynamically during runtime, so it is not surprising that malware authors encrypt function names to prevent static analysis. These names are decrypted during execution. Table A.6 describes format of External Table.

Object Table is referenced by Project Info and references Public Object Descriptors. It contains information about project, language, object count, and more. Format of Object Table is presented in Table A.7.

Public Object Descriptors are referenced by Object Table. POD is an array of structures, each describing an object. Objects are described by their name and names of their methods. To retrieve method names, an address of *method names array* from POD entry is read first. Method names array is an array of addresses. When iterating over this array, invalid or zero addresses are skipped. Valid addresses point to null-terminated ASCII method names. Method names are not needed for proper execution of a Visual Basic program. As a consequence, malware authors often patch method names with zeros. Format of Public Object Descriptors can be seen in Table A.8.

The *COM Register Data* structure contains information used if the image file is ActiveX and contains valuable data such as TypeLib information. COM Registed Data is referenced by VB Header and in case a valid COM object needs to be registered, it references a *COM Register Info* structure described in Table A.10. Format of COM Register Data is to be seen in Table A.9

Chapter 4

Malware Analysis Tools Developed by Avast Software

Avast is one of the most known anti-malware companies worldwide. Avast has multiple products providing security solutions for various platforms. In this section, some of these products related to malware classification and decompilation are described.

4.1 Clusty

Clusty is a *clustering* system designed for malware clustering and classification. Malware clustering is a process of dividing a set of executable samples into disjoint subsets called *clusters*. Clusters group together samples that are logically related. The main challenge of clustering lies in recognising which samples are related. Samples are clustered based on anexact match or similarity of their features.

Given that samples are grouped in clusters, one can analyze a single executable sample from certain cluster, determine its maliciousness and therefore classify the whole cluster as malware of a certain type. Clustering technology drastically speeds up malware detection process, because it reduces the number of samples to be analyzed to a smaller number of clusters. Today, as malware grows large, using clustering technologies is inevitable.

Clustering process consists of three main phases. First, samples are analyzed and features are extracted. For this task, Clusty uses FileInfo, a tool later described in Section 4.3. Second, samples are divided into clusters based on similarity of their features. These features do not necessary have to be the same for every cluster and can be prioritized. Third, extraction of cluster attributes is done. Clusters are described by their most significant characteristics. Having this information helps to quickly estimate the nature of samples present in a given cluster [23].

4.2 RetDec

RetDec is a decompilation tool used for partial reconstruction of source code of executables from their machine code. RetDec produces intermediate language that can be further abstracted into a C or Python like language [24]. RetDec does not target specific platform or operating system. Currently supported architectures are x86, x64, ARM, MIPS, and others. Supported file formats are PE, ELF, COFF, Mach-O, and others. Figure 2.2 shows a program decompiled by RetDec. For preprocessing of executables, FileInfo tool is used.

4.3 FileInfo

FileInfo is designed to dump numerous statically extracted features from various executable file formats. FileInfo is a part of the RetDec decompilation tool and clustering system Clusty. It can also be used as an early step during manual malware analysis.

The whole feature extraction process goes as follows: First, an input file is parsed into internal representation. This functionality is implemented in the *FileFormat* library, the library FileInfo is built on. Further, the parsed data is processed. This includes various statistical methods, of which some are described in later chapters. Finally, the relevant data is extracted and presented as uniform output.

Currently supported input file formats are ELF, PE, Mach-O, COFF, and others. Supported output formats are JSON and human readable format called plain format. Some of features that contemporary version of FileInfo is capable to extract are:

- Architecture information
- Various section information
- Imported and exported symbols
- Information about resources
- Relocation, certificate, dynamic, and symbol tables
- .NET information

and much more. Despite a complex variety of supported extractable features, these are not sufficient in some situations. Malware comes in all shapes and forms so FileInfo has to be continuously developed to detect previously undetected malware features. The following chapter is dedicated to design of such features.

Listing 4.1 shows an excerpt of information FileInfo is capable to extract.

```
Input file           : ../3e7126c600eb3d73c9b470aa98f2a416
CRC32                : 30226ccd
MD5                  : 3e7126c600eb3d73c9b470aa98f2a416
File format          : PE
File class           : 32-bit
File type            : Executable file
Architecture         : x86
Endianness           : Little endian
Image base address   : 0x400000
Entry point address  : 0x4010c4

Detected tool  : Visual Basic (6) (compiler), linker libraries heuristic
Detected tool  : Microsoft Linker (6.0) (linker), combined heuristic
Detected tool  : Microsoft Visual Basic (5.0) (compiler),
                 40 from 40 significant nibbles (100%)
Detected tool  : Microsoft Visual Basic (5.0 - 6.0) (compiler),
                 24 from 24 significant nibbles (100%)

Timestamp                    : 2010-09-12 13:05:04
Declared number of sections  : 3
Checksum                     : 399895
```

Listing 4.1: FileInfo output

Chapter 5

Design and Extraction of Malware Features

Features, as stated earlier, are of different nature. In this work, features are designed for multiple purposes: identification and classification of malware, decompilation assistance, and manual analysis. When designing features, availability, reliability, suitability for malware classification, and other aspects are taken into account. Designed features from various fields are described in the following sections.

5.1 .NET Features

FileInfo extracts some .NET features. For example, the support for .NET classes reconstruction has already been implemented by Marek Milkovič [23]. However, it is lacking any support for extraction of imported classes. In this section, TypeRef Table reconstruction is discussed. Further, a method of TypeRef Table hash creation is described.

5.1.1 TypeRef Table Reconstruction

The TypeRef Table contains rows of TypeRefs, each describing a single imported class. TypeRefs are defined by three attributes, by their name, namespace, and their source. The source can originate in an external module, current assembly, external assembly, or in another TypeRef. The goal here is to reconstruct the TypeRef Table so each imported class would have a string representation of its name and namespace. Additionally, a string representation of source is present for imported classes originating in external assembly.

TypeRef originating in another TypeRef implies a nested class. TypeRefs of descendant classes reference TypeRefs of their parents. These TypeRefs are linked after reconstruction of the table. Loops are detected and eliminated during the linking process. The whole process is depicted in Figure 5.1.

5.1.2 TypeRef Table Hashes

Exact match of two TypeRef Tables is implied by the exact match of their cryptographic hashes. Hashes are computed from concatenated string representation of table rows. Proper construction of hashed data is needed due to possible malformation of TypeRef Table. Two aspects have to be taken into account. First, the construction must cope with some TypeRef data missing. Second, maximization of uniqueness of hashes has to be considered. For this

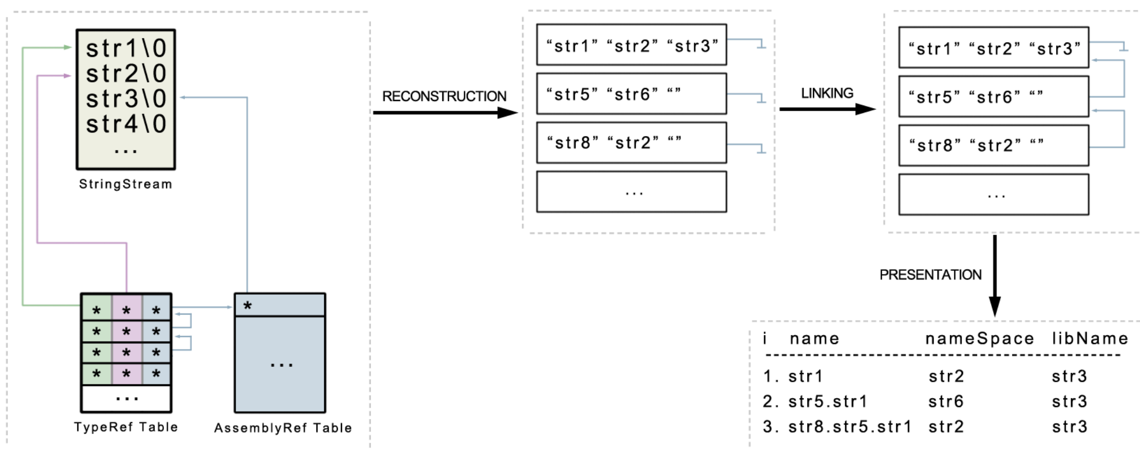

Figure 5.1: TypeRef Table reconstruction

reason, hashed data is constructed from raw TypeRef records rather than reconstructed and linked ones. Furthermore, sources of any type are stringified. Source string is appended with type-unique suffix to distinguish source types. TypeRef attributes are separated with non-colliding delimiter. Listing 5.1 shows an example where omitting a source type would result in producing same cryptographic hashes. SHA256 and MD5 hashes are computed. TypeRef Table hashes are suitable for malware classification, because TypeRef data is contained within a section with certain access privileges. These privileges are checked at runtime and in case of modification the assembly will not start [25].

```
AssemblyRef Table:
    0 AmbiguousName
    1 ...

TypeRef Table A:
    0 Name0         NameSpace0 assemblyRefTab[0]
    1 AmbiguousName NameSpace1 assemblyRef[1]

TypeRef Table B:
    0 Name0         NameSpace0 typeRefTab[1]
    1 AmbiguousName NameSpace1 assemblyRef[1]
```

Listing 5.1: TypeRef hashes ambiguity

5.2 Icon Features

Among malware, icons are used to deceive or attract their victims. People tend to trust executables with icons, especially if they are familiar to them. Malware authors often update or modify their software but leave the icons. Multiple problems complicate hashing of icons. First, if several icons are present, one that the user might see needs to be identified.

Second, malware authors often slightly modify icons to prevent them from being classified based on exact match of their cryptographic hashes as seen in Figure 5.2.

This section is dedicated to extraction and hashing of icons. Suitability for classification based on icon similarity is discussed. Further, constraints of presented methods are mentioned.

Figure 5.2: Application icon modified with noise to alter its cryptographic hash

5.2.1 Icon Parser

Icons in ICO file format are compiled into Icon and IconGroup resources. These resources are then stored in a resource tree. Every Icon resource points to an icon data stored in DIB format. The goal here is to parse an icon that will be displayed in desktop environment, further referred to as the *main icon*.

To achieve this, all icons are parsed and grouped based on their IconGroups. First, all resources in the resource tree are parsed and unrelated resources are filtered out. Second, all icons are grouped. Grouping algorithm is as follows: Entries of every stored IconGroup are searched for match of entry nameID and nameID of stored Icons. The matched icon is linked and its properties are set.

Data entry of a main icon resource is accessed after the icon has been chosen. Finally, data of the main icon is parsed. Proper icon parser should support all color depth formats of DIB.

Choice of main icon has not been discussed yet. Unfortunately, main icon cannot be determined precisely. This is due to fact that Windows desktop environment chooses an icon to render based on a curent DPI setting [6]. The best shot is to choose a commonly used DPI and investigate choice of rendered icon. Creation of icon priority list was made as follows: First, an executable with multiple distinguishable icons was created. These icons had different dimensions and color depths. Rendered icon, the most prioritized, is observed and then removed. This process is repeated until there are no icons left. Complete list of icon priorities can be found in Figure 5.3. Icon with lowest priority number is considered to be the most prioritized. Icons not present in the list are considered to have lower priority than those that are. This list was created on a 96 DPI Windows 10 system. The main icon is the most prioritized icon in most prioritized IconGroup. Most prioritized IconGroup is one with the lowest nameID value.

Obviously, using this method for determining main icon can produce wrong results. Another limitation is that the created priority list does not take *color count* into consideration. Color count is a member in icon group directory entry present in Figure 3.4. Violating its format can cause Windows to chose a suboptimal icon, thus resulting in an incorrect choice of main icon [12].

Priority	Dimensions	Color Depth (bpp)
0	32×32	32
1	24×24	32
2	48×48	32
3	32×32	8
4	16×16	32
5	64×64	32
6	24×24	8
7	48×48	8
8	16×16	8
9	64×64	8
10	96×96	32
11	96×96	8
12	128×128	32
13	128×128	8
14	256×256	32
15	256×256	8

Figure 5.3: Icon priority list

5.2.2 Icon Hashes

When main icon is extracted, its hashes are produced. SHA256 and MD5 hashes are produced from raw main icon resource data. Cryptographic hashes are suitable for clustering of executables based on an exact match of their icons. However, they are practically unusable when estimating icon similarity. Slight modification of an icon results in complete change of its cryptographic hashes. Icons of malware samples are often modified with noise to prevent testing for exact match. Such noise can be seen in Figure 5.2.

This limitation can be overcome by using *perceptual hashes*. Perceptual hashes, contrary to cryptographic hashes, are designed to represent images in a way that slight modification of the image results only in slight change of its perceptual hash. One such hash is so a called *AverageHash*. AverageHash behaves like a low pass filter effectively filtering out details.

Example 5.4 demonstrates the principle of AverageHash computation. Image is first resized to 8×8 and converted to greyscale and subsequently to black and white. Every row is a sequence of eight black or white pixels and therefore can be represented as a byte. Eight rows form an 8 byte AverageHash [22]. AverageHash is surprisingly effective considering its computational efficiency.

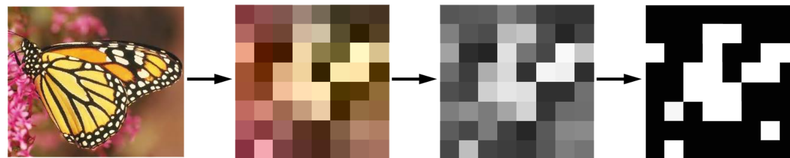

Figure 5.4: AverageHash computation

Similarity of icons is determined by Hamming distance of their average hashes. Icons are considered similar if Hamming distance of their perceptual hashes is smaller than given threshold. Threshold with a value of 3 has been experimentally proved to be suitable.

First step of AverageHash computation is image resizing. In this work, two resizing algorithms have been tested. *Box sampling algorithm* turned out to be more suitable than *bilinear interpolation algorithm*. This is because bilinear interpolation does not take all image pixels into consideration. This is unacceptable when working with small images such as icons.

5.3 Visual Basic Features

Visual Basic executables are rich in metadata. These contain information about project, code, some unique identifiers, objects, imports, COM information, and much more. In addition, duplicity of some data can be made use of. In this section, Visual Basic features, their extraction, their suitability for malware classification, and reliability is discussed.

5.3.1 Metadata Parser

To get as much information about a Visual Basic executable as possible, all relevant metadata structures are parsed first. Visual Basic metadata parser must be *robust*, meaning it is able to cope with malformed data. Some metadata can be extracted from multiple sources. If some of the data is corrupted, the parser should attempt to extract it from its valid duplicity. The parser implements a top-down approach. Top level structure of Visual Basic metadata hierarchy is parsed first. Further, structures that are referenced by already parsed ones are processed and so on.

5.3.2 Project Metadata Features

Visual Basic programs are mostly created as Visual Studio[1] projects, sometimes called *solutions*.

Malware authors often produce multiple versions of the same malware. In many cases, all of these versions come as products of the same project. This comes particularly handy when classifying malware based on its project source. On the other hand, this information is not necessary for proper execution of a Visual Basic program. Hence it is not a big surprise that malware authors remove or modify it to mislead analysts. Complete list of extracted features regarding project info can be found in Table 5.5. These features originate in various structures of Visual Basic metadata. Unique project identifiers are present in COM related structures and are described in Section 5.3.5.

5.3.3 External Table Features

External Table contains information about imported functions. Each row determines an imported module and a function name. Format of External Table is described in Section 3.5.2. The goal here is to reconstruct this table in a way that it can be further processed and presented.

Classification of malware based on its imports is a common practice. Import Tables of Visual Basic executables provide insufficient amount of data needed for classification. This is because external functions are imported through External Tables instead of Import Tables. Therefore, Visual Basic executables are classified based on External Tables rather than Import Tables. On the other hand, these functions are imported at runtime. Due

[1]Visual Studio is a widely spread integrated development environment created by Microsoft

Feature	Origin
project name	VB Header, Object Table, COM Register Data
project exe name	VB Header
project description	VB Header, COM Register Data
project path	Project Info
project help file name	VB Header, COM Register Data
primary language DLL name	VB Header
secondary language DLL name	VB Header
primary language DLL LCID	VB Header
secondary language DLL LCID	VB Header
project primary LCID	COM Register Data
project secondary LCID	COM Register Data

Figure 5.5: Visual Basic project features

to this, External Tables of malware sometimes contain encrypted function names that are decrypted during execution.

After reconstructing the External Table, its cryptographic hashes are computed. String representation of each row is concatenated consecutively. Row attributes are delimited with a non-colliding delimiter. SHA256 and MD5 hashes of the resulting string are produced. Such hashes are used for classification of malware based on an exact match of their External Tables.

5.3.4 Object Table Features

Object Table in combination with Public Object Descriptors contain information about object and method names.

When parsing Public Object Descriptors, one has to have on their mind that the declared number of methods in Public Object Descriptor is generally significantly greater than the number of valid method name addresses in method names array. As a consequence, null addresses must be skipped. Reconstruction goes as follows: First, Public Object Descriptors address and number of objects are read from the Object Table. Then Object Descriptors, an array of consecutive structures, are iterated over. Afterwards, object name and method name arrays are read from the current descriptor. Method name array points to a sequence of addresses. These addresses point to ASCII strings, method names.

Exact match of two Object Tables is indicated by an exact match of their cryptographic hashes. The Object Table is turned into a continuous string by concatenating string representation of objects. Object names are followed by method names separated with a non-colliding delimiter. SH256 and MD5 hashes are produced.

Object Tables, among other information, contain GUIDs identifying them. GUIDs are globally unique and bound to projects and therefore serve as valuable features as they are. Unfortunately, analysis showed that Object Table GUIDs are generated during every compilation and thus are not suitable for classification.

5.3.5 COM Information

Malware authors sometimes prefer ActiveX COM objects over regular executables as can be seen in Table 7.2. COM objects contain duplicate data such as project name and LCID, but

most importantly they contain numerous CLSID (project unique GUIDs). Multiple Visual Basic applications were compiled and analyzed to confirm that Object CLSID present in the COM Register Info structure is project unique and does not change with recompilation. TypeLib CLSID present in COM Register Data is project unique and nonvolatile as well, but only in case of a valid COM object.

5.3.6 P-Code Recognition

Visual Basic executables come in two forms, native and P-Code. Native executables run natively on the processor. Contrary, P-Code is an intermediate language interpreted by the Visual Basic virtual machine. As already stated, this work does not aim for deep code inspection, but having information about the nature of code is valuable. P-Code is indicated by a zero value of *native code address* member in Project Info structure described in Figure A.5. In any other case native code is implied.

5.4 Detection Features

Features introduced in this section are designed to provide additional information about analyzed samples. This includes detection of irregularities, auxiliary information added by compilers, statistical information about data, and extraction of information necessary for further processing.

5.4.1 PE Anomalies

Complex specification of PE file format defines value domains of numerous structure fields, but only a subset of them is used prevalently among legitimate software. Besides use of the unusual values, malformations are considered anomalies as well. Compilers sometimes violate PE file format despite the fact that its documentation is publicly available.

Format violation is however mostly a result of manual binary instrumentation. Malware authors frequently modify compiled executables in order to hide their malicious behavior and to avoid detection. Major problem of such modifications is that an executable with obsolete, missing, or malformed content can often be loaded by Windows loader. This is because numerous values defined in the file format specification are not checked by the loader.

Frequent occurrence of anomalies in executable can serve as a strong indication of malicious intentions. Katja Hahn has implemented a scanner for numerous PE anomalies as a part of her master thesis [16]. Moreover, she provided statistics on distribution of anomalies among malicious and legitimate software. The statistics were made out of 100.000 malware and 50.000 goodware samples. Subset of anomalies that are prevalent in malware and rarely occur in goodware were adopted. The subset of anomalies is listed in Table 5.1. Further, usual section name list of Hahn for anomaly 7 has been replaced with more general list from Hexacorn blog [18].

Some anomalies were newly designed as a part of this work. List of novel anomalies can be found in Table 5.2

List of usual packer section names for anomaly 6 has been adopted from the mentioned Hexacorn blog as well.

i	Anomaly	MW freq	GW freq
1	Section marked uninitialized but contains data	12.9%	0.13%
2	Physical range of sections overlaps	26.9%	1.3%
3	Entry point in last section	5.1%	0.1%
4	SizeOfRawData in SectionTable is zero	46.6%	5.8%
5	Entry point located in writable section	26.1%	3.1%
6	Unusual section name	51%	14%
7	Unusual section characteristics	42.7%	21.1%
8	Imports/Exports/Resources stretched over sections	14.5%	0.5%
9	SizeOfHeader not multiple of FileAlignment in OptHeader	15.9%	2.5%

Table 5.1: Adopted subset of Hahn's anomaly list

i	Anomaly
1	Duplicate section names
2	Entry point outside mapped sections
3	Entry point inside non-executable section
4	Section size over 100MB
5	Resource size over 100MB
6	Usual packer section name

Table 5.2: Newly designed anomaly list

5.4.2 VersionInfo Extraction

Compilers often add auxiliary information about the product to executable files. Such information can be obtained from VersionInfo resource. This data includes a list of supported languages and project properties stored as key-value string pairs. Resources are described in Section 3.1 and VersionInfo resource itself is discussed in more detail in Section 3.3.

VersionInfo data is extracted as follows: VersionInfo resources are separated during parsing of ResourceTree. Further, the set of obtained VersionInfo resources is processed. The resources contain a hierarchy of type-length-value (TLV) nested structures. These structures are parsed by a a top-down parser. VersionInfo structures share the same header format. Only the length of the structure is read from the header, rest of the fields is neglected. Type of structures can mostly be deduced from parsing context, as almost all structures contain children of a known type. If the type cannot be deduced, the type identifying string is read. Otherwise the type string is used as validation rather than identification.

The problem is that the number of children contained within the structures is not declared anywhere. Structure length is the only indication of possible number of children of the structure. For this reason, the parser processes children consecutively. Optimistic consumption of parsed bytes cannot exceed boundaries determined by structure length declared in the header structure.

List of supported languages is extracted from the Var structure. Child of var structure is an array of 32 bit language identification values. The lower word of the value is a 16 bits long LCID and the higher word of the value is an IBM code page identifier. Both the LCID and IBM code page identifier are translated to their string representation as defined by LCID table (see [8]) and IBM code page identifiers table (see [4]).

Key-value string pairs are obtained from String structures. Type identifying string of String structure is the key and its child is the value. These pairs can depict the origin of a file, the time it was created, and much more. Complete list of usual pairs can be found in VersionInfo string guide lines (see [5]).

VersionInfo resource is not used for anything besides providing additional information about applications. Their presence in the file is optional and they do not influence execution of programs. This is why malware authors corrupt or omit them completely. The analyst should not rely on information obtained from them.

5.4.3 Thread-local Storage Preprocessing

Malware authors sometimes implement malicious functionality in one of the thread-local storage callbacks as described in Section 3.1.2. Thus special attention should be paid to them. Addresses of the callbacks can be statically extracted from the executable file. Thread-local storage directory contains a reference to null-terminated array of callback addresses called *Address of Callbacks*. These addresses are particularly useful during decompilation process. One of major problems in decompilation is the inability to precisely distinguish data from code. Addresses of callbacks are therefore used to mark byte sequence as code for decompilation. Further, thread-local storage directory contains *Raw Data Start* and *Raw Data End* determining bounds of thread-local data used for decompilation as well.

5.4.4 Entropy

One of commonly used methods of avoiding static analysis that malware authors use is to *pack* their program. A packed binary contains compressed data and code that are decompressed during runtime by decompression routines. This is particularly inconvenient, because extraction of static features becomes very hard or practically impossible. One of possible approaches to statically extract data from packed binaries is to unpack them before proceeding with static analysis. Unpacking is unreliable and often lossy process and requires knowledge of the decompression algorithm. Given that decompression of a packed binary is not possible, detection of packed status of data is still valuable. Packer usage can be a strong indication of malicious intentions. For this reason, entropy of sections and overlay is computed.

Entropy of set of probabilities is defined by Shannon's formula:

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

The entropy is multiplied by 8, because of the fact that most tools for malware analysis express entropy as a number in $[0, 8]$ rather than in $[0, 1]$ interval.

High data entropy indicates significant data diversity and often implies compression. Thus it is used for detection of packed data. On the contrary, low data entropy indicates data similarity and can be used to detect blank data sections.

Chapter 6

Implementation of Extraction of Designed Malware Features

This chapter is dedicated to implementation details of features designed in previous chapters. Infrastructure of FileInfo, program design of each feature, and general development information are discussed in this chapter.

6.1 FileInfo Infrastructure

FileInfo is described in Section 4.3. Its design and structure is discussed hereafter. FileInfo is written in C++ and follows ISO C++14 standard. All designed features are implemented in ISO C++14 for this reason as well.

FileInfo is part of the RetDec decompiler described in Section 4.2. FileInfo shares some code with other RetDec subprojects, but the code is not relevant for this work and will be further neglected as FileInfo was self sufficient.

Figure 6.1 shows the most relevant parts of directory tree of RetDec source code. Architecture of FileInfo can be divided into two separate layers - extraction and presentation layer. Extraction layer consists of binary file format parsers, extractors, and postprocessors. This layer is implemented in *FileFormatl* library and third party libraries. Presentation layer wraps the extraction layer and prints extracted information.

FileInfo is implemented in the presentation layer and it is an abstraction of FileFormatl library. Everything presented by FileInfo executable is received from the FileFormatl library. FileInfo targets multiple file formats - PE, ELF, Mach-O, COFF, and Intel HEX. Information extracted from these formats is represented uniformly. Some structures, however, are format specific. A class diagram of FileInfo architecture can be seen in Figure 6.2.

This work primarily focuses on PE file format, which is parsed by *PeLib*. PeLib is a third party open source C++ library with the purpose to ease access to and modification of PE files. PeLib implements a multitude of classes which represent all important PE header and directory structures and which provide the necessary functions to read, modify, and write these structures. [27]. A subset of PeLib functionality is used by FileFormatl library.

FileFormatl library supports more advanced reconstruction of data structures than PeLib. What is more, FileFormat provides abstractions of previously mentioned file formats. The formats themselves are represented by `FileFormat::FileFormat` class. This complex class provides information common for all binary file formats, such as information

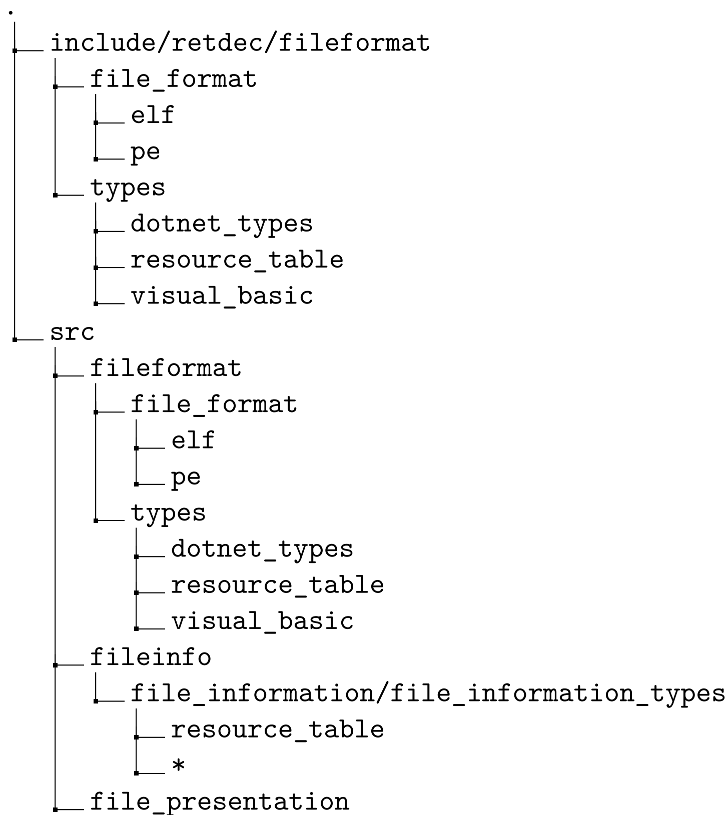

Figure 6.1: RetDec directory tree

about entry point, overlay, or imported functions. Format specific information is accesible via derived classes, such as `FileFormat::PeFormat` class.

Every presentable FileFormatl class is wrapped by a FileInfo wrapping class. These wrapping classes are contained withing a singleton class called `FileInformation`. This class is presented by so called presentators. FileInfo supports two presentators — plain and JSON. Plain presentator outputs information in human readable form. JSON presentator is designed to output information in JSON format to be easily processed by other programs, most notably the clustering system Clusty.

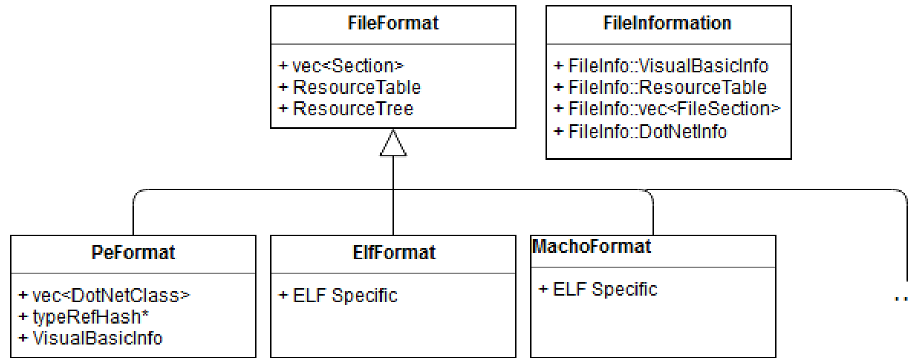

Figure 6.2: FileInfo infrastructure

6.2 Feature Extraction

Implementation of features designed in Chapter 5 is described in this section. Algorithms for feature extraction are implemented as a part of FileFormatl library and features themselves are accesible via FileInfo wrappers. Description of implementation of FileInfo wrappers is omitted since the implementation is trivial.

6.2.1 TypeRef Table Features

TypeRef Table described in Section 3.4.2 implements a class importing system of .NET framework. TypeRef, a single row of TypeRef Table, contains a name, namespace, and origin indices. Name and namespace indices point directly to #Strings stream. Origin index points to a table, type of which is determined by the index tag. It can either be Module, ModuleRef, AssemblyRef, or TypeRef Table. TypeRef originating in another TypeRef implies nested classes. If the origin index references an AssemblyRef Table then the TypeRef originates in an external assembly described by indexed record in the AssemblyRef Table.

Reconstruction of a TypeRef Table was already partially implemented by Marek Milkovič as a part of his thesis [23]. Reconstructed TypeRefs, which are implemented in `DotnetClass` class provided no information beside a set of indices. This reconstruction is innovated with extraction of string representation of a name and namespace referenced by first two indices. Name of an external assembly is obtained in case of AssemblyRef Table entry being referenced by an TypeRef origin index. TypeRefs originating in another TypeRef (nested classes) are linked, meaning that instances of `DotnetClass` contain a pointer to another (parent) instance. Set of mutually referenced TypeRefs are modeled with a directed graph. Depth First Search algorithm is used for detection of cycles in the directed graph of TypeRefs. This is necessary to ensure that the linking process is finite and will not result in crashing FileInfo. TypeRefs that are already processed are stored in a vector of processed nodes. TypeRefs being currently processed are pushed onto a stack. If a TypeRef references an already processed TypeRef or a TypeRef in the stack of currently processed TypeRefs then the edge is ignored and link to parent is omitted.

All of the three novel `DotnetClass` attributes are presented. A name of TypeRef is concatenated with a name of its parent and the parent name is concatenated with its parent's name and so on. If the parent's name is missing then it is replaced with a string representation of its TypeRef Table index. A namespace is presented as is. Origin is presented just in case of a referenced table being an AssemblyRef Table.

Listing 6.1 shows how reconstructed TypeRef Table can look.

```
"typeRefTable" : {
    "md5" : "5742603226df6e720f055413ba924c2c",
    "types" : [ ... , {
        "index" : "3",
        "libraryName" : "System.Runtime",
        "name" : "DebuggingModes.DebuggableAttribute",
        "nameSpace" : "System.Diagnostics" }, ...] }
```

Listing 6.1: JSON presentation of TypeRef Table

TypeRef Table hashes are computed from unlinked tables to maximize their uniqueness. Hashed data is constructed from a name, namespace, and origin separated with non-colliding delimiter from all TypeRefs as follows: If a name or a namespace of a TypeRef is available in string representation then it is concatenated to hashed data. Origin is represented as a name of referenced table entry postfixed with an abbreviation of table type. For example, origin of TypeRef referencing an AssemblyRef Table entry could be `"NameOfAssemblyRef" + "AR"`. This representation of origin is concatenated to referenced data.

Types of origin tables (or their abbreviations) need to be taken into consideration because omitting them may potentially result in production of same hashes for different TypeRef Tables. This could happen in case of a TypeRef referencing an AssemblyRef entry or another TypeRef Table entry and the entries having the same name.

6.2.2 Icon Features

Icons are stored in Icon resources grouped by IconGroup resources as described in Section 3.2 and Section 5.2. First step to producing cryptographic and perceptual hashes of the main icon is parsing of ResourceTree described in Section 3.1.1. FileInfo already parses ResourceTree into `ResourceTable` class as vector of `Resource` class instances but no further processing is done. Parsing of ResourceTree is modified to separate Icons and IconGroups from other resources.

`Resource` class contains just a nameID, typeID, langID, and a reference to raw resource data. Raw data of IconGroups is parsed first. Number of icons within a group and properties of the icons are read from the IconGroup data (see Figure 3.4). Separated Icon resources are then scanned for a match of their nameID and nameID declared in one of the icon properties read from an IconGroup data. If the nameIDs match, then the Icon resource is linked with the IconGroup. Properties of linked Icon resources are set to those that were read from raw IconGroup data.

Group with nameID equal to zero (the most prioritized) is searched for the main icon. The main icon is the most prioritized icon in the priority list shown in Figure 5.3. An array of Standard Template Library (STL) template type `std::pair` entries represent such a priority list. The first member of `std::pair` entries contains both dimensions of icon. A single numerical value is sufficient to reflect both icon dimensions, because only icons of same dimensions are taken into considetarion by the priority list. The second member is color depth of an icon. Selection of the main icon is done by a standard function `std::max_element` where the default comparison function is replaced with `iconCompare` function. The `iconCompare` function simply checks whether the widths of icons are equal

to their heights and then it iterates over the priority list. First match in the priority list determines which compared icon is more prioritized.

Raw data of the main icon is parsed in order to compute its AverageHash. `BitmapImage` class is newly designed to represent a two dimensional image with 32 bpp color depth. Icons of 1, 4, 8, 16, 24, and 32 bit color depths are parsed and represented by this class. The `BitmapImage` implementes numerious image operations necessary for AverageHash computation, namely downscaling, greyscale conversion, and black and white conversion.

Its worth mentioning that the resizing method of the class was implemented twice with different downscaling algorithms - box sampling and bilinear interpolation. Bilinear interpolation downscaling takes only a small subset of image pixels into consideration and is therefore unacceptable when working with such small images as icons, because the information loss may significantly affect hashing results. For this reason, the version with bilinear interpolation algorithm was neglected.

Cryptographic hashes are produced from raw Icon resource data and AverageHash is produced from `BitmapImage` representation of the main icon. Computed icon hashes can be observed in Listing 6.2.

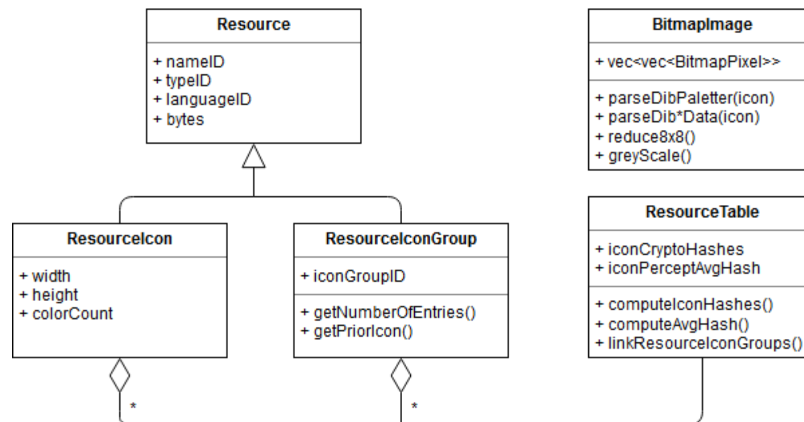

Figure 6.3: Icon class diagram

```
Resource table
--------------
Number of resources: 111
Icon CRC32          : c6009c34
Icon MD5            : 8b6fdb44e0b3e55bf9bc8ffda1800b79
Icon SHA256         : 4d8b8a948b29bf38cd8f186e75b58ed5-
                      017ed11aac7ebeb453752181b58f3bea
Icon AvgHash        : b7478387b4ffaeff
```

Listing 6.2: PLAIN presentation of icon perceptual and cryptographic hashes

6.2.3 Visual Basic Features

Visual Basic discussed in Section 3.5 and Section 5.3 is rich in metadata represented as a hierarchy of various structures. A pointer to the beginning of this hierarchy is passed as

the first parameter to *ThunRTMain* function. Because of this, if the first instruction at entry point of a Visual Basic executable is push <addr> then the address is extracted and metadata hierarchy parsing process begins.

PeFormat class was enriched by Visual Basic metadata parser implemented as a set of separate parsing methods. Overall approach of metadata parsing is as follows: an instance of VisualBasicInfo class is created as a container for all extractable Visual Basic information. For every simple metadata structure there is a C++ structure representation. There is a lot information stored as strings in the hierarchy, therefore two string reading methods for ASCII and Unicode are implemented. FileInfo does not support presentation of wide characters, therefore all unprintable characters are converted to a string representation of their hex codes. Reading of strings is limited to a certain length, since reading an invalid nonterminated string could result in pointless copying of large memory chunks. Further, every address and offset defined within a parsed structure is converted to a file data offset. A method for parsing a referenced object is called only in case the address conversion has been successful. All information extracted during parsing of structures is immediately stored in a VisualBasicInformation container.

There are two more complex structures that need extra attention. Object Table is represented as a vector of VisualBasicObject class instances and External Table is represented as a vector of VisualBasicExtern class instances. Both of the vectors are contained within VisualBasicInformation. Object Table cryptographic hashes are produced from object names separated with a delimiter followed by method names separated with another delimiter. External Table cryptographic hashes are produced from delimited module names and API names of Externals. Class diagram of Visual Basic related classes can be seen in Figure 6.4.

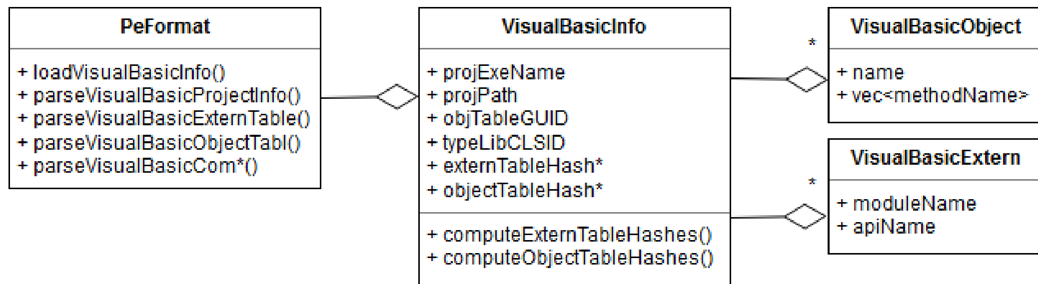

Figure 6.4: Visual Basic class diagram

Based on extracted Visual Basic features presented in Listing 6.3 one can assume that a Swedish malware author with a messy desktop targets Facebook accounts.

```
Visual Basic Information
------------------------
Is P-Code               : Yes
Project secondary LCID : Swedish - Sweden
Project path            : C:\Users\Admin\Desktop_old\Blackshades project\
                          Blackshades NET\server\server.vbp

Visual Basic Object table
-------------------------
MD5  : e0180b0247d3919726ac83645bacefdc
GUID : 6506AE50-051B-3840-A9E0-E9923E7243DF

0. object name: frmMain
   method name: C_Mutex
   method name: BROWSER_FB_OnQuit
   method name: FACEBOOK_START

Visual basic extern table
-------------------------
MD5 : fee82cb91720ee737a165c044609d03c

i   apiName             moduleName
--------------------------------------
24  FtpCreateDirectoryA  WinInet.dll
```

Listing 6.3: PLAIN presentation of small fraction of Visual Basic features

6.2.4 PE Anomalies

Anomalies are described in Section 5.4.1. An anomaly is considered to be either a violation of file format or some unusual characteristic. They are not necessarily bound to any specific format, however, the PE file format is the most relevant to this work. An anomaly container is implemented to be contained within the base class `FileFormat` rather than in the derived format specific class `PeFormat`. This design leaves the possibility of extending FileInfo with other format speicific anomalies open.

The anomalies are represented as a vector of `std::pair<std::string, std::string>`. The first member of each pair is an anomaly abbreviation. The abbreviations are meant to be used by a classifier and are easily processed, but they provide relatively little information. The second member is a description of a given anomaly. The description is longer and more informative, because it is meant to be read by humans.

Pe anomaly scanner is implemented via a set of scanning methods of the `PeFormat` class. During processing of PE files, the anomaly scanner `scanForAnomalies` is called. This

method sequentially calls scanning methods for subset of individual structures contained within `PeFormat`:

- Section table
- Resource table
- Import table
- Export table
- Optional header

Every anomaly present in Table 5.1 and Table 5.2 is implemented in one of the following scanners:

- scanForSectionAnomalies
- scanForResourceAnomalies
- scanForImportAnomalies
- scanForExportAnomalies
- scanForOptHeaderAnomalies

Listing 6.4 shows how detected anomalies are presented.

```
Anomaly table
-------------
Number of anomalies: 5

i  abbreviation      description
-----------------------------------------------------------
0  epInLastSec       Entry point in last section
1  epInWritableSec   Entry point in writable section
2  unusualSecName    Unusual section name: UPX0
3  packedSecName     Packer section name: UPX0
4  uninitSecHasData  Section UPX0 is marked uninitialized but contains data
```

Listing 6.4: PLAIN presentation of detected anomalies

6.2.5 VersionInfo Data

As described in Section 3.3 and Section 5.4.2, VersionInfo resources contain auxiliary data that compilers add to provide additional information about compiled applications. VersionInfo consists of multitude of nested type-length-value (TLV) structures. Type of these structures is identified by a UTF16 type identification string contained within them. This string is redundant if the type can be deduced from parsing context. Most valuable VersionInfo data is a list of supported languages obtainable from children of Var structures and a list of key-value strings present in String structures.

The VersionInfo resources are separated during parsing of the ResourceTree. Each VersionInfo resource is processed by a top-down parser. Because of the fact that first few members of VersionInfo structures are similar, a uniform header is parsed from every processed structure. Only the length of a structure is read from the header, other header values are neglected. The type of the structure is either deduced or the type identification string is converted to ASCII and then compared with strings from a domain of possible types.

LCIDs and IBM Code Page Identifiers are extracted from identification values of supported languages. IBM Code Page Identifiers and LCIDs are converted to string representation by translation functions afterwards. Both translation functions use the translated value as keys to hash tables. If no entry with a given key exists then `"unknown"` string is returned, otherwise the string value corresponding to the key is returned. Both identifier strings are stored in a vector of string pairs contained within a `ResourceTable` class.

If the processed structure is a String then its type identifying string is interpreted as a key and its child is interpreted as a value. The key and the value are converted from UTF16 to ASCII representation. If there is no ASCII representation of a UTF16 character then the character is replaced with a string representation of its hex code prefixed with `"\x"`. The strings are stored in a vector of string pairs present in the `ResourceTable` class as well.

A list of extracted VersionInfo strings and supported languages can be observed in Listing 6.5.

```
"versionInfo" : {
    "languages" : [
        { "codePage" : "utf-16", "lcid" : "German - Germany" },
        { "codePage" : "utf-16", "lcid" : "Unspecified" } ],
    "strings" : [
        { "name" : "CompanyName", "value" : "obama" },
        { "name" : "ProductName", "value" : "Projekt1" },
        { "name" : "InternalName", "value" : "my_st0re_loader_____" }, ...]}
```

Listing 6.5: JSON presentation of VersionInfo languages and strings

6.2.6 Thread-local Storage

Static data local to threads are stored within a .tls section described in Section 3.1.2. Thread-local initialization routines, often called callbacks, are called before entry point execution, therefore malware sometimes implements its malicious behavior in one of the callbacks. Among other information, addresses of callbacks are statically obtainable from TLS directory.

TLS directory is parsed in the PeLib library, however PeLib truncates the upper 32 bits from the 64 bit TLS related values. The library is modified so it would parse 64 bit values in case of a 64 bit PE executable. Parsed data is processed by `loadTlsInformation` method of `PeFormat` class. This method stores parsed data in TLS container class `TlsInfo`. Further, it converts the AddressOfCallbacks to a file offset and reads all addresses from the referenced null terminated array. The addresses of callbacks are stored in a vector contained within the `TlsInfo` class.

Listing 6.6 demonstrates how extracted thread-local storage directory information is presented.

```
"tlsInfo" : {
    "callbacks" : [ "0x401060" ],
    "callbacksAddress" : "0x408044",
    "characteristics" : "00000000000000000000000000000000",
    "indexAddress" : "0x40803c",
    "rawDataEndAddress" : "0",
    "rawDataStartAddress" : "0",
    "sizeOfZeroFill" : "0" }
```

Listing 6.6: JSON presentation of thread-local storage information

6.2.7 Entropy Computation

Malware authors often pack their programs to prevent them from being statically analyzed. High section entropy, as described in Section 5.4.4, can serve as an indication of packed data. Low section entropy suggests that the section is blank. Besides the sections, the file overlay is sometimes compressed to obfuscate its content as well.

General entropy calculation is done from raw data by the `computeDataEntropy` function. This function returns the entropy normalized to interval $[0, 8]$.

Sections are not unique to PE file format. In fact most of binary executable formats support the concept of sections. Because of this, sections are uniformly represented by the `FileSection` class. The `FileSection` class is used only as a container and entropy calculation is done for each format separately. Parsing of ELF, MACH-O, COFF, and PE sections is already implemented. Therefore, the parsers are modified to compute the section entropy right away. Entropy of sections is computed by one of the following functions:

- `ElfDetector::getSections`
- `PeWrapper::getFileSection`
- `MachODetector::getSections`
- `CoffDetector::getSections`

Overlay is uniformly represented as sequence of bytes in `FileFormat` class, therefore the overlay entropy is computed by a single function for all supported file formats.

Section entropies are shown in Listing 6.7.

```
Section table
-------------

i    name     flags    offset      fsize  ...  entropy
------------------------------------------------------
0    UPX0     uxrw     0x00400     0
1    UPX1     ixrw     0x00400     0x17600     7.993
2    UPX2     irw      0x17a00     0x00200     1.293
```

Listing 6.7: PLAIN presentation of section and overlay entropies

6.3 Source Codes, Compilation, and Execution

Throughout this work multiple source code files were either modified or written from scratch. Most of the work has been implemented in FileInfo source code, which is a part of RetDec source code. The source code of RetDec is publicly available in the official GitHub repository [9]. The repository was forked during development. After every completed task, the source code of the fork was merged with the official repository.

Source code of FileInfo regression tests is part of RetDec regression tests source code publicly available in another official GitHub repository [10]. Again, the repository was forked and merged after the working tests for each task were completed.

Portability of RetDec compilation is achieved by using *CMake*. CMake is a cross-platform tool for controlling software compilation process. It generates *Makefiles* later used by the *make* tool to build the actual applications. The building process is described in more detail in the RetDec GitHub repository.

FileInfo can be executed with numerous command line parameters. A small excerpt of the parameters can be seen in Table 6.1.

Abbrv	Full param	Description
-h	--help	Display help
-v	--verbose	Print more information
-p	--plain	Print output as plain text
-j	--json	Print output as JSON
-X	--explanatory	Print explanatory notes

Table 6.1: FileInfo parameters

Besides the mentioned source codes, two python scripts for statistical analysis of malware were written from scratch. Results of the analysis is discussed in the following chapter.

Chapter 7

Testing and Results

As with any other commercial software, FileInfo had to be properly tested before going to production. This chapter discusses testing methodology used during development of FileInfo. Further, statistical discoveries of malware and overall results of the work are presented.

7.1 Regression Tests

Numerous regression tests cases are designed to prevent modification of FileInfo source codes and addition of new features to affect FileInfo's behavior regarding previously implemented features. Every single novel feature is tested with a set of test cases implemented in Python3 programming language. These tests are then run by RetDec regression tests framework available at official GitHub repository [11].

Listing 7.1 depicts how tests are implemented.

```
class TestTypeRefHashDefault(Test):
    settings = TestSettings(
        tool='fileinfo',
        input='typeref_hash_default',
        args='--verbose --json'
    )

    def test_correctly_computes_typeref_hash(self):
        assert self.fileinfo.succeeded

        self.assertEqual(self.fileinfo.output['dotnetInfo']['typeRefTable']
            ['crc32'], 'bb390cc9')
        self.assertEqual(self.fileinfo.output['dotnetInfo']['typeRefTable']
            ['md5'], '93b7f964c87a94b07d1f6171f0b7d7c1')
```

Listing 7.1: Regression test case implementation

7.2 Statistics

Design of some features presented in this work was conditioned on their prevalence in malware. More specifically, format (DIB vs PNG), compression status, dimensions, and other properties of malware icons needed to be known before implementing of icon features. For this reason, a Python tool for statistical analysis of malware icons was designed. Produced statistical results can be observed in Table 7.1. The table shows frequency of property values among all analyzed malware samples and frequency among malware that contained an DIB icon in percentage.

Property	Value	All MW freq	DIB icon MW freq
icon present	true	49.71%	-
icon present	false	50.28%	-
icon format	DIB	49.35%	100.00%
width/height	32	38.19%	77.39%
width/height	48	6.98%	14.14%
width/height	16	2.76%	5.58%
width/height	64	0.41%	0.84%
width/height	47	0.4%	0.81%
color depth	32	24.86%	50.78%
color depth	4	11.18%	22.65%
color depth	8	7.84%	15.93%
color depth	24	4.94%	10.01%
compression	none	49.35%	100.00%

Table 7.1: Icon statistics among malware

Dataset for the analysis was constructed from 50.000 random PE malware samples obtained from VirusTotal[1] database with VirusTotal intelligence downloader tool as follows:

```
$ python2.7 vt_intelligence_downloader.py -n 5
    'type:peexe avast:infected nod32:infected kaspersky:infected'
```

Exact results of icon analysis is can be found in Appendix B.

Another tool was implemented to gather information about COM Visual Basic applications. The results are to be found in Table 7.2

Property	Value	DIB icon MW freq
COM Info structure present	true	11.64%

Table 7.2: COM Visual Basic statistics among malware

Filtering of applications based on use of Visual Basic is not supported in VirusTotal intelligence downloader, therefore a dataset for COM Visual Basic applications was constructed from proprietary malware collection of Avast company. A random sample was taken from each of 11.000 malware clusters where Visual Basic was the detected language.

[1] VirusTotal is a website service aggregating many antivirus products and providing an enormous malware dataset

7.3 Integration, Deployment and Feature Efficiency Evaluation

As stated earlier, FileInfo is used by an internal clustering system Clusty at Avast. Clusty divides real world malware into clusters based on features extracted by FileInfo. Statistics presented here are: size of clusters, detection rate (by Avast), classification as malware, potentially unwanted product, clean and unknown classes (also by Avast), and number of features found in each clustered sample.

TypeRef features are now used in production release of Clusty. Table 7.3 shows clusters of malware captured in one month period clustered solely by TypeRef Table MD5 hash. The clusters are massive and false positive rates are nearly negligible. TypeRef hash is a valuable feature as some of clustered samples barely share any other features.

Samples	Detection	MW	PUP	UNKNW	CLEAN	Shared
2.3M	99%	97%	0%	2%	1%	3
66K	100%	0%	98%	1%	1%	6
43K	1%	1%	0%	70%	29%	2

Table 7.3: TypeRef Table MD5 hash clusters (production release)

Icon MD5 hash is used in production release of Clusty as well. Table 7.4 shows samples clustered based on icon MD5 hash.

Samples	Detection	MW	PUP	UNKNW	CLEAN	Shared
23K	100%	99%	0%	1%	0%	3
21K	100%	99%	0%	1%	0%	2
8K	100%	99%	0%	1%	0%	2

Table 7.4: Icon MD5 hash clusters (production release)

Dataset of 11.000 PE malware samples with icons from 20 clusters were reclustered based on Icon MD5 and Icon AverageHash as shown in Table 7.5 and Table 7.6. Integration of Icon AverageHash into Clusty is problematic, therefore icons are tested for exact match of average hashes rather than the similarity metric designed in Section 5.2.2. Despite this, AverageHash is more generalizing and overall effective.

Samples	Detection	MW	PUP	UNKNW	CLEAN	Shared
951	100%	100%	0%	0%	0%	4
933	100%	100%	0%	0%	0%	4

Table 7.5: Icon MD5 hash clusters

Samples	Detection	MW	PUP	UNKNW	CLEAN	Shared
1171	100%	100%	0%	0%	0%	4
933	100%	100%	0%	0%	0%	4

Table 7.6: Icon AverageHash clusters

Dataset of 10.000 Visual Basic malware samples was constructed from 2.000 clusters. Clusters created based on Object Table MD5 hash can be seen in Table 7.7 and External

Table MD5 clusters are in Table 7.8. Object and External Tables are unreliable as previously stated, thus some clusters are constructed incorrectly.

Samples	Detection	MW	PUP	UNKNW	CLEAN	Shared
200	53%	53%	0%	13%	34%	2
95	73%	71%	1%	8%	20%	2
81	100%	100%	0%	0%	0%	15
80	100%	100%	0%	0%	0%	7

Table 7.7: Visual Basic Object Table clusters

Samples	Detection	MW	PUP	UNKNW	CLEAN	Shared
248	100%	95%	5%	0%	0%	2
81	100%	100%	0%	0%	0%	15
80	100%	100%	0%	0%	0%	7
57	69%	64%	0%	10%	26%	2

Table 7.8: Visual Basic External Table clusters

The same Visual Basic samples were clustered based on most promising Visual Basic CLSIDs. TypeLib CLSID clusters are shown in Table 7.9 and COM Object CLSID clusters are presented in Table 7.10.

Samples	Detection	MW	PUP	UNKNW	CLEAN	Shared
81	100%	100%	0%	0%	0%	15
80	100%	100%	0%	0%	0%	7
40	100%	100%	0%	0%	0%	15

Table 7.9: Visual Basic External Table clusters

Samples	Detection	MW	PUP	UNKNW	CLEAN	Shared
30	100%	97%	0%	3%	0%	9
10	100%	100%	0%	0%	0%	11

Table 7.10: Visual Basic External Table clusters

Innovated version of FileInfo is now being integrated into production version of Clusty. Further, it is planned to be integrated in newly designed internal systems of Avast company.

Chapter 8

Conclusion

This work focuses on static analysis of malicious binary executable software. The PE binary executable format is discussed in more detail. Goals declared in the first chapter are fulfilled. A multitude of features is designed, implemented, and tested on real world malware. The most promising features are tested by Avast clustering system as an extension of the original thesis assignment.

The symbol importing system of .NET binaries was analyzed, TypeRef Table reconstructed, and its features were produced. Clustering based on TypeRef MD5 hash turned out to be extremely effective. Binary files captured by Avast company within the span of a month were divided into clusters of over 2.3 million samples with 99% accuracy.

Visual Basic applications were dissected and their metadata including project and language related information extracted. Object Table and External Table were reconstructed and their features produced. As an addition to the original thesis assignment, statistics about COM data obtained from 11.000 analyzed Visual Basic malware samples were produced. Visual Basic GUIDs and CLSIDs and their persistence after recompilation was analyzed as another addition to the original thesis assignment. Over 11.000 malware samples were clustered based on Object Table MD5 hash, External Table MD5 hash, COM Object CLSID, and TypeLib CLSID.

Windows 10 desktop environment was analyzed for its choice of the main icon based on dimensions and color depths. As an addition to the original thesis assignment, a statistical analysis of icon properties of over 50.000 malware samples was done. An icon priority list was constructed and the main icon extracted. Cryptographic and perceptual hashes of the main icon were produced. Dataset of 11.000 malware samples with icons was clustered based on MD5 and AverageHash. AverageHash showed better results despite the fact that it is compared for exact match.

A PE anomaly scanner was implemented, section and overlay entropy computed, thread-local storage information extracted, and project related strings and language identifiers were obtained.

All features are implemented in feature extraction tool FileInfo. The innovated version of FileInfo is now a part of the internal Avast clustering system, an open source decompiler RetDec and it is used as a general purpose malware analysis tool.

This work primarily focuses on PE file format, however malware targets various platforms such as Linux, MacOS, or Android. Future work should involve extraction of more format specific features and an in depth analysis of overlay data to estimate its content.

The work was presented and demonstrated at a student's conference Excel@FIT.

Bibliography

[1] Microsoft: Microsoft Portable Executable and Common Object File Format Specification. Retreived from https://docs.microsoft.com/en-us/windows/desktop/Debug/pe-format. accessed: 2019-05-13.

[2] AV-TEST: Malware statistics 2018. Retreived from https://www.av-test.org/en/statistics/malware/. 2018. accessed: 2019-05-13.

[3] Microsoft: BITMAPINFO structure. Retreived from https://docs.microsoft.com/en-us/windows/desktop/api/wingdi/ns-wingdi-tagbitmapinfo. 2018. accessed: 2019-05-13.

[4] Microsoft: Code Page Identifiers. Retreived from https://docs.microsoft.com/en-us/windows/desktop/Intl/code-page-identifiers. 2018. accessed: 2019-05-13.

[5] Microsoft: String structure. Retreived from https://docs.microsoft.com/en-us/windows/desktop/menurc/string-str. 2018. accessed: 2019-05-13.

[6] Microsoft: UX guide - Icons. Retreived from https://docs.microsoft.com/en-us/windows/desktop/uxguide/vis-icons. 2018. accessed: 2019-05-13.

[7] Microsoft: Visual Basic Guide. Retreived from https://docs.microsoft.com/en-us/dotnet/visual-basic/. 2018. accessed: 2019-05-13.

[8] Microsoft: Windows Language Code Identifier (LCID) Reference. Retreived from https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-lcid/70feba9f-294e-491e-b6eb-56532684c37f. 2019. accessed: 2019-05-13.

[9] Avast: RetDec: A retargetable machine-code decompiler based on LLVM. Retreived from https://github.com/avast/retdec. 2019. accessed: 2019-05-13.

[10] Avast: RetDec Regression Tests. Retreived from https://github.com/avast/retdec-regression-tests. 2019. accessed: 2019-05-13.

[11] Avast: RetDec Regression Tests Framework. Retreived from https://github.com/avast/retdec-regression-tests-framework. 2019. accessed: 2019-05-13.

[12] Chen, R.: The evolution of the ICO file format. Retreived from https://blogs.msdn.microsoft.com/oldnewthing/20101018-00/?p=12513/. 2010. accessed: 2019-05-13.

[13] Chen, R.: The format of icon resources. Retreived from https://blogs.msdn.microsoft.com/oldnewthing/20120720-00/?p=7083. 2012. accessed: 2019-05-13.

[14] Chikofsky, E. J.; Cross, J. H.: Reverse engineering and design recovery: A taxonomy. *IEEE software*. vol. 7, no. 1. 1990: pp. 13–17.

[15] Geddon, A.: Visual Basic reversed - A decompiling approach. Retreived from http://sandsprite.com/vb-reversing/files/VISUAL%20BASIC%20REVERSED.pdf. accessed: 2019-05-13.

[16] Hahn, K.: Robust static analysis of portable executable malware. *Mater Thesis, HTWK Leipzig*. 2014.

[17] He, J.; Ivanov, P.; Tsankov, P.; et al.: Debin: Predicting Debug Information in Stripped Binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018. pp. 1667–1680.

[18] Hexacorn: PE Section names. Retreived from http://www.hexacorn.com/blog/2016/12/15/pe-section-names-re-visited/. 2016. accessed: 2019-05-13.

[19] Hornick, J.: Microsoft: Icons. Retreived from https://msdn.microsoft.com/en-us/library/ms997538.aspx. 1995. accessed: 2019-05-13.

[20] Ionescu, A.: Visual Basic Image Internal Structure Format. Retreived from http://sandsprite.com/vb-reversing/files/Alex_Ionescu_vb_structures.pdf. 2004. accessed: 2019-05-13.

[21] James Dickson Murray, W. V.: O'Reilly: Encyclopedia of Graphics File Formats. Retreived from https://www.fileformat.info/format/bmp/egff.htm. 1996. accessed: 2019-05-13.

[22] Krawetz, N.: Looks Like It. Retreived from http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html. 2011. accessed: 2019-05-13.

[23] Milkovič, M.: Systém pro detekci vzorů v binárních souborech. Brno University of Technology. 2017.

[24] Peter Matula, M. M.: RetDec - An Open-Source Machine-Code Decompiler. Retreived from https://retdec.com/static/publications/retdec-slides-recon-2018.pdf. 2018. accessed: 2019-05-13.

[25] Pistelli, D.: The .NET File Format. Retreived from https://www.ntcore.com/files/dotnetformat.htm. 2005. accessed: 2019-05-13.

[26] Plachy, J.: Portable executable file format. Retreived from http://skynet.ie/~caolan/pub/winresdump/winresdump/doc/pefile.html. 1997. accessed: 2019-05-13.

[27] Porst, S.: PeLib: An open-source C++ library to modify PE files. Retreived from http://www.pelib.com/index.php. 2005. accessed: 2019-05-13.

[28] Silver, M.: VB P-code Information. Retreived from http://sandsprite.com/vb-reversing/files/VB%20P-code%20Information%20by%20Mr%20Silver.html. accessed: 2019-05-13.

Appendix A

Format of Binary Data Structures

Multitude of structures are mentioned in this work. Excerpt of the most relevant ones are presented here. All of the structures were adopted from cited sources.

A.1 Icon and IconGroup Resources

Offset	Size	Icon Group Directory [12]
0x0	0x2	reserved (must be 0)
0x2	0x2	type (must be 1)
0x4	0x2	number of entries
0x6	*	entries array

Offset	Size	Icon Group Entry [12] [13]
0x0	0x1	icon width
0x1	0x1	icon height
0x2	0x1	color count
0x3	0x1	reserved (must be 0)
0x4	0x2	planes
0x6	0x2	bit count
0x8	0x4	size of icon in bytes
0xC	0x2	icon name identifier

Table A.1: IconGroup resource structure

Offset	Size	BITMAPINFOHEADER [21]
0x0	4	header size (40 bytes)
0x4	4	pixel width (signed)
0x8	4	pixel height (signed)
0xC	2	number of color planes (must be 1)
0xE	2	color depth; bits per pixel
0x10	4	compression method
0x14	4	image size in bytes (optionaly set)
0x18	4	horizontal resolution
0x1C	4	vertical resolution
0x20	4	color palette size (0 or 2^n)
0x24	4	number of colors used (ignored)

Table A.2: BITMAPINFOHEADER structure format

A.2 .NET Values

Value	ResolutionScope Tags [25]
0x0	Module Table
0x1	ModuleRef Table
0x2	AssemblyRef Table
0x3	TypeRef Table

Table A.3: ResolutionScope Tags

A.3 Visual Basic Structures

Offset	Size	VB Header [20]
0x0	0x4	signature („VB5!“)
0x6	0xE	language DLL
0x14	0xE	backup language DLL
0x24	0x4	primary LCID
0x28	0x4	backup LCID
0x2C	0x4	sub main code address (VB entry point)
0x30	0x4	Project Info structure address
0x54	0x4	COM Register Data structure address
0x58	0x4	offset to string containing EXE filename
0x5C	0x4	offset to string containing project's description
0x60	0x4	offset to string containing name of the Help file
0x64	0x4	offset to string containing project's name

Table A.4: VB Header structure format

Offset	Size	Project Info [20]
0x4	0x4	object Table address
0x20	0x4	native code address
0x24	0x210	project path in unicode
0x234	0x4	external Table address
0x238	0x4	number of externals (imports)

Table A.5: Project Info structure format

Offset	Size	External Table entry [15]
0x0	0x4	Import type (0x6 internal, 0x7 external)
0x4	0x4	Inside Import Data address/External Entry Data Address

Offset	Size	External Entry Data [15]
0x0	0x4	External module name address
0x0	0x4	External function name address

Table A.6: External Table structure format

Offset	Size	Object Table [20]
0x18	0x16	Object table GUID
0x2A	0x2	number of objects
0x30	0x4	Public Object Descriptors address
0x40	0x4	project name address
0x44	0x4	LCID
0x48	0x4	backup LCID

Table A.7: Object Table structure format

Offset	Size	Public Object Descriptors [20]
0x18	0x4	Object name address
0x1C	0x4	Number of methods
0x20	0x4	Method names array address

Table A.8: Public Object Descriptors structure format

Offset	Size	COM Register Data [20]
0x0	0x4	Offset to COM Register Info
0x4	0x4	Offset to project name
0x10	0x10	CLSID of TypeLib
0x20	0x4	LCID of TypeLib

Table A.9: COM Register Data structure format

Offset	Size	COM Register Info [20]
0x4	0x4	Offset to object name
0x14	0x10	CLSID of object
0x39	0x2	object type

Table A.10: COM Register Info structure format

A.4 Thread-local Storage Directory

Offset	Size	TLS directory [1]
0x0	0x4	Start address of TLS template
0x4	0x4	End address of TLS template
0xC	0x4	Address of TLS callbacks
0x14	0x4	Characteristics

Table A.11: TLS directory structure format

Appendix B

Statistical Analysis of Malware Icons

In Table B.1, there are results of statistical analysis done on 50.000 random PE VirusTotal malware samples. Table. The frequency of property values among all analyzed malware samples is represented by the number of samples containing the value. Presented properties are those of DIB header (BITMAPINFOHEADER) A.2. Values with minor prevalence were removed.

Property	Value	All MW freq	Property	Value	All MW freq
no icon:	-	25144	compression:	0	24675
not BMP:	-	259		0	7604
nonstand. dim.:	-	399	bitmapSize:	4224	3414
size:	40	24675		640	3058
	32	19095		4096	2063
width/height:	48	3489		3072	1722
	16	1378		1152	1403
	64	207		1024	862
	47	199		9600	774
	128	83		512	653
	24	80		192	606
	256	35		0	23722
	40	23	horizontalRes:	2835	485
	96	13		0	23722
	58	10	verticalRes:	2835	485
	1	24674		0	21220
planes:	0	1	colorsUsed:	256	2340
	32	12431		16	1111
bitCount:	4	5590		0	23633
	8	3930	colorImportant:	256	954
	24	2470		16	85
	1	250			
	16	3			
	0	1			

Table B.1: Complete icon statistics among malware