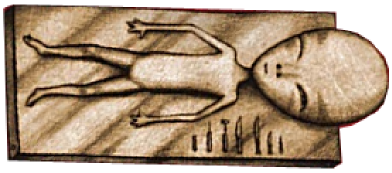
BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2019
http://excel.fit.vutbr.cz

Design of Binary File Features for Malware Classification

Jakub Pružinec


[1]

[2]

[3]

Abstract
Rapid and widespread adoption of information technologies lead to userbase diversification and their use among laymen. Due to this, we witness malicious software evolve and grow larger day by day endangering data of billions of users. Today's anti-malware companies seek automated solutions for malware detection. One of possible approaches to malware identification is to use artificial intelligence to classify it. Precision of malware classification is heavily dependent on available information about classified samples - features. Poor design of features may result in wrongly classifying legitimate software as malware, or even worse, to let malware slip by undetected. This article focuses on design, extraction, reliability and efficiency testing of static binary malware features. Moreover, malware feature extraction tool, FileInfo, is innovated. Work is done in cooperation with Avast company, where FileInfo is used in malware clustering system, binary file decompiler and as a general purpose static analysis tool on a daily basis.

Keywords: Binary malware analysis — Static analysis — Classification — Features — Reverse engineering

Supplementary Material: FileInfo source code

*j.pruzinec@gmail.com, xpruzi02@fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Due to modern world digitalization, property, privacy, and identity of people became dependent on information technologies. Unfortunately, crime has adopted digital nature as well. AV-tests institute recorded over 137 millions of malicious software samples in 2018 [4]. Cybersecurity became crucial to information technology vendors as they are responsible for safety of their clients. As a consequence of such frequent occurrence of malicious software, manual analysis of all samples is practically impossible. Therefore, automated malware classification is an absolute necessity when fighting today's malware.

Unsurprisingly, it is in great interest of malware to evade analysis and deceive classifiers to remain undetected. Evolution of malware and innovation of its obscure evasion methods may result in its missclassification. On the other hand, the fact that behavior and shape of malware often differ significantly from legitimate software is crucial for malware classification. To classify malware, numerous features are extracted from analyzed samples first. Designed features are general enough to reflect similarity between malware samples and fitting enough to distinguish between malicious from legitimate software. What's more, feature design has significant impact on time efficiency of classification algorithms, as it is determined by amount of features taken into consideration (dimensions). Quality of designed features used in a classifier can be estimated by ratio of correctly to incorrectly classified samples.

Despite significant work done in the field of static analysis of malware, most freely available tools are not suitable for malware classification. As a master thesis, Katja Hahn developed a complex malware static analysis tool, *PortEx* [5]. PortEx is a robust open-source library for dissection of binary malware. Unfortunately, PortEx is implemented in Java, thus potentially too slow to be used for malware classification. Further, PortEx supports only Windows binary file format and lacks support of most features presented later in this article.

FileInfo is a complex open-source feature extraction tool developed by Avast company [6]. Contrary to PortEx, FileInfo is implemented in C++ and supports numerous file formats like PE (Windows), ELF (Linux) and MACHO (Apple). Additionally, FileInfo supports extraction of framework based features, e.g. .NET features. Despite its complexity, FileInfo lacks support of features necessary for classification of ever-evolving malware presented hereafter.

This article primarily focuses on (but is not limited to) Windows binary file format static features. Goal of this work is to:

- produce cryptographic and perceptual hashes of program icons
- reconstruct TypeRef tables of .NET binaries and produce their cryptographic hashes
- parse, reconstruct and produce cryptographic hashes of several VisualBasic metadata structures
- compute section and overlay entropy as an indication of packed data
- extract information about product version, supported languages, trademark, copyright, original name and more
- determine addresses of thread-local variable initialization routines

Designed features play a major role when classifying stealthy malware. Malware implementing following analysis evasion methods can be now classified:

- conventional way of importing of external symbols substituted with .NET or VisualBasic symbol importing system
- distortion of icon to corrupt their cryptographic hashes
- packing of sections and overlay to evade static analysis

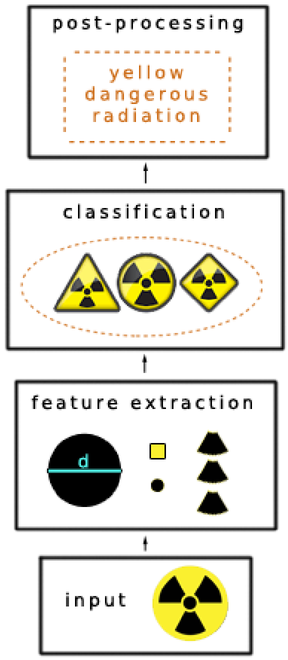


Figure 1. Classification process

Section 2 describes classification process in general, Sections 3, 4 and 5 discuss topic-based features. Additional features are presented in Section 6.

2. Malware Classification

Malware classification process starts with feature extraction from analyzed samples. These features can be of different nature, for example imported symbols or sequence of kernel calls. Afterwards, the classifier assigns classes to input samples based on previously extracted features. As a final step, information about the classes themselves is extracted. Classification process is depicted in Figure 1. Obviously, precision of classification is heavily dependent on extracted features. Features need to be discriminative, meaning the classifier has to be able to distinguish between classes based on them. Poorly designed features may result in classification of legitimate software as malware or the other way around.

Besides proper classification, well designed features may significantly increase time efficiency of classification algorithms. Time complexity of classification algorithms is dependent on number of features taken into consideration.

Features need to be general enough to reflect similarity of malware but should not be too general as it could affect their discriminability. In this context, information about imported libraries is much more valuable than observation that an analyzed sample contains some code. On the other hand, features need to be fitting enough to provide detailed information about

Figure 2. Icon modified with noise to evade classification based on exact match of cryptographic icon hash

classified malware. Overfitting features may suppress detection of similarity between analyzed samples. As an example, information about original file name can be made use of, but knowing that third letter of the name is ‘X’ alone, is useless. Following sections are dedicated to designed features.

3. Icon Features

From 50,000 PE malware samples, over 24,000 contained an icon. Malware authors often add icons to attract and deceive their victims. They often keep icons almost unchanged when updating or modifying malware. Because of this, icon hashes can serve as good features. Cryptographic hashes work well when testing icons for exact match, however a slightest change in icon data results in complete change of its cryptographic hash.

Malware authors are aware of this fact and add noise to their icons as can be seen in Figure 2. To bypass this inconvenience, icons have to parsed into internal representation. Parsing of icons is a lengthy and rigorous process that will be described in detail in my bachelor thesis. Simply put, in PE file format icons are embedded into a structure called *resource tree*. Despite the fact that PE binaries can contain multiple icons, only one *main icon* is shown in a desktop environment.

Main icon cannot be determined with certainty since the icon to be shown is dependent on desktop environment properties, such as DPI. Main icon detection algorithm will be described in my bachelor thesis as well.

Icon data itself has to be parsed after the main icon has been extracted from resource tree. Icons are stored in *DIB* file format. DIB icons of all supported color depths are parsed into uniform representation. Example in Figure 3 demonstrates how a 4 bpp DIB icon is parsed into two dimensional pixel array.

When main icon is parsed, one can produce its *perceptual hashes*. Perceptual hashes are hashes representing images in a way that can be tested for similarity. Contrary to cryptographic hashes, slight change of hashed image results only in slight change of its perceptual hash. One such hash is called *Average hash*. Principle of Average hash computation is shown in

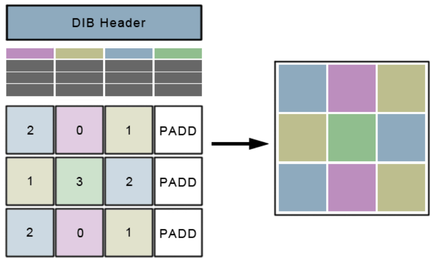


Figure 3. DIB with 4bit color depth parsed into uniform representation

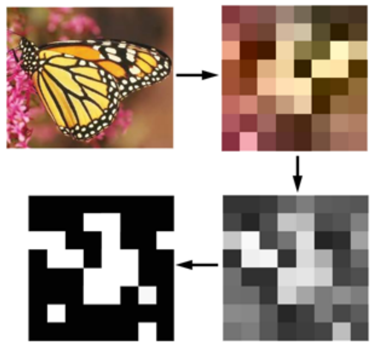


Figure 4. Average hash computation

Figure 4. Image is first resized to 8×8 dimensions to filter out image details. Image is then converted to greyscale and to black and white afterwards. Each of 8 rows of the newly created image is a sequence 8 black (0) or white (1) pixels. Therefore, 8 rows of new image form an 8 byte long average hash. Similarity of two images can be determined by *Hamming distance* of their Average hashes. If the distance is less than 3, images are considered to be similar.

Both cryptographic and perceptual hashes are presented in Listing 1.

Listing 1. Icon hashes

```
resourceTable: {
  "iconAvgHash" : "b7478387b4ffaeff",
  "iconCrc32" : "c6009c34",
  "iconMd5" :"8b6fdb44e0b3e55bf9bc8f-
             fda1800b79",
  "iconSha256" :
    "4d8b8a948b29bf38cd8f186e75b58-
    ed5017ed11aac7ebeb453752181b58f3bea",
  ...
}
```

4. .NET Features

.NET is an open-source framework developed by Microsoft. .NET programs are compiled in PE file format as data, metadata and bytecode interpreted by virtual machine.

FileInfo already supports reconstruction of some .NET structures but omits information about imported classes. This information is crucial when classifying malware based on imported symbols, because

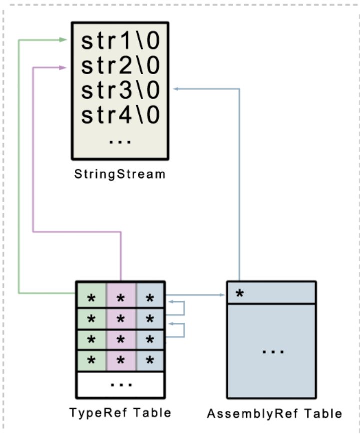


Figure 5. TypeRef table structure

.NET does not import most of its external functionality through conventional *PE import table*. Classes are imported through so called *TypeRef table*. Each entry of a TypeRef table may contain information about origin of an imported class such as *name*, *module type*, *library name* and *namespace*. Further, .NET classes may be nested, meaning one class can be defined inside of another class. On binary level, TypeRef entry of child class does not reference an external module but rather TypeRef entry of a parent class. Such situation is demonstrated on an example in Figure 5. Reconstruction of a TypeRef table takes several steps: parsing, linking, and presentation. During the linking process of a parsed parent/child TypeRef records, entries are checked for cyclic references. Cyclic references are exclusively a product of manual modification of a TypeRef table and may result in infinite table processing. To prevent this, one of the references in cycle is simply ignored and remains detached. After reconstruction of a TypeRef table, its cryptographic hashes are produced.

Listing 2 shows the first element in a reconstructed form of a TypeRef table.

Listing 2. TypeRef table features

```
"typeRefTable" : [
  { "libraryName" :
      "System.Runtime",
    "name" :
      "CompilationRelaxationsAttribute",
    "nameSpace" :
      "System.Runtime.CompilerServices"},
]
```

5. VisualBasic Features

Despite the claim that *VisualBasic* programs “benefit from security” [7], today they are almost exclusively created by malware authors. Same as .NET, VisualBasic is interpreted by a virtual machine, thus implements its own importing system besides *PE import table*.

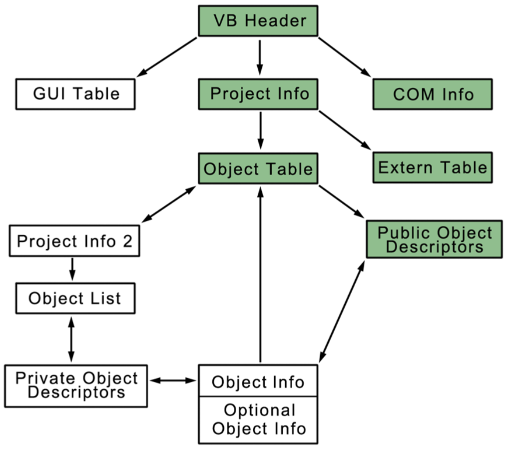


Figure 6. VisualBasic

VisualBasic format is closed-source, therefore it is easier for VisualBasic malware to hide its intentions. On the other hand, VisuaBasic applications are rich in metadata. Alex Ionescu has done remarkable work on reversing the metadata [8] and therefore lots of previously hidden VisualBasic malware can be now classified. Metadata includes information about *project names*, *identifiers*, *languages*, *imported functions*, *objects* and more. This information is spread across numerous structures referencing each other as shown in Figure 6.

To extract the necessary data, a top-down parser is implemented. For the sake of receiving names of imported functions, an external table is reconstructed. Further, an object table is reconstructed to obtain names and method names of implemented objects. External table and object table are hashed and all auxiliary features are presented.

Small excerpt of VisualBasic features is presented in Listing 3. It can be assumed that malware targets Facebook users and is written by an author of Swedish origin.

Listing 3. VisualBasic features excerpt

```
// imported functions
"externs" : [
  { "apiName" : "PlgBlt",
    "moduleName" : "gdi32" },
],
// implemented objects
"objects" : [
  { "name" : "frmMain"
    "methods" : [
      "C_Mutex",
      "BROWSER_FB_DocumentComplete",
      "BROWSER_FB_OnQuit",
      "FACEBOOK_START"},
],
```

```
// additional
"projectPath" :
  "C:\\Users\\Admin\\Desktop_old\\
  Blackshades project\\Blackshades
  NET\\server\\server.vbp",
"projectPrimaryLCID" :
  "English - United States",
"projectSecondaryLCID" :
  "Swedish - Sweden"
```

6. Other Features

One of commonly used methods to avoid static analysis malware authors use is to *pack* their program. 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. On the other hand, packed binary can be a strong indication of malicious intentions. For this reason, entropy of sections and overlay is computed. High data entropy indicates compression, thus can serve to detect packed data. Contrary, low entropy indicates small data diversity and can be used to detect blank data sections.

Listing 4 demonstrates detection of section compression status.

Listing 4. Section entropy

```
"sections" : [
  // packed
  {  name: ".text",
     entropy: "7.8632" },
  // normal
  {  name: ".rodata",
     entropy: "4.3242" },
  // empty
  {  name: ".fini_array",
     entropy: "0.8632" },
]
```

Thread-local data initialization routines are abused by malware authors to evade static analysis. Thread data is stored in dedicated directory in PE file format. This data needs to be initialized before entry point execution. In other words, thread-local data initializers are called before the *main()* function. Malware often implements its malicious behavior in one of thread-local data initializers for this reason. Intuitive investigation of *main()* function is useless in this case. What's more, many debuggers set breakpoints on entry point, therefore thread-local data initialization routines may infect host machine before the analyst has a chance to intervene.

Addresses are presented as shown in Listing 5.

Listing 5. Thread-local data intializers

```
// thread-local initialization routines
"threadLocalInitializers" : [
  "0x401060",
  "0x4010a0"
],
```

Besides the data necessary for proper program execution, compilers often add additional information about the product. Such information can be retrieved from *VersionInfo* resource tree entry in PE file format. This entry holds information about *supported languages*, *legal copyright*, *original file name*, *version*, *timestamps* and more.

Part of information extracted from VersionInfo of a malware is to be found in Listing 6. This information suggest that the author was of German origin. The author probably instrumented a Firefox executable built on 2010/09/14.

Listing 6. VersionInfo features

```
"versionInfo" : {
  "languages" : [
    { "codePage" : "utf-16",
      "lcid" : "German - Germany" }
  ],
  "strings" : [
    { "name" : "CompanyName",
      "value" : "obama" },
    { "name" : "OriginalFilename",
      "value" : "my_st0re_lo-
       ader______.exe" },
    { "name" : "ProductName",
      "value" : "Firefox" },
    { "name" : "BuildID",
      "value" : "20100914121323"
    }
  ]
}
```

7. Conclusions

Design of features and their impact on malware classification have been discussed in this article. Further, some static analysis evasion approaches and methods to deal with them were presented. Following contributions were made:

- design and extraction of .NET TypeRef table features
- design and extraction of icon features
- design and extraction of VisualBasic metadata features
- design and extraction of VersionInfo features
- design and extraction of entropy and thread-local directory features

So far, only icon and .NET features were integrated into Avast malware clustering system. In Table 1 and

Table 2 are statistics of clusters classified as malware based solely on a given feature. Beside the size of a cluster, detected malware ratio can be seen. Further, malware detection ratio of 1000 randomly selected samples from clusters analyzed by ESET and Kaspersky antivirus software is present.

N samples	Detected	ESET	Kaspersky
406	97%	95%	100%
221	96%	100%	100%
50	96%	100%	100%

Table 1. Icon MD5 based clusters

N samples	Detected	ESET	Kaspersky
1.3M	96%	100%	100%
63K	98%	-	82%
13K	98%	99%	100%

Table 2. TypeRef MD5 based clusters

In close future, newly designed features implemented in FileInfo will be integrated into the clustering system and RetDec decompiler developed by Avast company [6]. FileInfo implements an in depth analysis of the PE file format prevalently and soon it will be forced to respond to increasing occurrence of Linux and macOS malware with extraction of new features regarding ELF and MACHO file formats.

Further, FileInfo has great potential for improvement of data recognition features, such as overlay format detection. Besides that, features regarding file format violations should be implemented, as they serve as good indication of malicious behavior. Some of the problems mentioned here will be addressed in my bachelor thesis soon.

Acknowledgments

I would like to thank my supervisors Doc. Dr. Ing. Dušan Kolář, Ing. Marek Milkovič and Ing. Jakub Křoustek, PhD for help and support throughout the whole project.

References

[1] MBIS. Cyber forensic - scientific investigations. https://www.mbis-inc.net/cyber-forensic-services.html. [Online; accessed April 6, 2019].

[2] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[3] Warren Perez Araya. Is malware analysis right for your business? https://securityintelligence.com/is-malware-analysis-right-for-your-business/, 2018. [Online; accessed April 6, 2019].

[4] Av-test institute: Malware statistics 2018. https://www.av-test.org/en/statistics/malware/. [Online; accessed April 6, 2019].

[5] Katja Hahn. Robust static analysis of portable executable malware. *Mater Thesis, HTWK Leipzig*, 2014.

[6] J. Křoustek, P. Matula, and M. Milkovič. An open-source machine-code decompiler. https://retdec.com/static/publications/retdec-slides-recon-2018.pdf, 2018. [Online; accessed April 6, 2019].

[7] Microsoft corporation: Visual basic guide. https://docs.microsoft.com/en-us/dotnet/visual-basic/, 2018. [Online; accessed April 6, 2019].

[8] Alex Ionescu. *Visual Basic: Image Internal Structure Format*. Relsoft Technologies, 2004.