TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Computer Science

UT Centre for Digital Forensics and Cyber Security

ITC70LT

Vjatšeslav Panov

# IMPLEMENTATION OF A HASH FUNCTION FOR PORTABLE EXECUTABLE BASED ON STRUCTURAL INFORMATION

Master's thesis

Supervisor: Truls Ringkob

Supervisor's degree: TUT master's degree

Supervisor's position: IT College and visiting TUT lecturer

Tallinn 2016

# Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Vjatšeslav Panov

30.04.2016

# Abstract

Malware is big security threat on the Internet in current epoch. Anti-virus and security companies receive large number of malware samples every day. Malware samples need to be classified and grouped for further analysis. There are different types of malware analysis, clustering and classification methods available.

This thesis focuses on static analysis of malware by implementation of hash function for Portable Executable based on structural information of file introduced by Georg Wicherski from RWTH Aachen University. Main idea is to use selected properties of file headers and complexity of section contents in PE file for hash. Even if algorithm was described, example of implementation was given in pseudo code. Source code from author was never published. In addition, it does not have a strict definition and examples of test usage.

Goals of this work are trying to correctly implement pehash algorithm in python programming language, make a strict format definition, analyze and adapt algorithm for latest PE file examples and make it public with example of usage.

This thesis is written in English and is 47 pages long, including 6 chapters, 11 figures and 21 tables.

# Annotatsioon

**Räsifunktsiooni juurutamine kaasaskantav käivitatava failil struktuurse informatsiooni baasil.**

Õelvara Internetis on meie ajastu suureks ohuks. Viiruse-tõrje ja IT-turvalisuse firmad iga päev saavad suure hulka kahjuvara näiteid. Järgse analüüsi õelvara pead olema klassifitseeritud ning grupeeritud. õelvara analüüsimiseks olemas erinevad meetodid, näiteks grupeerimine ja klassifitseerimine.

Antud diplomitöö fokuseeritud õelvara staatilise analüüsile räsi funktsiooni juurutamisel kaasaskantav käivitatava failile struktuurse informatsiooni põhjal mis on kehtestatud Georg Wicherski poolt RWTH Aachen Ülikoolist. Põhi idee on kasutada failipäise valitud omadused ja PE faili sektsiooni sisu keerukust räsi jaoks. Isegi kui algoritm oli kirjeldatud, juurutamise näite oli rakendatud pseudokoodis. Autor pole kunagi lähtekoodi avaldanud. Samuti definitsioon ja test kastutamine ei ole rangelt määratud.

Töö eesmärgiks on katsetada õigesti juurutada PE räsi algoritmi Python programmeerimiskeeles, rangelt määrata definitsiooni, analüüsida ja kohandada algoritmi hiljemalt PE faili näited ning avalikustada..

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 47 leheküljel, 6 peatükki, 11 joonist, 21 tabelit.

# Table of abbreviations and terms

| Abbreviation | Definition |
|---|---|
| COFF | Common Object File Format |
| MD5 | Message Digest Algorithm |
| PE | Portable Executable |
| PE+ | Portable Executable 64 bits version |
| SHA1 | Secure Hash Algorithm 1 |
| SHA256 | Secure Hash Algorithm 256 |
| VA | Virtual Address |

# Table of contents

# List of figures

# List of tables

# 1. Introduction

As described in "Symantec 2016 Internet Security Threat Report Vol. 21" [1] which provides an overview and analysis of the year in global threat activity, Symantec discovered more than 430 million new unique pieces of malware in 2015, up 36 percent from the year before. As real life and online become indistinguishable from each other, cybercrime has become a part of our daily lives.

Every day of the week, there are thousands of new viruses annoying to put in danger your computer. However, from other side, rarely someone writes new malware from scratch. Most frequently, they are so named mutating and polymorphic malware or the new malware is just some transformation of previously written malware, thus it is usually similar to its predecessor. Finding this similarity is like to malware classification, which allows detecting malware family and then effectively finding the activities for it.

Nowadays we have different hash types for executable files like classical MD5, SHA-1, and SHA-256. More interesting for malware analyze are hash functions based not directly on the file's bits stream but on selected structural information fields of executable file. Most public well-known hashes of such type are "peHash" and "Import Hash".

"Import Hash" (imphash) was created and implemented by Mandiant and is a hash based on library/API names and their specific order within the executable. [2] It has a good definition and is implemented in python language and submitted as a patch to Ero Carrera's pefile. [3] It enables the calculation of the imphash value for a given PE. Imphash used in VirusTotal [4] and in lots of open-source malware analysis frameworks and utilities [5].

"peHash" (pehash) was introduced by Georg Wicherski in the paper "peHash: A Novel Approach to Fast Malware Clustering" [6]. “peHash” is a function for binaries in the Portable Executable (PE) format that makes hash of the structural information. Central idea is to use selected properties of file headers and complexity of section’s contents in PE file for hash.

The main goal of this thesis is to make correct and good defined variant of implementation of hash function for Portable Executable based on pehash algorithm suitable for the task of malware classification.

# 2. Portable Executable File Format

The PE (Portable Executable) file format is used for executable files on 32 and 64- bit Windows platforms. Not only EXE but also, for example, DLL, SYS and SCR files have the PE file structure, which is a modified version of the COFF format [7]. The PE file header consists of a MZ-DOS header and MS-DOS stub, the PE signature, the COFF (PE) file header, and an optional header, the file headers are followed by section headers and sections. Typical PE file layout shown on Figure 1.

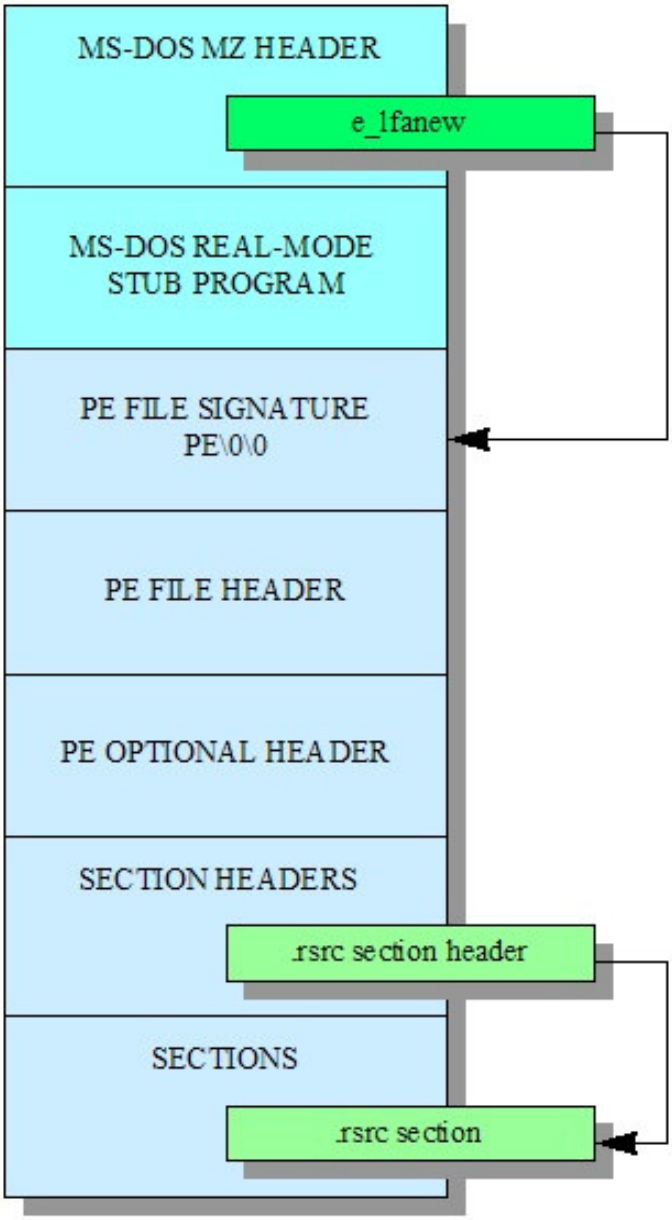


*Figure 1. PE file format structure. [7]*

## 2.1. MZ header and DOS stub

The first two bytes within the MZ-DOS header are always “MZ”, followed by some fields containing additional information and the offset to the actual PE header - *e_lfanew* (offset 0x3c). This is the PE header offset (in bytes) from the beginning of the file. This must point to the first byte of the PE signature, PE\x0\x0, aligned using the double-word boundary. In case the file is executed in DOS mode, the MS-DOS stub will be ran and will print out a sentence like “This program cannot be run in DOS mode”.

## 2.2. PE Signature

At the file offset specified at *e_lfanew*, is a 4-byte signature that identifies the file as a PE format image file. This signature is “PE\0\0”

## 2.3. PE (COFF) File Header

A standard PE (COFF) file header is immediately after the PE signature of an image file. Format of this header shown in Table 1. [7]

| Offset | Size | Field | Description |
|---|---|---|---|
| 0 | 2 | Machine | The number that identifies the type of target machine. More details in Table 2. |
| 2 | 2 | NumberOfSections | The number of sections. This indicates the size of the section table, which immediately follows the headers. |
| 4 | 4 | TimeDateStamp | The low 32 bits of the number of seconds since 00:00 January 1, 1970 that indicates when the file was created. |
| 8 | 4 | PointerToSymbolTable | This value should be zero for an image because COFF debugging information is deprecated. |
| 12 | 4 | NumberOfSymbols | This value should be zero for an image because COFF debugging information is deprecated. |
| 16 | 2 | SizeOfOptionalHeader | The size of the optional header, which is required for executable files but not for object files. This value should be zero for an object file. |

| Offset | Size | Field | Description |
|---|---|---|---|
| 18 | 2 | Characteristics | The flags that indicate the attributes of the file. For specific flag values. More details in Table 3. |

*Table 1. PE (COFF) File Header*

### 2.3.1. Machine Types

The Machine field has one of the following values that specifies its CPU type. Format of this field shown in Table 2. [7]

| Constant | Value | Description |
|---|---|---|
| IMAGE_FILE_MACHINE_UNKNOWN | 0x0 | The contents of this field are assumed to be applicable to any machine type |
| IMAGE_FILE_MACHINE_AM33 | 0x1d3 | Matsushita AM33 |
| IMAGE_FILE_MACHINE_AMD64 | 0x8664 | x64 |
| IMAGE_FILE_MACHINE_ARM | 0x1c0 | ARM little endian |
| IMAGE_FILE_MACHINE_ARMNT | 0x1c4 | ARMv7 (or higher) Thumb mode only |
| IMAGE_FILE_MACHINE_ARM64 | 0xaa64 | ARMv8 in 64-bit mode |
| IMAGE_FILE_MACHINE_EBC | 0xebc | EFI byte code |
| IMAGE_FILE_MACHINE_I386 | 0x14c | Intel 386 or later processors and compatible processors |
| IMAGE_FILE_MACHINE_IA64 | 0x200 | Intel Itanium processor family |
| IMAGE_FILE_MACHINE_M32R | 0x9041 | Mitsubishi M32R little endian |
| IMAGE_FILE_MACHINE_MIPS16 | 0x266 | MIPS16 |
| IMAGE_FILE_MACHINE_MIPSFPU | 0x366 | MIPS with FPU |
| IMAGE_FILE_MACHINE_MIPSFPU16 | 0x466 | MIPS16 with FPU |
| IMAGE_FILE_MACHINE_POWERPC | 0x1f0 | Power PC little endian |
| IMAGE_FILE_MACHINE_POWERPCFP | 0x1f1 | Power PC with floating point support |
| IMAGE_FILE_MACHINE_R4000 | 0x166 | MIPS little endian |
| IMAGE_FILE_MACHINE_SH3 | 0x1a2 | Hitachi SH3 |
| IMAGE_FILE_MACHINE_SH3DSP | 0x1a3 | Hitachi SH3 DSP |
| IMAGE_FILE_MACHINE_SH4 | 0x1a6 | Hitachi SH4 |
| IMAGE_FILE_MACHINE_SH5 | 0x1a8 | Hitachi SH5 |
| IMAGE_FILE_MACHINE_THUMB | 0x1c2 | ARM or Thumb (“interworking”) |

| Constant | Value | Description |
|---|---|---|
| IMAGE_FILE_MACHINE_WCEMIPSV2 | 0x169 | MIPS little-endian WCE v2 |

*Table 2. Machine Types.*

### 2.3.2. Image Characteristics

The Characteristics field contains flags that indicate attributes of the object or image file. The following flags are currently defined. [7]

| Flag | Value | Description |
|---|---|---|
| IMAGE_FILE_RELOCS_STRIPPED | 0x0001 | Image only, Windows CE, and Windows NT® and later. This indicates that the file does not contain base relocations and must therefore be loaded at its preferred base address. If the base address is not available, the loader reports an error. The default behavior of the linker is to strip base relocations from executable (EXE) files. |
| IMAGE_FILE_EXECUTABLE_IMAGE | 0x0002 | Image only. This indicates that the image file is valid and can be run. If this flag is not set, it indicates a linker error. |
| IMAGE_FILE_LINE_NUMS_STRIPPED | 0x0004 | This flag is deprecated and should be zero. |
| IMAGE_FILE_LOCAL_SYMS_STRIPPED | 0x0008 | This flag is deprecated and should be zero. |
| IMAGE_FILE_AGGRESSIVE_WS_TRIM | 0x0010 | This flag is deprecated for Windows 2000 and later and must be zero. |
| IMAGE_FILE_LARGE_ADDRESS_ AWARE | 0x0020 | Application can handle > 2-GB addresses. |
| | 0x0040 | This flag is reserved for future use. |
| IMAGE_FILE_BYTES_REVERSED_LO | 0x0080 | This flag is deprecated and should be zero. |
| IMAGE_FILE_32BIT_MACHINE | 0x0100 | Machine is based on a 32-bit-word architecture. |
| IMAGE_FILE_DEBUG_STRIPPED | 0x0200 | Debugging information is removed from the image file. |

| Flag | Value | Description |
|---|---|---|
| IMAGE_FILE_REMOVABLE_RUN_ FROM_SWAP | 0x0400 | If the image is on removable media, fully load it and copy it to the swap file. |
| IMAGE_FILE_NET_RUN_FROM_SWAP | 0x0800 | If the image is on network media, fully load it and copy it to the swap file. |
| IMAGE_FILE_SYSTEM | 0x1000 | The image file is a system file, not a user program. |
| IMAGE_FILE_DLL | 0x2000 | The image file is a dynamic-link library (DLL). Such files are considered executable files for almost all purposes, although they cannot be directly run. |
| IMAGE_FILE_UP_SYSTEM_ONLY | 0x4000 | The file should be run only on a uniprocessor machine. |
| IMAGE_FILE_BYTES_REVERSED_HI | 0x8000 | This flag is deprecated and should be zero. |

*Table 3. Image Characteristics*

## 2.4. Optional Header

Optional header specifies the structure of the page image in more detail. The term "optional" is not an appropriate choice for this header because it does not reflect reality. In fact, the file cannot be loaded without this header. This header is mandatory, not optional! This header has 3 parts. [7]

| Offset (PE32/PE32+) | Size (PE32/PE32+) | Header part | Description |
|---|---|---|---|
| 0 | 28/24 | Standard fields | Fields that are defined for all implementations of COFF, including UNIX. Table 5. |
| 28/24 | 68/88 | Windows-specific fields | Additional fields to support specific features of Windows. Table 6 |
| 96/112 | Variable | Data directories | Address/size pairs for special tables that are found in the image file and are used by the operating system (for example, the import table and the export table). Table 9 |

*Table 4. Optional header structure.*

### 2.4.1. Standard fields

These fields contain general information that is useful for loading and running an executable file. They are unchanged for the PE32+ format. [7]

| Offset | Size | Field | Description |
|---|---|---|---|
| 0 | 2 | Magic | The unsigned integer that identifies the state of the image file.<br>0x10B identifies it as a PE executable.<br>0x107 identifies it as a ROM image,<br>0x20B identifies it as a PE32+ executable. |
| 2 | 1 | MajorLinkerVersion | The linker major version number. |
| 3 | 1 | MinorLinkerVersion | The linker minor version number. |
| 4 | 4 | SizeOfCode | The size of the code (text) section, or the sum of all code sections if there are multiple sections. |
| 8 | 4 | SizeOfInitializedData | The size of the initialized data section, or the sum of all such sections if there are multiple data sections. |
| 12 | 4 | SizeOfUninitializedData | The size of the uninitialized data section (BSS), or the sum of all such sections if there are multiple BSS sections. |
| 16 | 4 | AddressOfEntryPoint | The address of the entry point relative to the image base when the executable file is loaded into memory. For program images, this is the starting address. For device drivers, this is the address of the initialization function. An entry point is optional for DLLs. When no entry point is present, this field must be zero. |
| 20 | 4 | BaseOfCode | The address that is relative to the image base of the beginning-of-code section when it is loaded into memory. |
| 24 | 4 | BaseOfData<br>PE file only, absent in PE32+ | The address that is relative to the image base of the beginning-of-data section when it is loaded into memory. |

*Table 5. Optional header standard fields.*

### 2.4.2. Windows specific fields

Fields are an extension to the COFF optional header format. They contain additional information that is required by the linker and loader in Windows. [7]

| Offset (PE32/ PE32+) | Size (PE32/ PE32+) | Field | Description |
|---|---|---|---|
| 28/24 | 4/8 | ImageBase | The preferred address of the first byte of image when loaded into memory; must be a multiple of 64 K. |
| 32/32 | 4 | SectionAlignment | The alignment (in bytes) of sections when they are loaded into memory. It must be greater than or equal to FileAlignment. The default is the page size for the architecture. |
| 36/36 | 4 | FileAlignment | The alignment factor (in bytes) that is used to align the raw data of sections in the image file. The value should be a power of 2 between 512 and 64 K, inclusive. The default is 512. If the SectionAlignment is less than the architecture's page size, then FileAlignment must match SectionAlignment. |
| 40/40 | 2 | MajorOperatingSystemVersion | The major version number of the required operating system. |
| 42/42 | 2 | MinorOperatingSystemVersion | The minor version number of the required operating system. |
| 44/44 | 2 | MajorImageVersion | The major version number of the image. |
| 46/46 | 2 | MinorImageVersion | The minor version number of the image. |
| 48/48 | 2 | MajorSubsystemVersion | The major version number of the subsystem. |
| 50/50 | 2 | MinorSubsystemVersion | The minor version number of the subsystem. |
| 52/52 | 4 | Win32VersionValue | Reserved, must be zero. |
| 56/56 | 4 | SizeOfImage | The size (in bytes) of the image, including all headers, as the image is loaded in memory. It must be a multiple of SectionAlignment. |
| 60/60 | 4 | SizeOfHeaders | The combined size of an MS-DOS stub, PE header, and section headers rounded up to a multiple of FileAlignment. |

| Offset (PE32/ PE32+) | Size (PE32/ PE32+) | Field | Description |
|---|---|---|---|
| 64/64 | 4 | CheckSum | The image file checksum. The algorithm for computing the checksum is incorporated into IMAGHELP.DLL. The following are checked for validation at load time: all drivers, any DLL loaded at boot time, and any DLL that is loaded into a critical Windows process. |
| 68/68 | 2 | Subsystem | The subsystem that is required to run this image. Table 7 |
| 70/70 | 2 | DllCharacteristics | DLL Characteristics. Table 8 |
| 72/72 | 4/8 | SizeOfStackReserve | The size of the stack to reserve. Only SizeOfStackCommit` is committed; the rest is made available one page at a time until the reserve size is reached. |
| 76/80 | 4/8 | SizeOfStackCommit | The size of the stack to commit. |
| 80/88 | 4/8 | SizeOfHeapReserve | The size of the local heap space to reserve. Only SizeOfHeapCommit is committed; the rest is made available one page at a time until the reserve size is reached. |
| 84/96 | 4/8 | SizeOfHeapCommit | The size of the local heap space to commit. |
| 88/104 | 4 | LoaderFlags | Reserved, must be zero. |
| 92/108 | 4 | NumberOfRvaAndSizes | The number of data-directory entries in the remainder of the optional header. Each describes a location and size. |

*Table 6. Optional Header Windows-Specific Fields.*

**Subsystem**

Values for the Subsystem field of the optional header determine which Windows subsystem is required to run the image. [7]

| Constant | Value | Description |
|---|---|---|
| IMAGE_SUBSYSTEM_UNKNOWN | 0 | An unknown subsystem |
| IMAGE_SUBSYSTEM_NATIVE | 1 | Device drivers and native Windows processes |
| IMAGE_SUBSYSTEM_WINDOWS_GUI | 2 | The Windows graphical user interface (GUI) subsystem |

| Constant | Value | Description |
|---|---|---|
| IMAGE_SUBSYSTEM_WINDOWS_CUI | 3 | The Windows character subsystem |
| IMAGE_SUBSYSTEM_POSIX_CUI | 7 | The Posix character subsystem |
| IMAGE_SUBSYSTEM_WINDOWS_CE_GUI | 9 | Windows CE |
| IMAGE_SUBSYSTEM_EFI_APPLICATION | 10 | An Extensible Firmware Interface (EFI) application |
| IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER | 11 | An EFI driver with boot services |
| IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER | 12 | An EFI driver with run-time services |
| IMAGE_SUBSYSTEM_EFI_ROM | 13 | An EFI ROM image |
| IMAGE_SUBSYSTEM_XBOX | 14 | XBOX |

*Table 7. Optional header Windows Subsystem.*

### DLL Characteristics

Values defined for the DllCharacteristics field of the optional header. [7]

| Constant | Value | Description |
|---|---|---|
| | 0x0001 | Reserved, must be zero. |
| | 0x0002 | Reserved, must be zero. |
| | 0x0004 | Reserved, must be zero. |
| | 0x0008 | Reserved, must be zero. |
| IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE | 0x0040 | DLL can be relocated at load time. |
| IMAGE_DLL_CHARACTERISTICS_FORCE_INTEGRITY | 0x0080 | Code Integrity checks are enforced. |
| IMAGE_DLL_CHARACTERISTICS_NX_COMPAT | 0x0100 | Image is NX compatible. |
| IMAGE_DLLCHARACTERISTICS_NO_ISOLATION | 0x0200 | Isolation aware, but do not isolate the image. |
| IMAGE_DLLCHARACTERISTICS_NO_SEH | 0x0400 | Does not use structured exception (SE) handling. No SE handler may be called in this image. |
| IMAGE_DLLCHARACTERISTICS_NO_BIND | 0x0800 | Do not bind the image. |
| | 0x1000 | Reserved, must be zero. |

| | | |
|---|---|---|
| IMAGE_DLLCHARACTERISTICS_ WDM_DRIVER | 0x2000 | A WDM driver. |
| IMAGE_DLLCHARACTERISTICS_ TERMINAL_SERVER_AWARE | 0x8000 | Terminal Server aware. |

*Table 8. Optional header DLL characteristics.*

### 2.4.3. Data directories

This is one of the most important structures in the optional header. It is the array of directory structures with pointer and size. Each data directory gives the address and size of a table or string that Windows uses. Structure defined as: [7]

```
typedef struct _IMAGE_DATA_DIRECTORY {
DWORD VirtualAddress;
DWORD Size;
} IMAGE_DATA_DIRECTORY,*PIMAGE_DATA_DIRECTORY;
```

*Figure 2. IMAGE_DATA_DIRECTORY structure. [7]*

The data directories fields are: [7]

| **Offset (PE/PE32+)** | **Size** | **Field** | **Description** |
|---|---|---|---|
| 96/112 | 8 | Export Table | The export table address and size (Image Only). |
| 104/120 | 8 | Import Table | The import table address and size. |
| 112/128 | 8 | Resource Table | The resource table address and size. |
| 120/136 | 8 | Exception Table | The exception table address and size. |
| 128/144 | 8 | Certificate Table | The attribute certificate table address and size. (Image Only). |
| 136/152 | 8 | Base Relocation Table | The base relocation table address and size (Image Only). |
| 144/160 | 8 | Debug | The debug data starting address and size. |
| 152/168 | 8 | Architecture | Reserved, must be 0 |
| 160/176 | 8 | Global Ptr | The RVA of the value to be stored in the global pointer register. The size member of this structure must be set to zero. |
| 168/184 | 8 | TLS Table | The thread local storage (TLS) table address and size. |
| 176/192 | 8 | Load Config Table | The load configuration table address and size. |
| 184/200 | 8 | Bound Import | The bound import table address and size. |
| 192/208 | 8 | IAT | The import address table address and size. |

| | | | |
|---|---|---|---|
| 200/216 | 8 | Delay Import Descriptor | The delay import descriptor address and size. |
| 208/224 | 8 | CLR Runtime Header | The CLR runtime header address and size. |
| 216/232 | 8 | Reserved, must be zero | |

*Table 9. Optional header Data directories.*

## 2.5. Section Table (Section Headers)

The section table, which holds an entry for every section of the file, its location is always determined as "the first byte behind the header". The number of entries in the section table is given by the NumberOfSections field in the file header. Entries in the section table are numbered starting from one (1). Each section header (section table entry) has the following format, for a total of 40 bytes per entry. [7]

| **Offset** | **Size** | **Field** | **Description** |
|---|---|---|---|
| 0 | 8 | Name | An 8-byte, null-padded UTF-8 encoded string. If the string is exactly 8 characters long, there is no terminating null. Long names in object files are truncated if they are emitted to an executable file. |
| 8 | 4 | VirtualSize | The total size of the section when loaded into memory. If this value is greater than SizeOfRawData, the section is zero-padded. |
| 12 | 4 | VirtualAddress | For executable images, the address of the first byte of the section relative to the image base when the section is loaded into memory. |
| 16 | 4 | SizeOfRawData | The size of the section (for object files) or the size of the initialized data on disk (for image files). For executable images, this must be a multiple of FileAlignment from the optional header. If this is less than VirtualSize, the remainder of the section is zero-filled. When a section contains only uninitialized data, this field should be zero. |
| 20 | 4 | PointerToRawData | The file pointer to the first page of the section within the COFF file. For executable images, this must be a multiple of FileAlignment from the optional header. When a section contains only uninitialized data, this field should be zero. |
| 24 | 4 | PointerToRelocations | The file pointer to the beginning of relocation entries for the section. This is set to zero for executable images or if there are no relocations. |

| Offset | Size | Field | Description |
|---|---|---|---|
| 28 | 4 | PointerToLinenumbers | The file pointer to the beginning of line-number entries for the section. This value should be zero for an image because COFF debugging information is deprecated. |
| 32 | 2 | NumberOfRelocations | The number of relocation entries for the section. This is set to zero for executable images. |
| 34 | 2 | NumberOfLinenumbers | The number of line-number entries for the section. This value should be zero for an image because COFF debugging information is deprecated. |
| 36 | 4 | Characteristics | The flags that describe the characteristics of the section (Section Flags). |

*Table 10. Section table entry*

### 2.5.1. Section Characteristics

The section flags in the Characteristics field defined as: [7]

| Flag | Value | Description |
|---|---|---|
| | 0x00000000 | Reserved for future use. |
| | 0x00000001 | Reserved for future use. |
| | 0x00000002 | Reserved for future use. |
| | 0x00000004 | Reserved for future use. |
| IMAGE_SCN_TYPE_NO_PAD | 0x00000008 | The section should not be padded to the next boundary. This flag is obsolete and is replaced by IMAGE_SCN_ALIGN_1BYTES . This is valid only for object files. |
| | 0x00000010 | Reserved for future use. |
| IMAGE_SCN_CNT_CODE | 0x00000020 | The section contains executable code. |
| IMAGE_SCN_CNT_INITIALIZED_DATA | 0x00000040 | The section contains initialized data. |
| IMAGE_SCN_CNT_UNINITIALIZED_DATA | 0x00000080 | The section contains uninitialized data. |
| IMAGE_SCN_LNK_OTHER | 0x00000100 | Reserved for future use. |
| IMAGE_SCN_LNK_INFO | 0x00000200 | The section contains comments or other information. The **.drectve** section has this type. This is valid for object files only. |

| Flag | Value | Description |
|---|---|---|
| | 0x00000400 | Reserved for future use. |
| IMAGE_SCN_LNK_REMOVE | 0x00000800 | The section will not become part of the image. This is valid only for object files. |
| IMAGE_SCN_LNK_COMDAT | 0x00001000 | The section contains COMDAT data. This is valid only for object files. |
| IMAGE_SCN_GPREL | 0x00008000 | The section contains data referenced through the global pointer (GP). |
| IMAGE_SCN_MEM_PURGEABLE | 0x00020000 | Reserved for future use. |
| IMAGE_SCN_MEM_16BIT | 0x00020000 | For ARM machine types, the section contains Thumb code. Reserved for future use with other machine types. |
| IMAGE_SCN_MEM_LOCKED | 0x00040000 | Reserved for future use. |
| IMAGE_SCN_MEM_PRELOAD | 0x00080000 | Reserved for future use. |
| IMAGE_SCN_ALIGN_1BYTES | 0x00100000 | Align data on a 1-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_2BYTES | 0x00200000 | Align data on a 2-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_4BYTES | 0x00300000 | Align data on a 4-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_8BYTES | 0x00400000 | Align data on an 8-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_16BYTES | 0x00500000 | Align data on a 16-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_32BYTES | 0x00600000 | Align data on a 32-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_64BYTES | 0x00700000 | Align data on a 64-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_128BYTES | 0x00800000 | Align data on a 128-byte boundary. Valid only for object files. |

| Flag | Value | Description |
|---|---|---|
| IMAGE_SCN_ALIGN_256BYTES | 0x00900000 | Align data on a 256-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_512BYTES | 0x00A00000 | Align data on a 512-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_1024BYTES | 0x00B00000 | Align data on a 1024-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_2048BYTES | 0x00C00000 | Align data on a 2048-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_4096BYTES | 0x00D00000 | Align data on a 4096-byte boundary. Valid only for object files. |
| IMAGE_SCN_ALIGN_8192BYTES | 0x00E00000 | Align data on an 8192-byte boundary. Valid only for object files. |
| IMAGE_SCN_LNK_NRELOC_OVFL | 0x01000000 | The section contains extended relocations. |
| IMAGE_SCN_MEM_DISCARDABLE | 0x02000000 | The section can be discarded as needed. |
| IMAGE_SCN_MEM_NOT_CACHED | 0x04000000 | The section cannot be cached. |
| IMAGE_SCN_MEM_NOT_PAGED | 0x08000000 | The section is not pageable. |
| IMAGE_SCN_MEM_SHARED | 0x10000000 | The section can be shared in memory. |
| IMAGE_SCN_MEM_EXECUTE | 0x20000000 | The section can be executed as code. |
| IMAGE_SCN_MEM_READ | 0x40000000 | The section can be read. |
| IMAGE_SCN_MEM_WRITE | 0x80000000 | The section can be written to. |

*Table 11. Section Characteristics field.*

## 2.6. Section Data

Typical sections contain code or data that linkers and loaders process without special knowledge of the section contents. A section consists of simple blocks of bytes. However, for sections that contain all zeros, the section data does not need to be included. The data for each section is located at the file offset that was given by the PointerToRawData field in the section header. The size of this data in the file is

indicated by the SizeOfRawData field. If SizeOfRawData is less than VirtualSize, the remainder is padded with zeros. In an image file, the section data must be aligned on a boundary as specified by the FileAlignment field in the optional header.

# 3. peHash function design

Generic hash function for Portable Executable files that generates a per-binary specific hash value based on structural data found in the file headers and structural information about the executable's section data was described in the paper "peHash: A Novel Approach to Fast Malware Clustering" [6]. The following Portable Executable properties are taken into account as well:

- **Image Characteristics**: General flags for the Portable Executable, e.g. whether the given file is a DLL or can only be run on a single processor machine. (Table 3)
- **Subsystem**: Indicates the Windows Subsystem this binary is to be run in, such as GUI, CLI or device driver.(Table 7)
- **Stack Commit Size**: The initial size of program stack to be allocated in bytes. This value is rounded up to a value divisible by 4096, Windows' page boundary, before inclusion in the hash as the Windows Portable Executable Loader does the same.(Table 5)
- **Heap Commit Size**: Initial size of program heap to be allocated, also rounded up to page boundary size. (Table 5)

For each section in the Portable Executable, the following structural information is included:

- **Virtual Address:** The address, the section's content is going to be loaded. (Table 10)
- **Raw Size**: Size of the section in the Portable Executable file itself; can be smaller than the actual size occupied in memory after loading due to rounding to page boundaries. (Table 10)
- **Section Characteristics**: Section flags describing initial privileges for the allocated memory, such as reading, writing and execution of code. (Table 11)
- **Result** of bzip2 compression ratio of section data scaled to [0 ... 7].

The following pseudo-code describes the exact generation for the hash value from the global properties, where $v$[8..24] means bits 8 to 24 of value v and $\oplus$ means XOR: [6]

$$
\begin{aligned}
hash[0] &:= characteristics[0..7] \\
&\quad \oplus \ characteristics[8..15] \\
hash[1] &:= subsystem[0..7] \\
&\quad \oplus \ subsystem[8..15] \\
hash[2] &:= stackcommit[8..15] \\
&\quad \oplus \ stackcommit[16..23] \\
&\quad \oplus \ stackcommit[24..31] \\
hash[3] &:= heapcommit[8..15] \\
&\quad \oplus \ heapcommit[16..23] \\
&\quad \oplus \ heapcommit[24..31]
\end{aligned}
$$

*Figure 3. Pseudo-code for header. [6]*

Additionally, for each section, the following sub-hash is appended to the hash:[6]

$$
\begin{aligned}
shash[0] &:= virtaddress[9..31] \\
shash[2] &:= rawsize[8..31] \\
shash[4] &:= characteristics[16..23] \\
&\quad \oplus \ characteristics[24..31] \\
shash[5] &:= kolmogorov \in [0..7] \subset \mathbb{N}
\end{aligned}
$$

*Figure 4. Pseudo-code for section. [6]*

The last step is to calculate the SHA1 value of the above hash buffer and use this as the final hash value.

# 4. Implementation and issues

There are some public implementation of this hash where made, but looks like there are programming and algorithm mistakes in the code realization. In the current implementation, quality for clustering instances of the same polymorphic malware is lower in comparison with the original idea.

Probably the first public usage and code implementation in python of peHash was made by #totalhash [8], currently this blog post is not available online, but it can be accessed by Google cache [9]. Later, some other frameworks [10] include pehash implementation based on #totalhash into code base. Most known are Viper Framework [11] and CRITs Services Collection [12]. Also one standalone implementation not based on #totalhash was found - pehashd from GitHub user endgameinc. [14] Let us look what the problems are in #totalhash implementation.

## 4.1. Incorrect padding

As far as implicit variant (*bitstring.BitArray(hex(X))*) is used for converting integer value to bits string, for fields "*image characteristics*" and "*subsystem*" wrong padding method is in place. When value of ***X*** in *bitstring.BitArray(hex(X))* is smaller than 256, then XORing parts of result will fail, as far as result will be 8 bits. Padding is for byte boundary only. An example of step by step execution shown on Figure 5.

```
>>>
>>> #image characteristics
>>> img_chars = bitstring.BitArray(hex(240))
>>> #pad to 16 bits
>>> img_chars = bitstring.BitArray(bytes=img_chars.tobytes())
>>> img_chars.bin
'11110000'
>>> img_chars_xor = img_chars[0:8] ^ img_chars[8:16]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "C:\MyFiles\PE Hash\bitstring.py", line 1130, in __xor__
    raise ValueError("Bitstrings must have the same length "
ValueError: Bitstrings must have the same length for ^ operator.
>>> img_chars[0:8], img_chars[8:16]
(BitArray('0xf0'), BitArray(''))
```

*Figure 5. Incorrect padding.*

As result, getting pehash will fail for many files.

## 4.2. Issue in bitstring module

Third-party python *bitstring* [13] module have some issues when handling output of hex() function with argument greater than 2147483647 (0x7fffffff). An example is shown on Figure 6.

```
>>> 0x7fffffff
2147483647
>>> bitstring.BitArray(hex(2147483647))
BitArray('0x7fffffff')
>>> bitstring.BitArray(hex(2147483647+1))
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "C:\MyFiles\PE Hash\bitstring.py", line 3047, in __new__
    y = Bits.__new__(BitArray, auto, length, offset, **kwargs)
  File "C:\MyFiles\PE Hash\bitstring.py", line 786, in __new__
    x._datastore._appendstore(Bits._init_with_token(*token)._datastore)
  File "C:\MyFiles\PE Hash\bitstring.py", line 1219, in _init_with_token
    b = cls(**{_tokenname_to_initialiser[name]: value})
  File "C:\MyFiles\PE Hash\bitstring.py", line 796, in __new__
    x._initialise(auto, length, offset, **kwargs)
  File "C:\MyFiles\PE Hash\bitstring.py", line 817, in _initialise
    init_without_length_or_offset[k](self, v)
  File "C:\MyFiles\PE Hash\bitstring.py", line 1910, in _sethex
    raise CreationError("Invalid symbol in hex initialiser.")
CreationError: Invalid symbol in hex initialiser.
```

*Figure 6. Issue in bitstring module.*

As result, getting pehash will fail for small amount of files.

## 4.3. Wrong value for field "Subsystem"

Field *"Subsystem"* has wrong value. It has value from FILE_HEADER.Machine of PE file instead of OPTIONAL_HEADER.Subsystem as shown on Figure 7.

```
#subsystem -
sub_chars = bitstring.BitArray(hex(exe.FILE_HEADER.Machine))
```

*Figure 7. Wrong value for field "Subsystem" [10] [12]*

## 4.4. Missing round up

Missing rounding up to a value divisible by 4096 for SizeOfStackCommit and SizeOfHeapCommit fields

## 4.5. Incorrect slicing

Another issue is in incorrect slicing. peHash pseudo-code means slicing as of bits in multi-byte fields. In #totalhash implementation, we can see that the same slicing order used for BitArray structure. That structure is in Big Endian format, as far as implicit method used for conversion from integer to BitArray and Python slicing working from index 0 for inerrable object. Result of slicing looks like:

- pseudo-code: X[0:7], [closed Interval], 8 Least Significant bits
- python: X[0:8], [half-closed intervals), 8 items from index 0

As shown on Figure 8 Python slicing start from left to right and does not maps to pseudo code meaning.

```
>>> x=bitstring.BitArray(hex(0x03020100))
>>> x
BitArray('0x3020100')
>>> x[0:8], x[8:16], x[8:16], x[16:24], x[24:32]
(BitArray('0x30'), BitArray('0x20'), BitArray('0x20'), BitArray('0x10'), BitArray('0x0'))
```

*Figure 8. Slicing of multi-byte BitArray.*

Totalhash implementation does not take care about that for SizeOfStackCommit, SizeOfHeapCommit, VirtualAddress and SizeOfRawData fields. As result, Least Significant Bits are NOT ignored. PE files with small different in RawSize value will have different hash. Example - virus [15] has made changes by adding into value of “RawSize" field incremental numbers in section header (second section), note that hashes are different, but they should be the same:

| **peHash** | **File name** |
|---|---|
| b2e0cb11d4f398d98dfd3e05a398de39e4a69204 | .\Backdoor.Win32.Agent.adwz |
| dfc119cf023c3ba276a87af20e81b21aa09b4861 | .\Backdoor.Win32.Agent.+16.adwz |
| 4fee405c231f2dc0f9e33033099d26eac3caa34b | .\Backdoor.Win32.Agent.+32.adwz |

| peHash | File name |
|---|---|
| 61a6f71c1154dfda7fbad57667a41207d4396f44 | .\Backdoor.Win32.Agent.+48.adwz |
| 4cf16f817d1d106b6eca1b8d23ab200c09f279eb | .\Backdoor.Win32.Agent.+64.adwz |

*Table 12. Hashes for modified file.*

## 4.6. Compression ratio scaling

Another issue with implementation is related to missing scale up of the result of the bzip2 section data compression ratio up to 7 (Figure 9).

```
#entropy calulation
address = section.VirtualAddress
size = section.SizeOfRawData
raw = exe.write()[address+size:]
if size == 0:
    kolmog = bitstring.BitArray(float=1, length=32)
    pehash_bin.append(kolmog[0:8])
    continue
bz2_raw = bz2.compress(raw)
bz2_size = len(bz2_raw)
#k = round(bz2_size / size, 5)
k = bz2_size / size
kolmog = bitstring.BitArray(float=k, length=32)
pehash_bin.append(kolmog[0:8])
```

*Figure 9. Compression ratio. [10], [12]*

Even more, incorrect float format for converting into BitArray is in place. Therefore slicing for bit presentation of float value is incorrectly used too. As result, value of this field have only 4 variants, instead of 8 as shown in Figure 10.

```
>>> for x in range(10):
...     print x/10.0, bitstring.BitArray(float=x/10.0, length=32)[0:8]
...
0.0 0x00
0.1 0x3d
0.2 0x3e
0.3 0x3e
0.4 0x3e
0.5 0x3f
0.6 0x3f
0.7 0x3f
0.8 0x3f
0.9 0x3f
```

*Figure 10. Slicing of float value.*

# 5. Proposed implementation

## 5.1. Corrected improper implementation

Implementation based on original idea with fixed known issues was made [appendix 1] [17]. Hash buffer defined as file header data and one or more section data as shown in Table 13 and Table 14. Sections are sorted by VirtualAddress, that friendly done by "pefile" module. All multi-byte values stored as unsigned integers in Big Endian format. Result of peHash function is SHA1 hash in hex-digest format of that buffer.

| **Field length in bytes** | **Value** |
|---|---|
| 1 | Image Characteristics, bytes are XOR-ed. |
| 1 | Subsystem, bytes are XOR-ed. |
| 1 | Stack Commit Size, rounded up to a value divisible by 4096, one least significant byte is discarded, all other bytes are XOR-ed. |
| 1 | Heap Commit Size, rounded up to a value divisible by 4096, one least significant byte is discarded, all other bytes are XOR-ed. |

*Table 13. peHash buffer for file header.*

| **Field length in bytes** | **Value** |
|---|---|
| 3 | VirtualAddress, right shift by 9 bits. |
| 3 | SizeOfRawData, right shift by 8 bits (one least significant byte is discarded). |
| 1 | Characteristics, right shift by 16 bits, (two least significant byte are discarded), all other bytes are XOR-ed. |
| 1 | Complexity, compression ratio of section data, scaled up to 7: complexity = int(round(lenCompressedData * 7.0 / lenData))<br>- 0 if SizeOfRawData is 0<br>- 7 if complexity > 7<br>- Complexity |

*Table 14. peHash buffer for section properties.*

## 5.2. Extended implementation

In additional to the original idea, some changes and enhancements are proposed. Main goal is to make fine-tuning for properties used in hash and add some new ones that can be changed only by relinking object files.

- **Image Characteristics field** – masking of deprecated and reserved bits was added. (Table 3)
- **Section Characteristics** – masking of reserved and not used for PE image file bits was added. (Table 11)
- **Remove XOR-ing** bytes for values of Image Characteristics, Sybsystem, Stack Commit Size, Heap Commit Size and Section Characteristics to prevent collisions.
- **SizeOfRawData** – do not skip eight least significant bits (1 byte) but round value up to 512 bits boundary. SizeOfRawData can be directly edited and data added up to 512 bits boundary value (to be correct – up to FileAlignment value) without relink object files.
- **VirtualAddress** - do not skip nine least significant bits but round value up to 512 bits boundary.
- **Compression ratio for section data –** fixed value 8 added for situation when size of compressed data is bigger than size of data itself.
- **SectionAlignment** and **FileAlignment** fields added into hash. The values rounded down to power of two. Round down selected to prevent overflow of four bytes integer. These values cannot be changed without relink object files. Default values are 4096 and 512 bytes.
- **Data Directories** (Table 9) status added into hash. Value is a bit flags, bit set to 1 when VirtualAddress for data directory is not zero, else set to 0. Data directories entries with index 7, 8, 15 ignored, as far as index 15 is reserved and system loader ignores indexes 7 and 8.

Hash buffer is defined as file header data and one or more section data as shown in Table 15 and Table 16. Sections are sorted by VirtualAddress, that friendly done by "pefile" module. All multi-byte values stored as unsigned integers in Big Endian format. Result of peHash function is SHA256 hash in hex-digest format of that buffer. [Appendix 2]

| Field length in bytes | Value |
|---|---|
| 2 | Image Characteristics, masked for unwanted bits.<br>mask: 0b0111111100100011 |
| 2 | Subsystem. |
| 4 | SectionAlignment, rounded down to power of two |
| 4 | FileAlignment, rounded down to power of two |
| 8 | SizeOfStackCommit, rounded up to a value divisible by 4096. |
| 8 | SizeOfHeapCommit, rounded up to a value divisible by 4096. |
| 2 | Data Directory Status, masked bit flags for data directories with index from 0 to value of NumberOfRvaAndSizes -1, but not bigger than 15 :<br>- 1 if VirtualAddress for directory is not 0<br>- 0 if VirtualAddress for directory is 0<br>- Mask: 0b0111111001111111 |

*Table 15. peHashNG buffer for file header.*

| Field length in bytes | Value |
|---|---|
| 4 | VirtualAddress, rounded up to a value divisible by 512. |
| 4 | SizeOfRawData, rounded up to a value divisible by 512. |
| 1 | Characteristics, right shift by 24 bits, (3 least significant byte are discarded). |
| 1 | Complexity, compression ratio of section data, scaled up to 7:<br>complexity = lenCompressedData * 7.0 / lenData<br>- 0 if SizeOfRawData is 0<br>- 8 if complexity > 7<br>- int(round(complexity)) |

*Table 16. peHashNG buffer for section properties.*

# 6. Statistics and conclusions

Currently we have two implementation of pehash – one correct (I hope) “peHash” [17] implementation based on original idea and one extended “peHashNG” [18]. During development of extended version, Windows 10,500 system files with 38,000 sections and collection of 235,000 malware PE examples [18] with about 1 million sections total were analyzed.

For Image Characteristics in Table 17 we can see counters of unique values in windows system and malware files. XOR-ing of bytes in this field has many collisions, so the masking of unwanted bits for this field is better for clustering.

| **Type of data** | **Malware** | **Windows** |
|---|---|---|
| Raw (as is in the file) | 102 | 22 |
| XOR-ed (as in peHash) | 22 | 6 |
| Masked (as in peHashNG) | 37 | 18 |

*Table 17. Image Characteristics unique counters*

The same story is with Subsystem, collisions again, so we will store raw values into hash (Table 18).

| **Type of data** | **Malware** | **Windows** |
|---|---|---|
| Raw (as is in the file) | 8 | 4 |
| XOR-ed (as in peHash) | 2 | 1 |

*Table 18. Subsystem unique counters.*

SizeOfStackCommit and SizeOfHeapCommit fields - with round up only we have more accurate clustering (Table 19, Table 20)

| **Type of data** | **Malware** | **Windows** |
|---|---|---|
| Raw (as is in the file) | 93 | 21 |
| Round up and XOR-ed (as in peHash) | 58 | 20 |
| Round up only (as in peHashNG) | 72 | 21 |

*Table 19. SizeOfStackCommit unique counters.*

| Type of data | Malware | Windows |
|---|---|---|
| Raw (as is in the file) | 53 | 3 |
| Round up and XOR-ed (as in peHash) | 35 | 3 |
| Round up only (as in peHashNG) | 40 | 3 |

*Table 20. SizeOfHeapCommit unique counters.*

SectionAlignment and FileAlignment fields - for Windows 7 PE system files in "sysnative" and "sysWOW64" directories we have only eight combinations of Section and File alignments. For malware collection, the number of combinations increased to 35. Therefore, we have additional stable parameters for clustering in peHashNG.

Data Directory field is another interesting and useful one for clustering. It is very important for PE file area. In, probably, all cases of any incorrect data in it, system loader will fail to load PE file. Note, that we do not store values from Data Directory into hash, we only detect – is that directory existing (used by PE) or not by checking VirtualAddress. If VA is not zero we set bit flag to 1 for this directory, otherwise – set it to 0. Windows and malware files statistic for Data Directory status is shown on Figure 11. As we can see, this data can be useful for clustering. Unused by system loader and reserved directory numbers are ignored by masking bits.

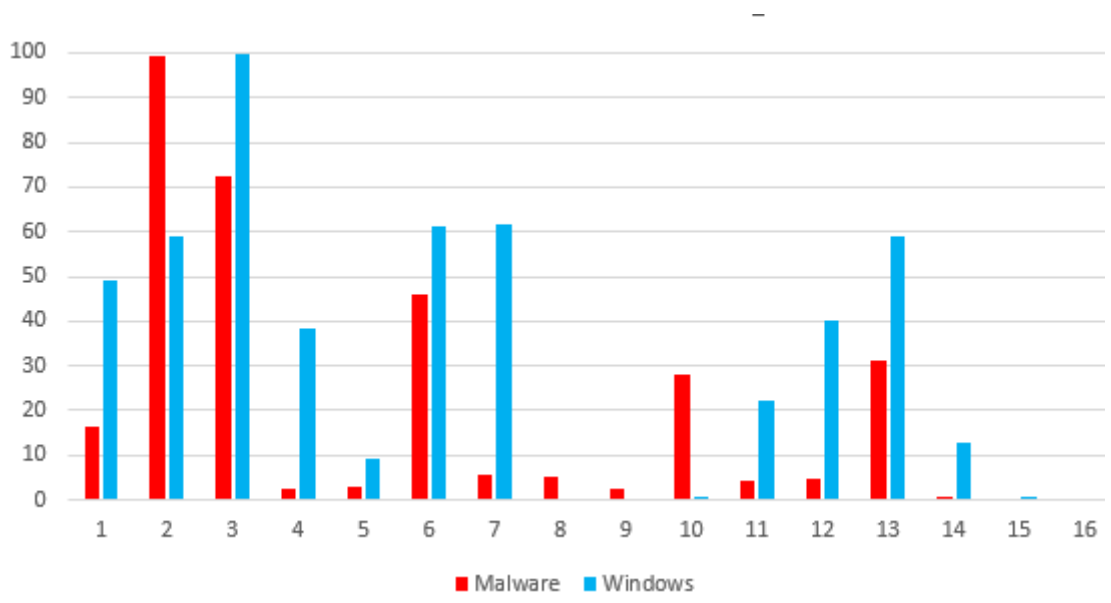


*Figure 11. Data Directories usage.*

Comparison of counts of hashes with different numbers of files in cluster is shown in Table 21. As we can see even if peHashNg has a more strictly algorithm for hash, total count of hashes and count of clusters with one file are smaller than ones for totalhash. At the same time counts of hashes for clusters with 2...100 files are bigger, so we can make a conclusion that grouping files by peHashNg is more accurate in comparison with totalhash.

| **Hash** | **1 file in cluster** | **2...10 files in cluster** | **11...100 files in cluster** | **> 100 files in cluster** | **Total hashes** |
|---|---|---|---|---|---|
| #totalhash | 113815 | 21024 | 1832 | 62 | 136733 |
| peHashNg | 106061 | 23059 | 2011 | 50 | 131181 |

*Table 21. Counts of hashes with different numbers of files in cluster.*

Implementation is made using Python programming language. Source code of peHash and peHashNG hashes is published on GitHub. [17], [18]. That hash can be used independently or in cooperation with "imphash" [2] and classical bits stream (black/white lists) hashes for quick classification of PE files.

# References


[1] Symantec, "Internet Security Threat Report (ISTR), Volume 21," 13 April 2016. [Online]. Available: https://www.symantec.com/about/newsroom/media-resources/press-kits/istr-21.

[2] FireEye, "Tracking Malware with Import Hashing," 23 Jan 2014. [Online]. Available: https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html.

[3] E. Carrera, "pefile is a Python module to read and work with PE (Portable Executable) files," 15 Apr 2016. [Online]. Available: https://github.com/erocarrera/pefile. [Accessed 15 Apr 2016].

[4] VirusTotal, "VirusTotal += imphash," 3 Feb 2014. [Online]. Available: http://blog.virustotal.com/2014/02/virustotal-imphash.html. [Accessed 12 03 2016].

[5] GitHub, "GitHub search imphash," 2016. [Online]. Available: https://github.com/search?l=python&q=imphash&type=Code.

[6] G. Wicherski, "peHash: A Novel Approach to Fast Malware Clustering," 7 December 2018. [Online]. Available: https://www.usenix.org/legacy/event/leet09/tech/full_papers/wicherski/wicherski_html/. [Accessed 12 03 2016].

[7] Microsoft, "Microsoft Portable Executable and Common Object File Format Specification," 6 Feb 2013. [Online]. Available: https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx. [Accessed 12 Feb 2016].

[8] OSDev.org, "PE," 11 March 2016. [Online]. Available: http://wiki.osdev.org/PE. [Accessed 25 March 2016].

[9] #totalhash, " PEhash Source Code," 9 Oct 2013. [Online]. Available: https://totalhash.cymru.com/blog/pehash-source-code. [Accessed 24 Feb 2016].

[10] #totalhash, "google cached version of https://totalhash.cymru.com/blog/pehash-source-code/," 29 Feb 2016. [Online]. Available:

https://webcache.googleusercontent.com/search?q=cache:https://totalhash.cymru.com/blog/page/5/. [Accessed 24 April 2016].

[11] Github, "Search result for pehash," 29 Feb 2016. [Online]. Available: https://github.com/search?l=python&q=pehash&ref=searchresults&type=Code&utf8=%E2%9C%93. [Accessed 05 April 2016].

[12] Viper Framework, "Viper is a binary analysis and management framework," [Online]. Available: https://github.com/viper-framework/viper/blob/master/viper/modules/pehash/pehasher.py. [Accessed 15 March 2016].

[13] CRITs Services Collection, "Services for CRITs that allow you to extend its functionality," [Online]. Available: https://github.com/crits/crits_services/blob/master/peinfo_service/__init__.py. [Accessed 21 March 2016].

[14] engameinc, "pehashd," 29 Jun 2013. [Online]. Available: https://github.com/endgameinc/pehashd. [Accessed 18 Jan 2016].

[15] S. Griffiths, "A Python module to help you manage your bits," 21 Mar 2016. [Online]. Available: https://github.com/scott-griffiths/bitstring. [Accessed 28 Mar 2016].

[16] VirusTotal, "Antivirus scan," 20 Jan 2016. [Online]. Available: https://virustotal.com/en/file/510b0eb1d98c544dcf4d7ddf594a89c2fbbf76e335fef093324f06c999afbd71/analysis/. [Accessed 4 March 2016].

[17] AnyMaster, "pehash for PE file, sha1 of PE structural properties.," 03 May 2015. [Online]. Available: https://github.com/AnyMaster/pehash. [Accessed 27 Mar 2016].

[18] AnyMaster, "revised "peHash: A Novel Approach to Fast Malware Clustering"," 30 Apr 2016. [Online]. Available: https://github.com/AnyMaster/pehashng. [Accessed 1 May 2016].

[19] J. Scott, "VX Heavens Snapshot (2010-05-18)," 22 Nov 2013. [Online]. Available: https://archive.org/details/vxheavens-2010-05-18. [Accessed 28 Mar 2015].

# Appendix 1 – peHash source code

Listing of "pehash.py" file:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
pehash, Portable Executable hash of structural properties

@author: AnyMaster
https://github.com/AnyMaster/pehash
"""
__version__ = '1.1.0'
__author__ = 'AnyMaster'

from hashlib import sha1
from bz2 import compress

from pefile import PE
from bitstring import pack


def get_pehash(pe_file):
    """ Return pehash for PE file, sha1 of PE structural properties.

    :param pe_file:     file name or instance of pefile.PE() class
    :rtype : string     SHA1 in hexdigest format
    """

    if isinstance(pe_file, PE):  # minimize mem. usage and time of execution
        exe = pe_file
    else:
        exe = PE(pe_file, fast_load=True)

    # Image Characteristics
    img_chars = pack('uint:16', exe.FILE_HEADER.Characteristics)
    pehash_bin = img_chars[0:8] ^ img_chars[8:16]

    # Subsystem
    subsystem = pack('uint:16', exe.OPTIONAL_HEADER.Subsystem)
    pehash_bin.append(subsystem[0:8] ^ subsystem[8:16])

    # Stack Commit Size, rounded up to a value divisible by 4096,
    # Windows page boundary, 8 lower bits must be discarded
    # in PE32+ is 8 bytes
    stack_commit = exe.OPTIONAL_HEADER.SizeOfStackCommit
    if stack_commit % 4096:
        stack_commit += 4096 - stack_commit % 4096
    stack_commit = pack('uint:56', stack_commit >> 8)
    pehash_bin.append(
        stack_commit[:8] ^ stack_commit[8:16] ^
        stack_commit[16:24] ^ stack_commit[24:32] ^
        stack_commit[32:40] ^ stack_commit[40:48] ^ stack_commit[48:56])

    # Heap Commit Size, rounded up to page boundary size,
    # 8 lower bits must be discarded
    # in PE32+ is 8 bytes
    heap_commit = exe.OPTIONAL_HEADER.SizeOfHeapCommit
    if heap_commit % 4096:
        heap_commit += 4096 - heap_commit % 4096
    heap_commit = pack('uint:56', heap_commit >> 8)
    pehash_bin.append(
        heap_commit[:8] ^ heap_commit[8:16] ^
```

```
            heap_commit[16:24] ^ heap_commit[24:32] ^
            heap_commit[32:40] ^ heap_commit[40:48] ^ heap_commit[48:56])

        # Section structural information
        for section in exe.sections:
            # Virtual Address, 9 lower bits must be discarded
            pehash_bin.append(pack('uint:24', section.VirtualAddress >> 9))

            # Size Of Raw Data, 8 lower bits must be discarded
            pehash_bin.append(pack('uint:24', section.SizeOfRawData >> 8))

            # Section Characteristics, 16 lower bits must be discarded
            sect_chars = pack('uint:16', section.Characteristics >> 16)
            pehash_bin.append(sect_chars[:8] ^ sect_chars[8:16])

            # Kolmogorov Complexity, len(Bzip2(data))/len(data)
            # (0..1} ∈ R   ->  [0..7] ⊂ N
            kolmogorov = 0
            if section.SizeOfRawData:
                kolmogorov = int(round(
                    len(compress(section.get_data()))
                    * 7.0 /
                    section.SizeOfRawData))
                if kolmogorov > 7:
                    kolmogorov = 7
            pehash_bin.append(pack('uint:8', kolmogorov))

        assert 0 == pehash_bin.len % 8
        if not isinstance(pe_file, PE):
            exe.close()

        return sha1(pehash_bin.tobytes()).hexdigest()

    if __name__ == '__main__':
        import sys
        if len(sys.argv) < 2:
            print "Error: no file specified"
            sys.exit(0)
        print get_pehash(sys.argv[1]), sys.argv[1]
```

# Appendix 2 – peHashNG source code

Listing of "pehashng.py" file:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
peHashNG, Portable Executable hash of structural properties

@author: AnyMaster
https://github.com/AnyMaster/pehashng
"""

import logging
from bz2 import compress
from hashlib import sha256
from struct import pack

from pefile import PE, PEFormatError

__version__ = '1.0.0'
__author__ = 'AnyMaster'


def pehashng(pe_file):
    """ Return pehashng for PE file, sha256 of PE structural properties.

    :param pe_file: file name or instance of pefile.PE() class
    :return: SHA256 in hexdigest format, None in case of pefile.PE() error
    :rtype: str
    """

    if isinstance(pe_file, PE):
        exe = pe_file
    else:
        try:
            exe = PE(pe_file, fast_load=True)
        except PEFormatError as exc:
            logging.error("Exception in pefile.PE('%s') - %s", pe_file, exc)
            return

    def align_down_p2(number):
        return 1 << (number.bit_length() - 1) if number else 0

    def align_up(number, boundary_p2):
        assert not boundary_p2 & (boundary_p2 - 1), \
            "Boundary '%d' is not a power of 2" % boundary_p2
        boundary_p2 -= 1
        return (number + boundary_p2) & ~ boundary_p2

    def get_dirs_status():
        dirs_status = 0
        for idx in range(min(exe.OPTIONAL_HEADER.NumberOfRvaAndSizes, 16)):
            if exe.OPTIONAL_HEADER.DATA_DIRECTORY[idx].VirtualAddress:
                dirs_status |= (1 << idx)
        return dirs_status

    def get_complexity():
        complexity = 0
        if section.SizeOfRawData:
            complexity = (len(compress(section.get_data())) *
                          7.0 /
                          section.SizeOfRawData)
```

```
            complexity = 8 if complexity > 7 else int(round(complexity))
        return complexity

    characteristics_mask = 0b0111111100100011
    data_directory_mask = 0b0111111001111111

    data = [
        pack('> H', exe.FILE_HEADER.Characteristics & characteristics_mask),
        pack('> H', exe.OPTIONAL_HEADER.Subsystem),
        pack("> I", align_down_p2(exe.OPTIONAL_HEADER.SectionAlignment)),
        pack("> I", align_down_p2(exe.OPTIONAL_HEADER.FileAlignment)),
        pack("> Q", align_up(exe.OPTIONAL_HEADER.SizeOfStackCommit, 4096)),
        pack("> Q", align_up(exe.OPTIONAL_HEADER.SizeOfHeapCommit, 4096)),
        pack('> H', get_dirs_status() & data_directory_mask)]

    for section in exe.sections:
        data += [
            pack('> I', align_up(section.VirtualAddress, 512)),
            pack('> I', align_up(section.SizeOfRawData, 512)),
            pack('> B', section.Characteristics >> 24),
            pack("> B", get_complexity())]

    if not isinstance(pe_file, PE):
        exe.close()
    data_sha256 = sha256("".join(data)).hexdigest()

    return data_sha256


if __name__ == '__main__':
    import sys
    if len(sys.argv) < 2:
        print "Usage: pehashng.py path_to_file"
        sys.exit(0)
    print pehashng(sys.argv[1]), sys.argv[1]
```