Pre-processing Memory Dumps to Improve Similarity Score of Windows Modules

Miguel Martín-Pérez[a], Ricardo J. Rodríguez[a,*], Davide Balzarotti[b]

[a] *Dpto. de Informática e Ingeniería de Sistemas*
Universidad de Zaragoza, Spain
[b] *EURECOM, France*

Abstract

Memory forensics is useful to provide a fast triage on running processes at the time of memory acquisition in order to avoid unnecessary forensic analysis. However, due to the effects of the execution of the process itself, traditional cryptographic hashes, normally used in disk forensics to identify files, are unsuitable in memory forensics. Similarity digest algorithms allow an analyst to compute a similarity score of inputs that can be slightly different. In this paper, we focus on the issues caused by relocation of Windows processes and system libraries when computing similarities between them. To overcome these issues, we introduce two methods (GUIDED DE-RELOCATION and LINEAR SWEEP DE-RELOCATION) to pre-process a memory dump. The goal of both methods is to identify and undo the effect of relocation in every module contained in the dump, providing *sanitized* inputs to similarity digest algorithms that improve similarity scores between modules. GUIDED DE-RELOCATION relies on specific structures of the Windows PE format, while LINEAR SWEEP DE-RELOCATION relies on a disassembling process to identify assembly instructions having memory operands that address to the memory range of the module. We have integrated both methods in a Volatility plugin and evaluated them in different scenarios. Our results demonstrate that pre-processing memory dumps with these methods significantly improves similarity scores between memory modules.

Keywords: similarity digest algorithms, memory forensics, Windows, relocation

1. Introduction

Memory forensics is a branch of the computer forensics process, normally carried out as part of the detection and analysis stage in an incident response process (Cichonski et al., 2012). In particular, memory forensics, unlike disk forensics, deals with the recovery of digital evidence from computer memory instead of computer storage media.

The recovery of evidence from volatile instead of non-volatile storage is useful in scenarios where encrypted or remote storage media are used, enhancing the traditional forensic techniques that are more focused on non-volatile media (Ligh et al., 2014). For instance, memory forensics enables a forensic examiner to retrieve encryption keys or to analyze malicious software (malware) that solely resides in RAM. Furthermore, the initial triage in memory forensics is faster than in persistent storage forensics since the quantity of data to be analyzed is smaller.

The memory of a system can be acquired in different ways, depending on the underlying operating system and the hardware architecture. A recent complete survey of state-of-the-art memory acquisition techniques is given in (Latzo et al., 2019). In memory forensics, the current state of the system's memory is acquired and saved to disk as a *memory dump* file, which is later taken off-site and analyzed with dedicated software (such as Volatility (Walters and Petroni, 2007), Rekall (Rekall, 2014), or Helix3, to name a few) to seek evidence of the security incident. Among other items (such as logged users or open sockets), a memory dump contains data regarding the running processes in the system at the acquisition time. Any data susceptible to analysis in the memory dump is called a *digital artifact*.

A memory forensic analyst can triage the list of processes running at the acquisition time to discard well-known processes or to focus her attention on particular ones. Thus, she needs some way to identify processes. In disk forensics, cryptographic hash (one-way) functions (Goldreich, 2006) such as MD5, SHA-1, or SHA-256 functions are commonly used for data integrity and file identification of a seized device (Harichandran et al., 2016). A desirable property of any cryptographic hash function is the avalanche effect property (Webster and Tavares, 1986), which guarantees that the hash values of two similar, but not identical, inputs (e.g., inputs where only a single bit has been flipped) produce radically different outputs. Due to this property, these crypto hash functions are unsuitable for identifying common processes that belong to the same binary application, but in different executions.

A common pitfall is to think that the content of a running process and its corresponding executable file (i.e., the content of the file stored on the disk) are identical. In fact, the OS loader may apply a number of transformations when the executable file is mapped into memory. For instance, software defense techniques such as Address Space Layout Ran-

*Corresponding author
Email address: rjrodriguez@unizar.es (Ricardo J. Rodríguez)

domization (ASLR) ensure that executable files are mapped in memory regions that are different among consecutive executions (in GNU/Linux and other UNIX-base systems alike) or among consecutive system reboots (in Windows systems). Furthermore, the size of the executable file in the memory may be larger than on disk due to memory alignment issues, as the granularity of the memory subsystem OS manager determines the minimum quantity of memory space that is allocated (for instance, 4 KiB in Windows, macOS, and GNU/Linux).

To overcome these limitations, approximate matching or *similarity digest* algorithms have emerged in recent years as a prominent approach that is more robust against active adversaries than traditional hashing (Harichandran et al., 2016). A similarity digest algorithm identifies similarities between two digital artifacts providing a measure of similarity, normally in the range of $[0, 100]$. This similarity score enables an analyst to find out whether artifacts resemble each other or whether an artifact is contained in another artifact (Breitinger et al., 2014a).

As mentioned above, the differences between processes are mainly motivated by the work of the relocation process as well as the memory subsystem: certain parts of a process which are unused in a given moment can be swapped out from memory to free space for other applications. These differences, in turn, negatively affect the similarity scores provided by the similarity digest algorithms (in some cases even resulting in a similarity close to zero).

To minimize the effect of these differences, in this paper we propose two methods to process the input given to a similarity digest algorithm prior to computing its similarity hash. We focus on Windows OS (Windows, for short), since at the time of writing it is the most predominant target of attacks (AV-TEST GmbH, 2019). Both pre-processing methods undo the work performed by the relocation process, but in different ways: the method called GUIDED DE-RELOCATION relies on particular kernel-space structures that might be contained in the memory dump, while the LINEAR SWEEP DE-RELOCATION method performs a linear sweep disassembly of the binary code of a process. Both methods work on Windows small memory page granularity (4 KiB in size). We have evaluated both methods by comparing the similarity scores generated by the `dcfldd`, `ssdeep`, `sdhash`, and `TLSH` similarity algorithms, and shown that the similarity score is improved when any of these methods is used. Let us remark that our pre-processing methods improve the similarity score in terms of similarity accuracy between compared objects. Otherwise, when no pre-processing methods are applied, the similarity score between similar objects quickly drops even when few bytes are different.

In addition, we have also evaluated to what extent the similarity score of each algorithm considered in the paper is affected by the loading process. We found that these algorithms are particularly sensitive to byte modifications, and that *intelligent* byte modifications can dramatically affect the similarity score for some of these algorithms (such as `ssdeep`).

In summary, the contribution of this paper is two-fold:

- We have developed two pre-processing methods that undo the work performed by the Windows relocation process. These methods can be applied in a memory dump to extract the dumped modules in a way that facilitates the comparison between them. Recall that due to relocation, there are byte changes in the binary code that make the comparison between the same modules coming from different memory dumps difficult. To facilitate this comparison, our methods detect and undo these changes.
- We have evaluated the effect of the Windows program binary loading process on the similarity scores of four similarity digest algorithms widely used in forensics: `dcfldd`, `ssdeep`, `sdhash`, and `TLSH`.

The outline of this paper is as follows. Section 2 introduces previous concepts needed to understand our work. Namely, we describe the structure of Windows applications, the Window memory subsystem, and the similarity digest algorithms evaluated in this paper. Section 3 reviews related work. Section 4 is devoted to the two pre-processing methods proposed in this paper. Section 5 presents our evaluation of both methods. We have also studied the robustness of the similarity digest algorithms to byte changes. Finally, Section 6 concludes the paper and states lines of future work.

2. Background

This section introduces first the structure of Windows applications, how the Windows memory subsystem works and then the similarity digest algorithms that we have evaluated.

2.1. Windows PE

The Windows Portable Executable (PE) format is the standard format used by Windows to represent executable files (Microsoft Software Developer Network, 2019). Windows PE was introduced from WinNT 3.1 onward as a replacement for the previous executable format, Common Object File Format (COFF). As an interesting historical remark, let us remember that COFF was also used on the Unix-based systems before being replaced by Executable and Linkable Format (ELF), the current format of executable files.

The Windows PE format is a data structure defined in the Windows' SDK `WinNT.h` file, divided into different parts as sketched in Figure 1. First, there are the *MS-DOS headers* for backward compatibility. These headers comprise the MS-DOS header and the MS-DOS stub, which is a piece of code to inform that the program binary cannot run in DOS mode. Then come the *PE/COFF headers*, which include the magic bytes of the PE signature as an ASCII string ("PE" followed by two null bytes), the PE file header (defining characteristics of the program binary such as the machine architecture for which the PE file was compiled, the endianness, whether it is stripped, etc.), and optionally the PE optional header.

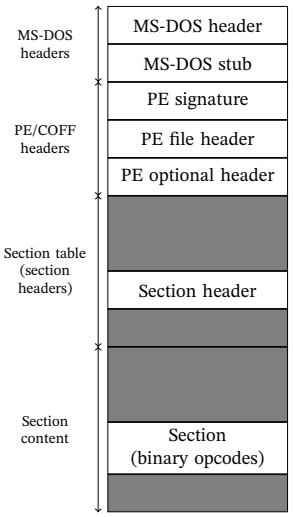


Figure 1: The Windows PE format.

This last part of the PE/COFF headers is only optional for object files, and is always present for executable files. The optional header includes important information about the program binary, such as its preferred virtual memory address and a structure named `DataDirectory`, which contains relevant data such as the export and import directories of the program binary, as well as its relocation table. The relocation table is used by the Windows PE loader to change any instruction or data reference when the program binary is mapped into a virtual address different from its preferred one.

After the PE/COFF headers appear the *Section headers*. For every section contained within the program binary, a section header exists which defines the section size in the binary file as well as in the memory, apart from other characteristics (whether the section data is executable, readable, writable, etc.). Finally, the content of each section follows as a linear byte stream. The starting and ending limits of every section are defined in each section header.

2.2. The Windows Memory Subsystem

Every Windows process has a private memory address space, named *virtual address space*, that defines the set of virtual addresses available for the process. Although a memory address space belongs to a process, a user-space process cannot modify its own address space layout since it is only accessible in kernel-mode.

The default size of the virtual address space of a 32-bit Window process is 2 GiB (prior to Windows 8) (Yosifovich et al., 2017). On 64-bit Windows 8.1 (and later), this size grows theoretically to 128 TiB although in practice it is limited at the time of this writing to less than 24 TiB.

The Windows memory subsystem handles the virtual memory by dividing it into *memory pages* (Huffman, 2015). A memory page is a fixed-length contiguous block of virtual memory. Windows defines two different page sizes: small pages of 4 KiB and large pages that range from 2 MiB (in x86 and x64 architectures) to 4 MiB (in ARM).

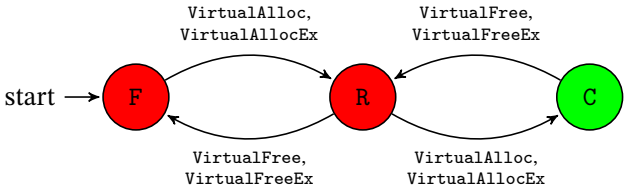


Figure 2: State machine of memory page states in Windows: `F` (free), `R` (reserved), `C` (committed). The state color indicates whether the page is accessible to the process (green) or not (red). Edge labels indicate the Windows system call used to transition between states.

Note that a process virtual address space might be larger (or smaller) than the physical memory on the machine. Thus, the Windows memory subsystem maintains the page table entries to ensure that when a thread in the context of a process reads/writes to a virtual address space, the correct physical address is referenced (Microsoft Software Developer Network, 2018a). The page table entries map a process virtual memory page into a physical memory page.

Apart from maintaining page table entries, the Windows memory subsystem is also responsible for paging memory pages to disk when the memory required by the running threads exceeds the available physical memory. Likewise, it also retrieves the memory paged out to disk and brings it back into the physical memory when needed.

A memory page can be in different states in a process virtual address space (Microsoft Software Developer Network, 2018b). A memory page is *free*, when the page has never been used or is no longer in use. Initially, all memory pages of a process are free pages. A process cannot access a free page (an access violation exception arises if a free page is read), but the process can reserve, commit, or simultaneously reserve and commit pages using Windows APIs such as `VirtualAlloc` or `VirtualAllocEx`. A memory page is *reserved* when the process has reserved memory pages within its virtual address space for future use. Note that a reserved page only guarantees that its range of addresses is unusable by other memory allocation functions, since the page cannot yet be accessed for the process. Furthermore, a reversed page can be committed. A memory page is *committed* when the page has been allocated from the physical memory and is ready to be used by the process.

Windows provides several system calls to manage memory pages at a process level. Figure 2 depicts the states of a memory page and the state transitions (every arrow is labeled with the appropriate system calls). The red states indicate that the memory page is not accessible to the process, unlike the green states. Note that Windows allows a programmer to reserve and commit a memory page with a single call, as well as to free a committed memory page.

2.3. Similarity Digest Algorithms

A *similarity digest algorithm* (also known as approximate matching algorithm (Breitinger et al., 2014a)) identifies similarities between digital artifacts. In particular, the algorithm

outputs a *similarity digest* (also known as *signature*, *fingerprint*, or simply *hash*) which depends on the input data, and can then be compared with other digests. The comparison value provided by the algorithm is named *similarity score*, and normally ranges from 0 to 100, where 0 represents no similarity while 100 means total similarity between the digital artifacts. Note that the similarity score is not a percentage of similarity between artifacts, it is just a value that ranges from 0 to 100.

There are three main categories of similarity digest algorithms (Breitinger et al., 2014a): *bytewise* algorithms, which rely on the raw sequence of bytes of digital artifacts to compute similarity; *syntactic*, when the algorithm relies on internal structures present in the digital artifacts under analysis; and *semantic*, when the algorithm uses contextual attributes to interpret digital artifacts and estimate their similarity.

In this work, we have focused on bytewise similarity digest algorithms. The underlying methods on which bytewise similarity digest algorithms rely characterize the family of the algorithm. *Block Based Hashing* (BBH) algorithms split data into blocks and concatenate their cryptographic hashes to build the output. *Context Trigger Piecewise Hashing* (CTPH) algorithms use parts of the input to decide how data must be split. *Statistically Improbable Features* (SIF) algorithms select the most relevant blocks of the input using statistical methods. *Locality Sensitive Hash* (LSH) algorithms cluster equivalent elements into buckets and compare the number of elements per bucket. In this work, we have selected for evaluation the most relevant algorithm of each family. The selected algorithms are `dcfldd`, `ssdeep`, `sdhash`, and `TLSH`, respectively. We briefly introduce each of these algorithms below.

Before describing the properties of these algorithms that are relevant to understanding our work, we first state the notation used throughout the paper.

Terminology

Inspired by (Baier and Breitinger, 2011; Breitinger et al., 2014a), we define the following general terms to establish a common ground for understanding:

Chunk. A *chunk* is the byte sequence in which the input is divided. Some similarity digest algorithms determine its length dynamically, while others use a fixed length.

Window. A *window* is a sliding fixed-length sequence of bytes that the similarity digest algorithm uses to process the input.

Feature. A *feature* is a unique identifier of a chunk. It is usually the *cryptographic hash* of the chunk.

Compression function. A *compression function* is a function that processes chunks and produces features as output.

Similarity digest. A *similarity digest* is the final output of a similarity digest algorithm. This digest can be compared with other digests. In fact, a digest is an aggregation of features.

Similarity function. A *similarity function* is a function that receives two similarity digests, compares them, and provides a similarity score as output.

Similarity score. A *similarity score* is a value provided by a similarity function that gives insights into the similarity between the similarity digests.

2.3.1. `dcfldd`

The `dcfldd` is a similarity digest algorithm that belongs to the *Block Based Hashing* family. Published in 2002 (Harbour, 2002), it is an enhanced version of the well-known GNU's `dd` command-line tool. `dcfldd` was initially developed to ensure data integrity during forensic imaging.

The compression function of `dcfldd` splits input data into fixed-length chunks (by default, 512 bytes) and computes a *cryptographic hash* (it supports MD5, SHA-1, and almost all SHA-2 variants; MD5 is used by default) as a feature for every chunk. These features are concatenated to form the similarity digest. As a similarity function, it computes the ratio of equal features (i.e., the number of chunks having the same cryptographic hash). Its similarity score ranges in the integer interval $[0, 1, \ldots, 99, 100]$.

The most significant issue of `dcfldd` is the *alignment robustness* problem: just adding (or deleting) a single byte at the beginning of the input changes the offset of all subsequent bytes and thus all the computed features are modified. In addition, when a byte within a chunk changes, its corresponding feature changes dramatically due to the avalanche effect of cryptographic hashes (Webster and Tavares, 1986).

2.3.2. `ssdeep`

`ssdeep` is an example of *Context Triggered Piecewise Hashing* (CTPH), developed by Jesse Kornblum in 2006 to avoid the alignment robustness issue of `dcfldd` (Kornblum, 2006).

`ssdeep` splits input into chunks based on certain bytes of the input itself. In particular, it uses a sliding window of 7 bytes and a customized Alder32 checksum function (Deutsch, Peter and Gailly, J, 1996) to generate a value v that is compared against a *trigger value* t, initially set to $t = 3 \cdot 2^{\lfloor \log_2 \frac{N}{3 \cdot S} \rfloor}$, where N is the size of the input (in bytes) and $S = 64$ is the desirable number of chunks generated by the algorithm. A chunk is built sliding the window through the input and when $v \bmod t = t-1$ holds, the end of the current chunk is identified and the construction of a new chunk is started. When the algorithm ends the 63rd chunk, the remainder of the input makes up the last chunk. In contrast, if the algorithm finishes the input processing with less than 32 chunks, t is halved and the process of identifying chunks starts again. Simultaneously, a second sequence of chunks is

calculated for a value equal to $2t$. The calculation of the second sequence of features increases the possibilities of comparing digests from files of different sizes.

As a compression function, `ssdeep` processes each chunk with the Fowler–Noll–Vo (FNV) function, which is a non-cryptographic hash algorithm with good dispersion properties (Fowler et al., 2011), and encodes the last 6 bits of the FNV output with base64 encoding, providing a single character as a feature. The similarity digest is then built as the concatenation of the value t and the two sequences of features after processing chunks built with t as the trigger value and chunks with $2t$. The digest also ends with the path of the input file. The size of the digests provided in the similarity digest ranges from 67 to 137 bytes, since the number of generated chunks may vary.

As a similarity function, `ssdeep` checks the existence of feature sequences that have the same trigger value in the digests. If no pair exists, then the similarity score drops to 0. Otherwise, if both pairs of feature sequences t and $2t$ have a common trigger value, the similarity score is the maximum of the similarity scores of t and $2t$. To compare two feature sequences, all repeated character sequences are reduced to three characters. Then, they need to share at least a sequence of 7 bytes to reduce the false positive rate. Otherwise, the similarity score will be 0. Finally, the Damerau-Levenshtein edit distance between these sequences is calculated (Wallace, 2015), and its value is normalized between 0 and 100. This normalized value is the similarity score between both digests.

2.3.3. `sdhash`

Vassil Roussev introduced in 2010 a new family of similarity digest algorithms that rely on statistically improvable features (Roussev, 2010). In particular, they use entropy to pick features that are very unlikely to occur and store them inside Bloom filters. A Bloom filter is a probabilistic data structure used to test set membership (Bloom, 1970).

In this case, the input is divided into 64-byte overlapping sequences with overlapping bytes (i.e., a 68-byte input has 5 chunks), and `sdhash` yields a normalized entropy for all chunks. Then, each entropy value is mapped to a precedence rank to highlight the improvable chunks (Roussev et al., 2008). This precedence rank is empirically determined as the probability of having a chunk with a given entropy in a given ground truth dataset (the higher the probability, the higher the precedence rank). Chunks with extremely high (over 990) or low (100 or below) entropy values are discarded to reduce the false positive rate. Next, the algorithm iterates the sequence of precedence ranks in a window of 64 elements to calculate the *popularity* of each chunk, selecting the chunk with the lowest value and left-most position in the sequence. Finally, all chunks with a popularity higher than or equal to a threshold t (by default, $t = 16$) are selected as statistically improvable chunks.

`sdhash` uses the cryptographic hash SHA-1 as a compression function over the selected chunks and the features are added into the Bloom filters. Every filter has a maximum of 160 elements. When a filter is full, a new filter is created. The similarity digest is the concatenation of the Bloom filters.

To compute the similarity between two digests d and d', `sdhash` calculates the similarity score as the average of the similarity filter scores. In particular, the similarity filter score between each digest filter with fewer Bloom filters and each filter of the other digest is computed, and the maximum value of these scores is considered for the average. A minimum of 16 elements is needed in a filter, otherwise, the similarity filter score is -1 (Vassil Roussev, 2013).

2.3.4. `TLSH`

J. Oliver et al. developed in 2013 a new similarity digest algorithm based on a Locality Sensitive hashing (LSH) scheme, named `TLSH` (Oliver et al., 2013). LSH was previously used to detect spam emails (Damiani et al., 2004).

`TLSH` populates an array of 128 counting buckets with 1-byte identifiers that are computed for all tri-grams generated using a sliding window of 5 bytes over the input. Then, it calculates the first, second, and third quartile points of the counting buckets. The similarity digest is composed of two parts: first, the header digest (3 bytes), computed as the checksum (modulo 256) of the input, the logarithm of length (modulo 256), and the ratio between the first and third quartiles and between the second and third quartiles (both ratios are in modulo 16); and second, the body digest (32 bytes), which consists of 2 bits per bucket to indicate in which quartile the bucket is contained.

To compare two similarity digests, `TLSH` calculates the weighted Hamming distance between the body digests and sums to it the weighted Hamming distance between the lengths and quartile ratios from the header digests. When the Hamming distance between any of the quartile ratios is greater than one, the value to be summed up is multiplied by 12. This implies that variations of a few bytes in the input can modify the quartile distribution, increasing the odds of changing the quartiles ratios. An increment of 2 units in both quartile ratios implies an increment of 48 units in the similarity score. Note that `TLSH`, unlike the other algorithms described here, works in an opposite manner: a zero similarity score represents that the inputs are nearly identical inputs while the greater the score, the lower the similarity between the inputs.

3. Related Work

Similarity digest algorithms have been applied mainly in the forensics analysis area to identify total or partial files (Harbour, 2002; Kornblum, 2006; Roussev, 2010; Oliver et al., 2013). Likewise, the authors in (Breitinger and Baggili, 2014) proposed these algorithms to identify known files in network traffic. These algorithms have also been proposed to cluster malware, since they allow similarities between binary files to be captured (Li et al., 2015; Upchurch and Zhou, 2015). Unlike our work, these works only consider executable files as they are stored on disk.

In 2014, the NIST published a technical report establishing a common definition and terminology for approximate matching (Breitinger et al., 2014a). The list of desirable properties for a new similarity digest was recently proposed in (Moia and Henriques, 2017). Among others, these properties are a high compression rate, full coverage, ease of digest generation and comparison, obfuscation resistance, random noise resistance, and a GPU-based design to speed up the generation and comparison function. Some of these properties are inherited from cryptographic hashes while others are the consequences of common issues detected among similarity digest algorithms.

Several authors have studied how similarity digest algorithms behave in terms of performance and robustness. Their performance has been extensively studied (Breitinger et al., 2013, 2014b; Breitinger and Roussev, 2014). These articles propose and develop a generic framework to evaluate similarity digest algorithms. The authors consider `ssdeep`, `sdhash`, and `mrsh` for comparison. The robustness of similarity digest algorithms against random byte modification attacks is evaluated in Oliver et al. (2014). In particular, they study the effects of image manipulation, text file manipulation, and executable manipulation, that is the modification of source code before compiling the executable. In this study, the authors evaluate `ssdeep`, `sdhash` and `TLSH`. Finally, in (Pagani et al., 2018) the authors study the similarity score of `sdhash`, `TLSH`, and `mrsh-v2`, which is an enhancement of `ssdeep` that uses a similar feature function with a minimal feature size and Bloom filters to store features (Breitinger and Baier, 2012). In particular, they evaluated the similarity scores in three different scenarios: library identification, different tool-chains and optimizations, and different versions of an application. The authors stated that `sdhash` was better when dealing with compilation tool-chain changes, while `TLSH` is preferable when the changes involve source code modifications.

Similarity measurement is also used as a matching technique for secure computing and forensics applications in other works, but focused on other underlying models instead of program binary code. For instance, in (Nia et al., 2019) the authors used a similarity measurement to evaluate the similarity in attribute-based attack graphs. Likewise, random-walk computation to evaluate similarities between subgraphs has been proposed especially in the context of collaborative recommendation and natural language processing (Fouss et al., 2007; Minkov and Cohen, 2008).

Regarding pre-processing methods, in (Moia et al., 2020) the authors propose excluding common features to enhance the performance of `sdhash` and `mrsh-v2`. In particular, the features that appear above a given threshold are considered as a common feature and are thus discarded. Unlike their method, our methods are independent of the particular digest algorithm used for computing similarity since our pre-processing inputs work in the input rather than in the inner working of the algorithms.

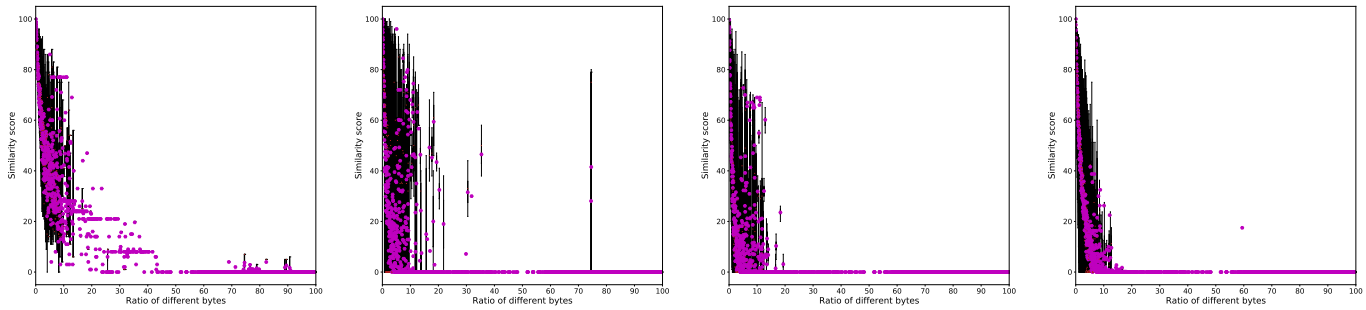
Finally, in (White et al., 2013) the authors propose a pre-processing method that normalizes the bytes affected by the relocation process and the imported functions of a binary file by overwriting the full addresses with constant values. In the first phase, their approach recreates the Windows PE loader, transforming the PE into its virtual layout. The method uses a cryptographic hash to create one signature per memory page and stores the offset of normalized addresses. When they need to validate a memory page, ensuring that the page has been unmodified in the memory, the method uses the stored offsets to normalize the addresses inside the page and then to compare the hash computed over the memory page. Unlike their approach, our pre-processing methods do not need binary files to identify the bytes affected by relocation. In addition, we are less conservative since we normalize the possible bytes affected by relocation (or ASLR) considering 64-byte memory alignment.

4. Pre-Processing Methods

As previously mentioned, the relocation process randomizes the locations of memory segments where a program binary is mapped, including code and data memory segments. Likewise, ASLR ensures this relocation occurs as a software defense technique to thwart control-flow hijacking attacks (Szekeres et al., 2013). ASLR can be seen as a special relocation process. By default, these data and code relocation processes are performed in Windows's system libraries every time the OS is rebooted.

Note that these byte modifications may affect the similarity score of the bytewise similarity digest algorithms introduced in Section 2.3. To assess this possibility, we have evaluated to what extent the relocation processes affect each of these bytewise similarity digest algorithms when comparing similarity between image files. In this regard, we have calculated the similarity score and the number of dissimilar bytes between pairs of memory pages. In particular, we have considered 868,673 comparisons over 44,398 valid memory pages (more details about the experimental data are given in Section 5). Figure 3 shows boxplots of the similarity scores for each algorithm with respect to the ratio of dissimilar bytes. The mean values of the boxplots are plotted with a dot. Our results indicate that more than 46% of the memory pages compared contain dissimilar bytes, and the similarity score of all the algorithms drops quickly when the ratio of different bytes is between 1% and 10%. Note that the similarity score of `TLSH` is normalized to be comparable with the other scores. The normalization process performed is further explained in Section 5. Based on these results, we consider that bytewise similarity digest algorithms do not provide a good degree of confidence for a forensic analysis due to the dropping of the similarity scores when the number of different bytes grows slightly, as well as due to the high score variability.

Let us recall that these byte differences in the image files coming from the same application are due to program binary relocation. In this paper, we propose to pre-process the image files prior to the application of bytewise similarity digest algorithms to mitigate these problems. In particular, we in-



(a) `dcfldd` (b) `ssdeep` (c) `sdhash` (d) `TLSH`

Figure 3: Similarity scores with respect to the ratio of dissimilar bytes. The similarity score of `TLSH` is normalized to be comparable with the other scores.

troduce two pre-processing methods of image files that rely on the available information within the image file.

Problem Statement

Figure 4 sketches the system model of our problem. A memory dump D_i, obtained in a forensically sound manner (e.g., using Volatility (Walters and Petroni, 2007) or Rekall (Rekall, 2014) tools), is pre-processed before extracting the modules $\{m_{i,1}, ..., m_{i,N'}\}$ contained in the dump with any of the pre-processing methods proposed in this work. The extracted modules have been appropriately derelocated and thus the similarity scores of each pair $(m_{i,j}, m_{i,k}), j \neq k, 1 \leq j \leq N', 1 \leq k \leq N'$, are more accurate than when the pre-processing method is not applied. We have highlighted in the figure where the pre-processing methods take place.

To formally present the algorithms of pre-processing methods, we have adopted the notation described in Table 1. Below, we explain in detail how our proposed pre-processing methods work.

4.1. *Pre-Processing Method 1:* GUIDED DE-RELOCATION

We have named the first method GUIDED DE-RELOCATION, since it simply "undoes" the work performed by the Windows OS due to the program binary relocation process guided by information contained within PE modules. Roughly speaking, this method identifies and changes every byte affected by the program binary relocation process by relying on the `.reloc` section of an image file.

The `.reloc` section is a section within the Windows PE structure (see Section 2.1) added to a program binary by the compiler. It contains the necessary information to allow the Windows PE loader to make any adjustment needed in the program binary code and data of the application due to relocation. Recall that the relocation process occurs when the program binary cannot be mapped into its preferred address (because there is already another element mapped to a conflicting region) and thus instructions or variables within the program binary can be relocated with that information.

The information of the `.reloc` section is divided into blocks, where each block represents the adjustments needed for a 4K memory page. Every block contains an `IMAGE_BASE_RELOCATION` structure, which contains the

Notation	Description
$\mathcal{D} = \langle \mathcal{F}_\mathcal{D}, \mathcal{M}_\mathcal{D} \rangle$	A memory dump $\mathcal{D}$.
$\mathcal{F}_\mathcal{D} = \{f_1, ..., f_N\}$	Set of file objects contained in $\mathcal{D}$.
$\mathcal{M}_\mathcal{D} = \{m_1, ..., m_M\}$	Set of modules contained in $\mathcal{D}$.
memory_range(m)	Returns the range of virtual memory addresses of a module m.
PEheader_datadir(m)	Returns the fields of the PE header and data directories of a module m.
points_to(p)	Returns the address pointed by the field p.
derelocate(m, a)	De-relocates the address a in the module m.
file_object(m)	Returns the file object associated to a module m.
sections(f)	Returns the set of section names of a file object f.
copy(m)	Returns a byte copy of the module m.
blocks(f)	Returns the set of blocks of the `.reloc` section of a file object f.
rvaddress(b)	Returns the relative virtual address of the block b.
entries(b)	Returns the set of entries of a block b.
offset(e)	Returns the offset of an entry e.
section_code(m)	Returns the bytes contained in the code section of a module m.
lookup_tables($\mathcal{C}$)	Returns the set of lookup tables contained in $\mathcal{C}$.
32bit_image(m)	Returns a boolean indicating whether the module m is a 32-bit image.
strings_padding($\mathcal{C}$)	Returns the set of (UNICODE and ASCII) strings and padding bytes contained in $\mathcal{C}$.
byte_patterns($\mathcal{C}$)	Returns the set of common byte patterns in $\mathcal{C}$.
subsequent(p)	Returns the subsequent bytes of a pattern p.
build_sequences(b)	Returns the sequences of valid assembly instructions, considering as first byte of each sequence b_i, $0 \leq i \leq 14, b_0 = b$ (see further explanation and example in Section 4.2).
memoperand(i)	Returns the memory operand of an assembly instruction i.

Table 1: Summary of formal notation used in the pre-processing algorithms.

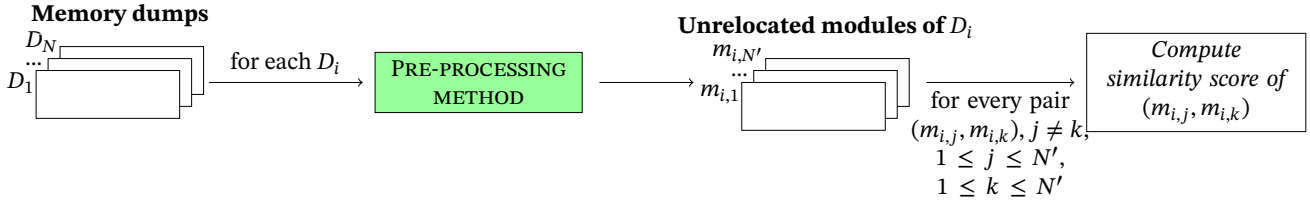


Figure 4: Sketch of the system model. The place where our proposed methods can take place has been highlighted.

RVA of the page and the block size. The block size field is then followed by any number of 2-byte entries (i.e., a word size), which codifies a value that indicates the type of base relocation to be applied (first 4 bits of the word) and an offset from the RVA of the page that specifies where the base relocation is to be applied (the remaining 12 bits).

However, the `.reloc` section is normally stripped off from an image file once the Windows PE loader has relocated it appropriately (Uroz and Rodríguez, 2020). Luckily, a memory dump may contain other elements rather than the image file that represent the image file and do contain such a `.reloc` section, such as `File Objects` (Microsoft Corporation, 2019a). A `File Object` is an internal Windows structure which represents the files mapped into the kernel memory, and acts as the logical interface between the kernel and the user-space and the corresponding data file stored in a physical disk (Yosifovich et al., 2017).

In particular, this kernel-level structure contains a pointer to another structure which in turn is made up of three opaque pointers: `DataSectionObject`, `SharedCacheMap`, and `ImageSectionObject`. An opaque pointer points to a data structure whose contents are unknown at the time of its definition. From these structures, both `DataSectionObject` and `ImageSectionObject` may point to a memory zone where the program binary was mapped either as a data file (that is, containing all its content as in the program binary itself) or as an image file (that is, once the Windows PE loader has relocated). Both memory representations contain the `.reloc` section of the program binary, as stated in (Uroz and Rodríguez, 2020).

Algorithm 1 shows the pseudo-algorithm of the GUIDED DE-RELOCATION pre-processing method. As input, it takes a memory dump $\mathcal{D} = \langle \mathcal{F}_\mathcal{D}, \mathcal{M}_\mathcal{D} \rangle$, where $\mathcal{F}_\mathcal{D} = \{f_1, ..., f_N\}$ and $\mathcal{M}_\mathcal{D} = \{m_1, ..., m_M\}$ are the set of file objects and modules contained in $\mathcal{D}$, respectively. As output, it returns the list of unrelocated modules $\mathcal{U}$ obtained from $\mathcal{D}$. Line 1 initializes $\mathcal{U}$ with empty set. Then, the list of file objects $\mathcal{F}$ is retrieved from $\mathcal{D}$. In this regard, we have used the Volatility plugin `filescan`, which finds `File Object` structures in physical memory using pool tag scanning. Next, we iterate for each module m in $\mathcal{M}_\mathcal{D}$. The list of modules of $\mathcal{M}_\mathcal{D}$ is retrieved using the Volatility plugin `modules`. We first retrieve the range $\mathcal{A}$ of virtual memory addresses of m, obtaining its base address and its image size (which is the size of the program binary in virtual memory; line 4). Then, we walk through the PE structure looking for every PE field which is a memory address pointing to $\mathcal{A}$ (line 5). When found, we leave the two-less significant bytes of such a field unmodified, while zeroing the others (for the sake of brevity, in the following we refer to this process as the *de-relocation* process). We then check if $f \in \mathcal{F}$ such that f corresponds to the retrieved module m and if f has a `.reloc` section. If so, the de-relocation process in m is performed, using the information given by the `.reloc` section of f (lines 6–15). First, a copy m' of the module m is created (line 7). The set of blocks contained in the `.reloc` section is retrieved next, and stored in $\mathcal{B}$ (line 8). Then, for each block $b \in \mathcal{B}$ the RVA of its memory page is taken as a. The set of entries of the block b is stored in $\mathcal{E}$ (line 11). Next, for each entry block $e \in \mathcal{E}$, the memory address a plus the offset of the entry e is unrelocated in the module m' (line 13). Recall that we leave the two-less significant bytes of the address $[a + o]$ in m' unmodified while zeroing the others, since we assume that the relocation process always takes place with 64-byte alignment (as ASLR indeed does (Yosifovich et al., 2017)). Once the de-relocation process finishes, the module m' is added to the set $\mathcal{U}$ (line 16).

Input: A memory dump $\mathcal{D} = \langle \mathcal{F}_\mathcal{D}, \mathcal{M}_\mathcal{D} \rangle$
Output: Set of unrelocated modules $\mathcal{U}$

```
1  U = ∅
2  F ← F_D = {f_1, ..., f_N}
3  foreach m ∈ M_D do
4      A ← memory_range(m)
5      ∀p ∈ PEheader_datadir(m) : points_to(p) ∈
         A ∴ derelocate(m, p)
6      if ∃f ∈ F : file_object(m) =
         f, sections(f) ⋂ {".reloc"} ≠ ∅ then
7          m' ← copy(m)
8          B ← blocks(f)
9          foreach b ∈ B do
10             a ← rvaddress(b)
11             E ← entries(b)
12             foreach e ∈ E do
13                 derelocate(m', a_m + offset(e))
14             end
15         end
16         U = U ⋃ {m'}
17     end
18 end
```

Algorithm 1: GUIDED DE-RELOCATION pre-processing method.

Note that this method, however, does not check in advance whether the memory page corresponding to a block b is paged out (i.e., its content is zeroed). In fact, the time

Input: A memory dump $\mathcal{D} = \langle \mathcal{F}_\mathcal{D}, \mathcal{M}_\mathcal{D} \rangle$
Output: Set of unrelocated modules $\mathcal{U}$

1 $\mathcal{U} = \emptyset, \mathcal{V} = \emptyset$
2 **foreach** $m \in \mathcal{M}_\mathcal{D}$ **do**
3 $\quad \mathcal{V} = \mathcal{V} \bigcup$ empty(m)
4 $\quad \mathcal{A} \leftarrow$ memory_range(m)
`/* Phase 1: structured data processing */`
5 $\quad \mathcal{P} \leftarrow$ PEheader_datadir(m)
6 $\quad \mathcal{V} = \mathcal{V} \bigcup \mathcal{P}$
7 $\quad \forall p \in \mathcal{P} :$ points_to$(p) \in \mathcal{A} \therefore$ derelocate(m, p)
`/* Phase 2: unstructured data processing */`
8 $\quad \mathcal{C} \leftarrow$ section_code(m)
`/* Tag lookup tables */`
9 $\quad \mathcal{L} \leftarrow$ lookup_tables$(\mathcal{C})$
10 $\quad \mathcal{V} = \mathcal{V} \bigcup \mathcal{L}$
11 $\quad \forall l \in \mathcal{L} :$ points_to$(l) \in \mathcal{A} \therefore$ derelocate(m, l)
12 $\quad$ **if** *32bit_image*(m) **then**
$\quad\quad$ `/* Tag strings */`
13 $\quad\quad \mathcal{S} \leftarrow$ strings_padding$(\mathcal{C})$
14 $\quad\quad \mathcal{V} = \mathcal{V} \bigcup \mathcal{S}$
$\quad\quad$ `/* Tag byte patterns */`
15 $\quad\quad \mathcal{B} \leftarrow$ byte_patterns$(\mathcal{C})$
16 $\quad\quad \forall p \in \mathcal{B}, \mathcal{U} \leftarrow$ subsequent$(p) : a =$ points_to$(U) \in \mathcal{A} \therefore$derelocate$(m, a), \mathcal{V} = \mathcal{V} \bigcup \mathcal{U}$
$\quad\quad$ `/* Process the rest of bytes in` $\mathcal{C}$ `*/`
17 $\quad\quad$ **foreach** $b \in \mathcal{C} \setminus \mathcal{V}$ **do**
18 $\quad\quad\quad \mathcal{I} \leftarrow$ build_sequences(b)
19 $\quad\quad\quad I = \{I \in \mathcal{I} : \forall S_i \in \mathcal{I}, S_i \neq I, |S_i| < |I|\}$
20 $\quad\quad\quad \forall i \in I : a =$ memoperand$(i) \in \mathcal{A} \therefore$derelocate$(m, a)$
21 $\quad\quad\quad \mathcal{V} = \mathcal{V} \bigcup \mathcal{I}$
22 $\quad\quad$ **end**
23 $\quad$ **end**
24 $\quad \mathcal{U} = \mathcal{U} \bigcup \{m\}$
25 **end**

Algorithm 2: LINEAR SWEEP DE-RELOCATION pre-processing method.

required to iterate over every entry of the block b is less than the time required to test if the memory page is full of zero bytes. The computational complexity can be expressed as $O(M \cdot F \cdot B \cdot E)$, where M is the number of modules and F is the number of file objects contained in the memory dump, respectively, B is the total number of blocks per file object, and E is the total number of entries per block.

4.2. *Pre-Processing Method 2:* LINEAR SWEEP DE-RELOCATION

Our previous pre-processing method, sketched in Algorithm 1, relies heavily on the existence of `.reloc` sections in the `File Object` structures retrieved from a memory dump. However, this section is not always found in the modules of a memory dump. In addition, it may happen that the physical memory page into which the `.reloc` section was mapped has been outswapped to disk and so it cannot be fully retrieved. Therefore, we propose a second pre-processing method, named LINEAR SWEEP DE-RELOCATION, which works independently from the `File Object` structures.

Algorithm 2 shows the pseudo-algorithm of the new pre-processing method. As input, it takes a memory dump $\mathcal{D} = \langle \mathcal{F}_\mathcal{D}, \mathcal{M}_\mathcal{D} \rangle$, where $\mathcal{F}_\mathcal{D} = \{f_1, ..., f_N\}$ and $\mathcal{M}_\mathcal{D} = \{m_1, ..., m_M\}$ are the set of file objects and modules contained in $\mathcal{D}$, respectively. Line 1 initializes $\mathcal{U}$ and $\mathcal{V}$ (which will be a set to store the bytes marked as *visited*) with empty sets. Then, we iterate for each module m that can be retrieved from $\mathcal{M}_\mathcal{D}$. We use again the plugin `modules` of Volatility. In line 3, we first identify all memory pages of 4096 bytes swapped out from memory by means of the Volatility framework. In particular, we retrieve the memory address space of every module and then check whether the first byte of each memory page is valid. A memory page is valid if it resides in the memory. Every byte of the outswapped memory page is tagged as visited (that is, they are incorporated to the set $\mathcal{V}$). Then, in line 4 we retrieve the range $\mathcal{A}$ of virtual memory addresses of m, obtaining its base address and its image size (which is the size of the program binary in virtual memory).

This algorithm works in two phases. In the first phase, it processes all the structured data of the PE header of m, walking through the PE structure and tagging every byte within the PE structure as visited byte (lines 5 and 6). In addition, we also look for fields in the PE structure which are memory addresses pointing to $\mathcal{A}$ and if found, the de-relocation process takes place (line 7). As before, we assume that the relocation process always takes place with 64-byte alignment (Yosifovich et al., 2017). In this part of the PE structure processing, the entries of the import address table of the module are *de-relocated* and tagged.

Then, the second phase of the algorithm begins (from line 8 to the end). We first retrieve the memory space $\mathcal{C} \subset \mathcal{A}$ into which the code section of m is mapped (line 8). In this regard, we consider as PE code section the section that has `IMAGE_SCN_CNT_CODE`, `IMAGE_SCN_MEM_READ`, and `IMAGE_SCN_MEM_EXECUTE` section flags (Microsoft Corporation, 2019b). In this phase, our aim is to locate sequences of bytes which are memory addresses targeting to $\mathcal{C}$. Therefore, different work is needed depending on the target architecture of the module m.

Note that the 64-bit mode in Intel introduced a new addressing form named *relative Instruction Pointer addressing* (RIP-relative addressing), which is the default for many 64-bit instructions that reference memory in any of their operands (Intel Corporation, 2016). Therefore, none of the 64-bits instructions contain absolute memory addresses targeting to $\mathcal{C}$ and hence there is no need to locate and de-relocate them. If m is a 64-bit image file, we only need to identify lookup tables of memory addresses targeting to $\mathcal{C}$ and mark them as visited bytes (line 9 and 10). For each entry of these tables, the de-relocation process takes place if the entry targets to $\mathcal{C}$. Note that six bytes would be zeroed in this case, assuming a 64-byte alignment (Yosifovich et al., 2017). The same process is applied when the module m is a 32-bit image file, although zeroing two bytes instead of four.

In addition, if m is a 32-bit image file, a little more of work is needed (line 12 to line 23). We first aim to identify known byte patterns (lines 13 to 16). In this regard, we identify null-terminated UNICODE and ASCII strings in $\mathcal{C}$, looking for sequences of printable characters. We have set a minimum of 5 characters to identify the byte sequence as a string (line 13). Every byte of the identified strings is tagged as a visited byte (line 14). In our experiments, we found that some bytes that

make up a memory address were preceded by easily recognized byte patterns. Therefore, as a next step we identify common byte patterns in $\mathcal{C}$, looking for sequences of bytes such as `FE FF FF FF` or `FE FF` (line 15). For every match, we check if the next two double words are memory addresses that point to $\mathcal{A}$. If so, the de-relocation process takes place. As before, every pattern byte identified and the subsequent addresses are also tagged as visited bytes (line 16).

Finally, the last part of the algorithm (lines 17 to 22) iterates in each byte $b \in \mathcal{C}$ which was not visited by any of the aforementioned processes. For every non-visited byte, we build sequences (in bytes) of valid assembly instruction $\mathcal{I}$. In this regard, we get slices of the contiguous 15 bytes starting at the address of b, considering the maximum length of Intel assembly instructions (Intel Corporation, 2016). Note that we get the contiguous bytes of b, regardless of whether they are visited bytes or not. Then, we get a set of sequences of valid instructions $\mathcal{I}^i$ starting at $b_i, 0 \leq i \leq 14, b_0 = b$, and whose bytes are not already visited (line 18). Our algorithm processes these sequences in an optimized way to avoid redundant disassembling. In particular, we iterate in each instruction of the sequence, marking the beginning of every instruction in an auxiliary structure until we detect an instruction which was previously marked as the beginning of an instruction in another sequence. In such a case, we discard the current sequence of instructions since we have reached a subsequent sequence of instructions already recognized by another previous sequence of instructions and thus, the previous sequence will always be greater in length than the current one. We rely on the Capstone disassembly framework (Capstone) to obtain the valid sequences of instructions.

Then, we select the longest byte sequence of valid assembly instructions $I \in \mathcal{I}$ (line 19). We iterate in each instruction in this sequence, tagging every byte of the instruction as a visited byte and checking if the instruction $i \in I$ has an operand which is a memory address that points to $\mathcal{A}$. If so, the de-relocation process takes place (line 20). This iteration finishes marking every byte of the instructions stored in $\mathcal{I}$ as visited (line 21).

The iteration process of the algorithm ends adding the modified module m to the set of unrelocated modules $\mathcal{U}$ (line 24). Note that unlike Algorithm 1, this algorithm is more complete since $\mathcal{U}$ contains all the modules retrievable from the given memory dump (recall that Algorithm 1 only returns the modules that have a relocation section). The computational complexity can be expressed as $O(M \cdot 4S)$, where M is the number of modules contained in the memory dump and S is the total size of the code section (in bytes) per module. Note that every operation of Algorithm 2 that works on $\mathcal{C}$ is iterating in every byte contained in $\mathcal{C}$. Compared with the previous algorithm, note that $B \cdot E < S$, since blocks and entries are subparts of a module. Likewise, $F \ll S$, since S is much greater than the number of file objects available in a memory dump (about three orders of magnitude greater).

Let us illustrate how the processing of sequences of instructions works by providing an example. Assume the following snippet of real assembly code of a Windows library (instructions are shown in hexadecimal representation and in mnemonics) whose bytes were not identified by any of the previous steps of the algorithm:

```
FC                  CLD
FEFF                ???
FFFF                ???
E8 39000000         CALL 0x1043
8B45 08             MOV EAX,DWORD PTR SS:[EBP+0x8]
E8 A487FFFF         CALL KernelBa.752917F0
C2 0C00             RETN 0xC
90                  NOP
FE                  ???
FFFF                ???
FF00                INC DWORD PTR DS:[EAX]
0000                ADD BYTE PTR DS:[EAX],AL
00CC                ADD AH,CL
FFFF                ???
```

We first get a slice of 15 bytes, starting at byte `FC`. Figure 5 shows the initial condition of a scenario where no bytes in the slice were previously visited. The sequence of valid instructions starting at `FC` is as follows (we consider `0x1000` as the base address of the code snippet):

```
0x1000: cld
```

This sequence is solely one byte. In each iteration in the slice, we have highlighted in yellow the byte considered as the starting byte. As an optimization method, to avoid the selection of bytes that we already know make up some other valid sequence, we define a `length` vector and iterate in each instruction of the sequence, setting a value of `-1` in the `length` vector in the byte following the end byte of the instruction. In this case, the second position in the `length` vector is updated with `-1` to indicate the end of this instruction. In addition, the first component of the `length` vector is updated with the value of 1, which is the length of the sequence of instructions already processed and starting at byte `FC`. Then, we move to the next non-visited byte in the slice and whose length is still set to zero value in the `length` vector.

The byte `FF` is then considered. Since this constitutes an empty sequence of valid instructions, its position in the `length` vector is updated with a `-1` value. The same situation occurs with the following bytes, until the byte `E8` is reached. The valid sequence of assembly instructions is:

```
0x1005: call    0x1043
0x100a: mov     eax, dword ptr [rbp + 8]
0x100d: call    0xffffffffffff97b1
0x1012: ret     0xc
0x1015: nop
```

In this case, the sixth component of the `length` vector is updated with the value of 17, while the eleventh and fourteenth components are updated with the value of `-1`. However, since the next instruction in the sequence is out of the current slice, the `length` vector is no longer updated. Starting the disassembly in byte `39`, the sequence of valid instructions is:

```
0x1006: cmp     dword ptr [rax], eax
0x1008: add     byte ptr [rax], al
0x100a: mov     eax, dword ptr [rbp + 8]
0x100d: call    0xffffffffffff97b0
0x1012: ret     0xc
0x1015: nop
```

Here, first the ninth component is updated with a value of -1 (the beginning of instruction add byte ptr [rax], al). When processing the next instruction, our optimization algorithm sees that the instruction at byte 0x1008 has already been visited by a previous sequence of instructions. In this case, the processing of this sequence is skipped and the seventh component of the vector is updated with -1. Therefore, the rest of the instructions in the sequence after the third instruction are no longer disassembled.

Then, we move to the byte 00, obtaining the following sequence:

```
0x1007: add     byte ptr [rax], al
0x1009: add     byte ptr [rbx - 0x5b17f7bb], cl
0x100f: xchg    edi, edi
0x1011: inc     edx
0x1013: or      al, 0
0x1015: nop
```

This sequence has a size of 14 bytes. However, since its last instruction is the same as the last instruction in the previous sequence, its component is updated with -1. Note that the tenth component of the vector is also updated with a value of -1. The next byte to be considered as a starting byte is therefore 45:

```
0x100b: inc ebp
0x100c: or      r8b, r13b
0x100e: movsb   byte ptr [rdi], byte ptr [rsi]
0x100f: xchg    edi, edi
0x1011: inc     edx
0x1013: or      al, 0
0x1015: nop
```

Again, its component in the length vector is updated with -1 since the fourth instruction in the sequence has already been processed. Prior to reaching the repeated instruction, the thirteenth and the last component are also set to the value of -1.

Since all bytes in the slice have been checked, now the sequence starting at the sixth byte is considered as the longest sequence of instructions. All the bytes in the slice are now marked as visited, as well as all the bytes that make up the sequence obtained starting at byte E8. In addition, if any of the instruction in this sequence has a memory operand that targets to $\mathcal{A}$, its address is de-relocated. The next slice of 15-byte length would start at the byte FE which follows the NOP assembly instruction, since all the previous bytes are now marked as visited.

4.3. Implementation

We have implemented both pre-processing methods in a plugin for the Volatility memory analysis framework (Walters, 2007). Our plugin, called Similarity Unrelocated Module (SUM), is an improvement of our previous tool introduced in (Rodríguez et al., 2018). We have released the code of SUM under GNU Affero GPL version 3 license in (Martín-Pérez, 2020).

Unlike our previous tool, SUM yields a similarity digest for every non-zero memory page from every module which is retrieved from a memory dump, while the comparison is an array of similarity scores by pages. The forensic analyst can choose to pre-process every module with the GUIDED DE-RELOCATION method only (when the .reloc section is retrievable), with the LINEAR SWEEP DE-RELOCATION method only, or with both (it first tries to recover the .reloc section to apply the first pre-processing method, and applies the second method if it fails). By default, SUM applies no pre-processing method.

The plugin also supports the use of more than one similarity digest algorithm at once, the selection of only specific sections of the modules for similarity comparison, and selecting processes by PID or processes and shared libraries by name.

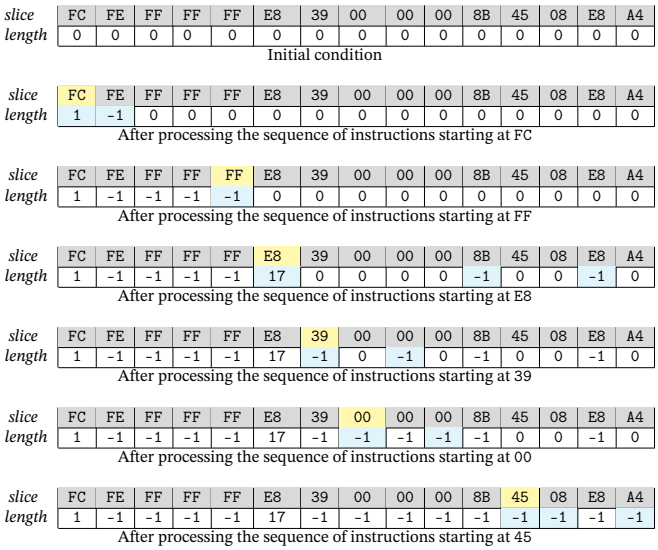


Figure 5: Example of selection of the longest sequence of instructions, per 15-byte slices.

5. Experiments and Discussion

In this section, we assess our pre-processing methods measuring the similarity between modules with different similarity digest algorithms (specifically, dcfldd, ssdeep, sdhash, and TLSH). Apart from this assessment, we have also studied to what extent the similarity score of each similarity digest algorithm is affected when bytes are changed.

Description of Experiments

As experimental settings, we have considered three versions of Windows (Windows 7 6.1.7601, Windows 8.1 6.3.9600, and Windows 10 10.0.14393) in both 32-bit and 64-bit architectures, running on top of the VirtualBox hypervisor. We acquired the memory of these virtual machines ten minutes after a fresh boot, without interacting with the virtual system. This process was repeated ten times. The virtual machines were rebooted between consecutive memory acquisitions to guarantee that ASLR takes place and thus system files are relocated.

As experimental software, we have used Volatility 2.6.1 and Capstone 4.0.0 for the disassembling process performed

by our LINEAR SWEEP DE-RELOCATION pre-processing method.

For comparison, we have selected three sets of modules such that they have a `.reloc` section and thus are valid for our first pre-processing method: *system libraries*, which are used in almost all processes (we chose `ntdll.dll`, `kernel32.dll`, and `advapi32.dll`); *system programs*, which are system processes common to all Windows OS considered in the evaluation (we chose `winlogon.exe`, `lsass.exe`, and `spoolsv.exe`); and *workstation programs*, which include common workstation software such as `Notepad++` version v7.5.8 and `vlc` version 3.0.4.

For every memory dump, we extracted these modules and computed the similarity hashes under three scenarios: no pre-processing (we termed this as RAW *scenario*), applying the GUIDED DE-RELOCATION pre-processing method (GUIDED DE-RELOCATION *scenario*), and applying the LINEAR SWEEP DE-RELOCATION method (LINEAR SWEEP DE-RELOCATION *scenario*).

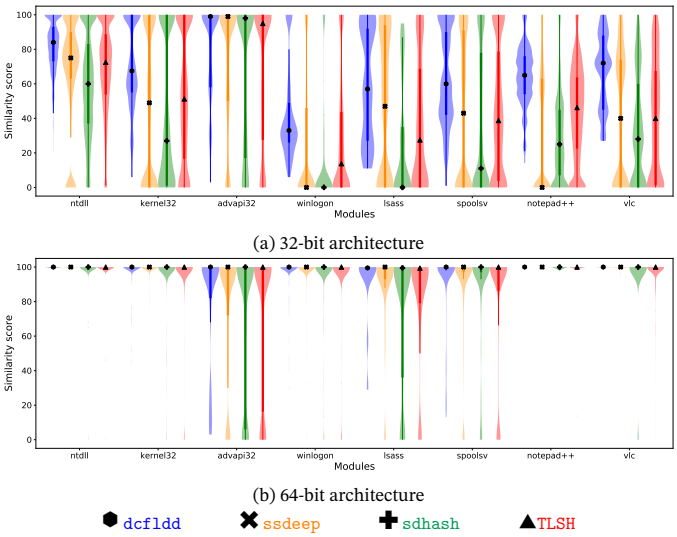
Since our pre-processing methods work mainly on the PE header and the code section of modules, we only consider as input for the similarity digest algorithms the first memory page of every module (this usually contains the PE header since the header size is commonly less than 4KiB) plus the memory pages containing the code section (that is, a subset of the memory pages of every module).

The similarity of the modules is computed as an aggregate similarity score of pairs of memory pages of the modules that are comparable. As similarity digest algorithms, we use `dcfldd`, `ssdeep`, `sdhash`, and `TLSH` (described in Section 2.3). Since the score provided by `TLSH` has an unclear upper limit and works in a reverse mode (the lower the similarity score, the greater the similarity between the inputs), it is difficult to compare `TLSH` with other similarity digest algorithms. Therefore, we normalize the similarity score yielded by `TLSH` as follows.

Based on our experiments, we set the value of 80 as a threshold t to indicate that there is no relation between two memory pages. The value $t = 80$ is near to 85, which is the threshold for comparing versions of program applications proposed in (Oliver et al., 2013). The normalization function is defined as: $\texttt{TLSH}_{norm}(x) = \begin{cases} 100\left(1 - \frac{x}{t}\right) & \text{, if } x \leq t \\ 0 & \text{, otherwise} \end{cases}$

5.1. Related Comparison

In this case, we compare valid (i.e., non-null) memory pages in the same relative offset, using the same pre-processing method. We have considered here all memory dumps. In total, we have 16710 valid pages from 38800 total pages in 32-bit scenarios, and a similar number of valid pages (namely, 16831 valid pages) from 42350 total pages in 64-bit scenarios. On average, the number of non-null memory pages obtained in the memory dumps is around 40%. We plan to study further the process of outswapping memory pages in Windows.



(a) 32-bit architecture

(b) 64-bit architecture

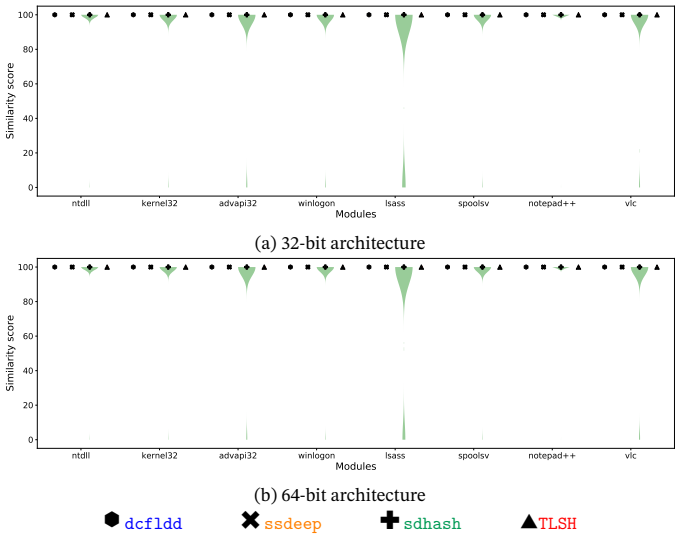
Figure 6: Related comparison: similarity scores when none pre-processing method is applied (RAW *scenario*).

We discuss below the results for each comparison scenario. The results are plotted using violin plots (Hintze and Nelson, 1998), which show the median as an inner mark, a thick vertical bar that represents the interquartile range, and the lower/upper adjacent values to the first quartile and third quartile (the thin vertical lines stretching from the thick bar). For the sake of readability, we have set for each similarity digest algorithm a different mark and color: ⬢ `dcfldd`, ✖ `ssdeep`, ✚ `sdhash`, and ▲ `TLSH`.

Note that we compare memory pages versus memory pages instead of memory pages versus PE files on disk. While the latter comparison would provide a better ground truth, the former comparison is also applicable to situations where data from disk is not present, such as when data comes from packed executables or other types of dynamic-generated code. Note that we compare memory pages versus memory pages instead of memory pages versus PE files as on disk. While the latter comparison would provide a better ground truth, the former comparison is also applicable to situations where data from disk is not present, such as when data comes from packed executables or other types of dynamic-generated code.

RAW *scenario.* Figure 6 shows the aggregated similarity scores considering the sixty memory dumps, for every selected module in 32-bit Windows (upper section of the figure) and in 64-bit Windows (lower section), when no pre-processing method is applied. In total, we have performed a total of 102214 and 99842 comparisons for every algorithm in 32-bit and 64-bit architectures, respectively.

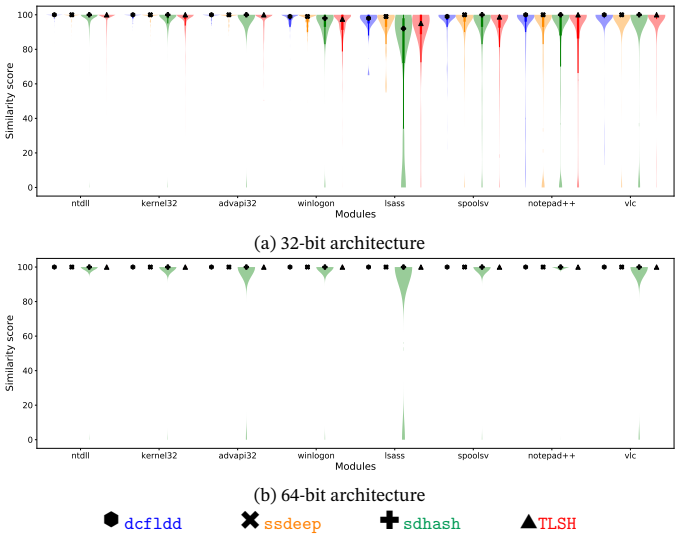
The results in 64-bit architecture are more stable than in 32-bit architecture. Note that the median of the similarity score is near to 100 for all algorithms and all modules. Only the lower adjacent values of `advapi32.dll`, `lsass.exe`, and `spoolsv.exe` have a wider interval. We have manually checked these results and found that they are due to the mod-



(a) 32-bit architecture

(b) 64-bit architecture

Figure 7: Related comparison: similarity scores when the GUIDED DE-RELOCATION pre-processing method is applied (GUIDED DE-RELOCATION *scenario*).



(a) 32-bit architecture

(b) 64-bit architecture

Figure 8: Related comparison: similarity scores when the LINEAR SWEEP DE-RELOCATION pre-processing method is applied (LINEAR SWEEP DE-RELOCATION *scenario*).

ules retrieved from Windows 8. In particular, the dissimilar bytes are caused by lookup tables within the code section of the modules. These good results for 64-bit architecture may be due to the new addressing form introduced with the 64-bit mode in Intel. As explained previously, Intel introduced RIP-relative addressing, which guarantees that no assembly instruction incorporates an absolute memory address within the binary representation of the instruction itself.

In the case of 32-bit architecture, the similarity scores are more disperse and the lower/upper adjacent values are normally all in the range of possible scores, independently of the module or the algorithm. Regarding the values of similarity score, `sdhash` has the lowest score, followed by `TLSH`.

Note that the similarity scores are especially low for 32-bit Windows OS and very good for 64-bit, with some data dispersion for some modules. Nevertheless, we expect that the results in both architectures will be improved when applying our pre-processing methods.

GUIDED DE-RELOCATION *scenario.* Figure 7 shows the results of the similarity score when the GUIDED DE-RELOCATION pre-processing method is applied in the modules of every memory dump. Recall that this pre-processing method is only applicable when the `.reloc` section is retrievable. Although the selected modules have a `.reloc` section, sometimes the memory pages where it was mapped were not present in some of the memory dumps of 32-bit Windows OS machines. In particular, we found this issue when dealing with memory dumps in 32-bit Windows 7. In this case, we performed a total of 72036 and 99842 comparisons for every algorithm in 32-bit and 64-bit architectures, respectively.

The results show that the GUIDED DE-RELOCATION pre-processing method performs particularly well, having the median values at the top of the plots for every algorithm and every module in both architectures. Some outside values appear in the case of `sdhash`. This issue is caused by the `sdhash` way of working. As explained in Section 2.3.3, `sdhash` selects features based on entropy. In addition, the algorithm requires at least 16 features to compare a digest. When this minimum threshold of features is not reached, the similarity score is zero. In our experiments, we found that some memory pages, which were located at the end of the memory address space of the module, contained a few non-zero bytes followed by a large quantity of zero bytes as padding bytes. However, these data are insufficient to yield 16 valid features, so although `sdhash` is able to produce digests, these digests are incomparable. Nevertheless, the number of these outsider values are insignificant.

Note that after applying the GUIDED DE-RELOCATION pre-processing method there are still differences among certain memory pages. However, the percentage of these memory pages is quite low (only 180 out of the 2327720 comparisons). Furthermore, almost all of them occurred in the first memory page of the code section, which usually contains the IAT. We have empirically observed that these changes are caused by the IAT of the modules, which unfortunately was not covered by the `.reloc` section. Note that currently GUIDED DE-RELOCATION does not consider any of the PE related fields. As future work, we will improve our pre-processing method to consider them.

LINEAR SWEEP DE-RELOCATION *scenario.* Last, Figure 8 shows the results of comparisons when the LINEAR SWEEP DE-RELOCATION pre-processing method is applied. As in the first scenario, we performed a total of 102214 and 99842 comparisons for every algorithm in 32-bit and 64-bit architectures, respectively.

As in the previous scenario, the results of the similarity scores are extremely good. Note that we are considering now all the memory dumps of 32-bit Windows OS, unlike the previous scenario in which we discarded the memory modules

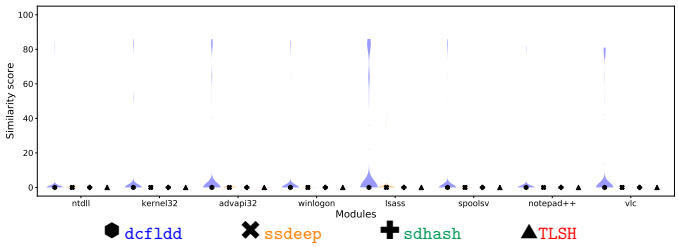


Figure 9: Unrelated comparison: similarity scores aggregated for all Windows OS and scenarios considered.

whose `.reloc` sections are unrecoverable. Thus, comparing these similarity scores with those shown in Figure 6a, it is proved that the application of the LINEAR SWEEP DE-RELOCATION pre-processing method helps to improve the similarity scores of the modules. In 32-bit architecture, the median values range from 90 to 100 while the lower adjacent values are over 80 for all the algorithms, except for `sdhash` and TLSH, which have lower adjacent values of less than 80 for `lsass` and `notepad++` modules. We have manually verified these results, and found that they are caused by memory pages with very limited content and large portions of zero bytes.

Similarly to the previous scenario, the results in the case of 64-bit architecture have almost perfect similarity, having some outsider values in the case of the `sdhash` algorithm. As before, these *almost perfect* results may be motivated due to RIP-relative addressing.

5.2. Unrelated Comparison

In this section, we compare valid memory pages from different modules (but with the same relative offset within the module) using the same pre-processing method. To limit the number of comparisons, we have restricted them to modules coming from the same memory dump. Figure 9 shows the results in this case. As the results are very similar in all systems and architectures, regardless of the pre-processing method applied, we decided to aggregate all the results in a single plot. We performed a total of 990776 and of 1055685 comparisons in 32-bit and 64-bit architectures, respectively.

In this case, only `dcfldd` has similarity scores greater than 0 in some modules, while all the other algorithms find no similarity. We have manually verified these `dcfldd` results and found that they occur because the algorithm considers sequences of zero bytes as relevant data. Therefore, the similarity score of the end padding bytes of memory pages yields a non-zero value.

5.3. Related Comparison with Cross Pre-processing Methods

As with the first comparison method, we now compare valid memory pages with the same relative offset. However, we pre-process the pages to compare using the different methods (either GUIDED DE-RELOCATION or LINEAR SWEEP DE-RELOCATION). The idea of this experiment is to evaluate whether comparing similarity digests from one pre-processing method against the other method is feasible. For

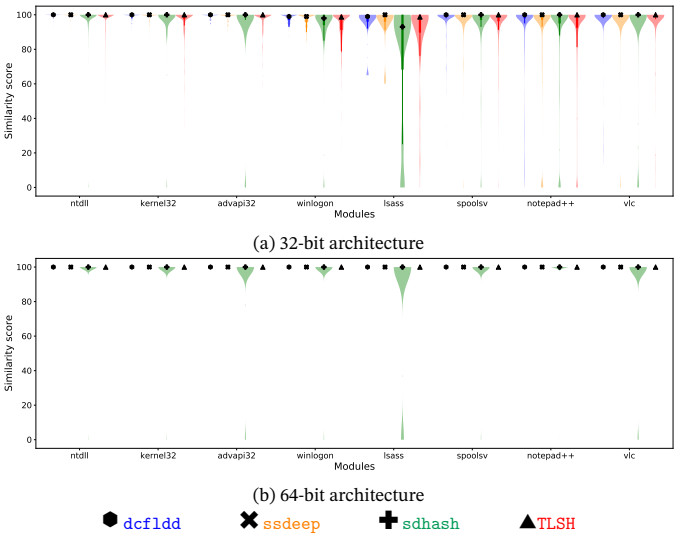


Figure 10: Related comparison with cross pre-processing methods.

this experiment, we performed 160198 and 221967 comparisons in 32-bit and 64-bit architectures, respectively.

Figure 10 plots the results of this experiment. As shown, the results in this experiment are very similar to those obtained in the LINEAR SWEEP DE-RELOCATION scenario of the related comparison experiment. Therefore, these results prove that both pre-processing methods are comparable and that the similarity score results in this experiment are similar to the worst results obtained when applying the pre-processing methods individually.

5.4. Effect of Byte Changes on the Similarity Score

As a final experiment, we have evaluated to what extent the similarity score of each similarity digest algorithm is affected when an arbitrary number of bytes is changed. In particular, we have first grouped the memory page comparisons of the RAW scenario according to the similarity score between them and then looked for the pair of memory pages with the fewest dissimilar bytes in each value of the similarity score.

The results are shown in Figure 11. Each algorithm is plotted with a single line, keeping the colors as in the previous plots for the sake of readability.

It is worth mentioning that the trend seems to be similar in all cases, having different sensitivity to byte differences. The left-side of the plot shows high similarity scores, in which we only appreciate the different sensitivity between the algorithms. In this regard, `dcfldd` is the algorithm that needs more byte changes while TLSH is the algorithm less sensitive to byte changes.

The algorithms tend to show different behavior when the similarity score is under 50 (right-side of the plot). For `ssdeep`, the number of different bytes seems to grow until a similarity score value of 20, with 1200 different bytes. Note that the function seems to be a step-wise function, mainly caused by the granularity of the algorithm: `ssdeep` generates 64 features as a maximum, and thus the number of bytes is

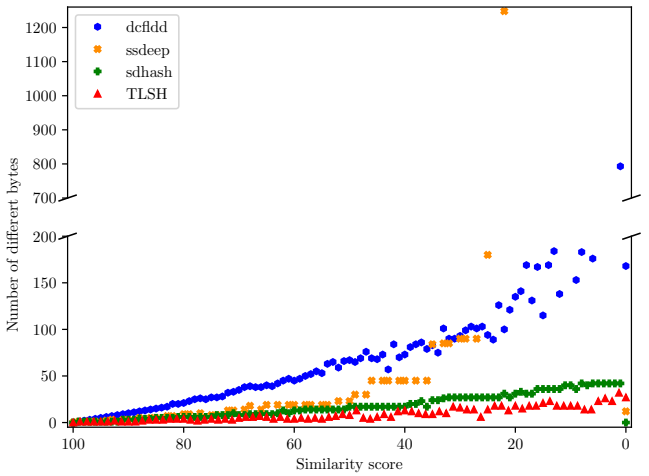


Figure 11: Minimum number of different bytes in a small memory page (4096 bytes) that drops similarity score.

always limited. Likewise, `ssdeep` does not yield any score under approximately 20, as this algorithm requires at least 7 common consecutive features between two digests to reduce false positives (Baier and Breitinger, 2011). Based on our findings, we conclude that the randomness of the affected bytes by relocation causes this necessary condition does not hold and then the similarity score drops quickly to zero.

Similarly, `sdhash` shows stable behavior until the last value. According to the plot, there is one pair of identical pages having zero similarity. This value is caused by the low entropy of the memory pages, as `sdhash` does not select features with low entropy, as well as the number of selected features in the digest being fewer than the minimum number required by `sdhash` to compare a digest. We have empirically corroborated that the generated digests in the last value contain fewer than 16 features and thus they are incomparable, yielding to a similarity score of zero value (Vassil Roussev, 2013). `TLSH` shows the most stable behavior, although it needs fewer different bytes to provide a similarity score of zero value.

6. Conclusions

Memory forensics is part of the incident response process, carried out after a security incident has occurred. Common approaches to identify similar content in files (such as the use of cryptographic hash functions) are unsuitable for identifying similarities between processes or system libraries mainly caused by the relocation process and the memory subsystem operations (memory swapping and demand paging) that provoke small byte modifications and partial content to be zeroed out.

Similarity digest algorithms can be used to overcome these limitations. These algorithms provide a measure of similarity, normally in the range of [0, 100], which enables an analyst to find out whether a memory artifact (data from memory suitable for forensic analysis) resembles another artifact or whether it is contained in other artifacts.

In this paper, we have focused on Windows systems and presented two methods that pre-process the memory modules of a Windows process to undo the work performed by the relocation process in different ways. The method called GUIDED DE-RELOCATION relies on `File Objects`, a particular kernel-space structure that may be found in a memory dump. These structures are useful for identifying the relocated bytes. The other method, called LINEAR SWEEP DE-RELOCATION, performs a linear sweep of the binary code of the given process to identify instructions that contain (absolute) memory addresses as operands. Hence, it also helps to identify relocated bytes too. Both methods have been assessed in different scenarios with different similarity digest algorithms (specifically, `dcfldd`, `ssdeep`, `sdhash`, and `TLSH`). Our evaluation has shown that the similarity score is improved when any of the pre-processing methods is used.

We have also evaluated the sensitivity of these algorithms to byte modifications, and found that *intelligent* arbitrary byte modifications can dramatically affect the similarity score for some of these algorithms (e.g., `ssdeep`). This is an interesting finding that deserves further study.

As future work, we aim to improve the disassembling process of the binary code to better detect the scope of each function. We also aim to incorporate other sections contained in the memory modules, such as memory regions with execution permission, into the input provided to the similarity digest algorithms.

Acknowledgements

The research by Miguel Martín-Pérez and Ricardo J. Rodríguez was supported in part by the Spanish Ministry of Science, Innovation and Universities under grant MEDRESE-RTI2018-098543-B-I00 and by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-20R). The research by Miguel-Martín Pérez was also supported by the Spanish National Cybersecurity Institute (INCIBE) "Ayudas para la excelencia de los equipos de investigación avanzada en ciberseguridad", grant numbers INCIBEC-2015-02486 and INCIBEI-2015-27300. This work was also supported in part by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 771844 BitCrumbs). This research has been developed during a short-research term in EURECOM supported by *Campus de Excelencia Internacional del Valle del Ebro* (Campus Iberus), "Convenio de subvención Erasmus+ Educación Superior para prácticas Consorcio Iberus+", and *Universidad de Zaragoza, Fundación Bancaria Ibercaja y Fundación CAI* "Programa Ibercaja-CAI de Estancias de Investigación", grant number IT 7/19.

References

AV-TEST GmbH, 2019. AV-TEST Security Report 2018/19. [Online, https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2018-2019.pdf.]. Accessed on April 15, 2020.

Baier, H., Breitinger, F., 2011. Security aspects of piecewise hashing in computer forensics, in: 2011 Sixth International Conference on IT Security Incident Management and IT Forensics, IEEE. pp. 21–36.

Bloom, B.H., 1970. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13, 422–426.

Breitinger, F., Baggili, I., 2014. File detection on network traffic using approximate matching. Journal of Digital Forensics, Security and Law 9, 15.

Breitinger, F., Baier, H., 2012. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2, in: International conference on digital forensics and cyber crime, Springer. pp. 167–182.

Breitinger, F., Guttman, B., McCarrin, M., Roussev, V., White, D., 2014a. Approximate Matching: Definition and Terminology. techreport NIST Special Publication 800-168. National Institute of Standards and Technology.

Breitinger, F., Roussev, V., 2014. Automated evaluation of approximate matching algorithms on real data. Digital Investigation 11, S10–S17.

Breitinger, F., Stivaktakis, G., Baier, H., 2013. FRASH: A framework to test algorithms of similarity hashing. Digital Investigation 10, S50–S58.

Breitinger, F., Stivaktakis, G., Roussev, V., 2014b. Evaluating detection error trade-offs for bytewise approximate matching algorithms. Digital Investigation 11, 81–89.

Capstone, . Capstone – the ultimate disassembler. Online, http://www.capstone-engine.org/. Accessed on July 10, 2020.

Cichonski, P., Millar, T., Grance, T., Scarfone, K., 2012. Computer Security Incident Handling Guide. techreport SP 800-61 Rev. 2. National Institute of Standards and Technology (NIST). Special Publication (NIST SP).

Damiani, E., di Vimercati, S.D.C., Paraboschi, S., Samarati, P., 2004. An Open Digest-based Technique for Spam Detection. ISCA PDCS 2004, 559–564.

Deutsch, Peter and Gailly, J, 1996. Zlib compressed data format specification version 3.3. Technical Report. RFC 1950, May.

Fouss, F., Pirotte, A., Renders, J., Saerens, M., 2007. Random-Walk Computation of Similarities between Nodes of a Graph with Application to Collaborative Recommendation. IEEE Transactions on Knowledge and Data Engineering 19, 355–369.

Fowler, G., Noll, L.C., Vo, K.P., Eastlake, D., Hansen, T., 2011. The FNV non-cryptographic hash algorithm. Ietf-draft .

Goldreich, O., 2006. Foundations of Cryptography: Volume 1. Cambridge University Press, New York, NY, USA.

Harbour, N., 2002. Dcfldd. Defense Computer Forensics Lab. http://dcfldd.sourceforge.net 5, 1.

Harichandran, V.S., Breitinger, F., Baggili, I., 2016. Bytewise Approximate Matching: the Good, the Bad, and the Unknown. Journal of Digital Forensics, Security and Law 11.

Hintze, J.L., Nelson, R.D., 1998. Violin Plots: A Box Plot-Density Trace Synergism. The American Statistician 52, 181–184.

Huffman, C., 2015. Chapter 4 - Process memory, in: Huffman, C. (Ed.), Windows Performance Analysis Field Guide. Syngress, Boston, pp. 93 – 127.

Intel Corporation, 2016. Intel® 64 and IA-32 Architectures Software Developer's Manual–Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z. Intel Corporation. Online; https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf. Accessed on July 10, 2020.

Kornblum, J., 2006. Identifying almost identical files using context triggered piecewise hashing. Digital investigation 3, 91–97.

Latzo, T., Palutke, R., Freiling, F., 2019. A universal taxonomy and survey of forensic memory acquisition techniques. Digital Investigation 28, 56–69.

Li, Y., Sundaramurthy, S.C., Bardas, A.G., Ou, X., Caragea, D., Hu, X., Jang, J., 2015. Experimental Study of Fuzzy Hashing in Malware Clustering Analysis, in: 8th Workshop on Cyber Security Experimentation and Test (CSET 15), USENIX Association, Washington, D.C.. p. 8.

Ligh, M.H., Case, A., Levy, J., Walter, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. John Wiley & Sons, Inc.

Martín-Pérez, M., 2020. Similarity Unrelocated Module Volatility plugin. [Online; https://github.com/reverseame/similarity-unrelocated-module]. Accessed on Jul 20, 2020.

Microsoft Corporation, 2019a. FILE_OBJECT structure. [online; https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/ns-wdm-_file_object]. Accessed on September 30, 2019.

Microsoft Corporation, 2019b. PE Format. [Online; https://docs.microsoft.com/en-us/windows/win32/debug/pe-format]. Accessed on Mar, 2020.

Microsoft Software Developer Network, 2018a. Memory Management. [Online; https://docs.microsoft.com/en-us/windows/win32/memory/memory-management]. Accessed on February 15, 2020.

Microsoft Software Developer Network, 2018b. Page State. [Online; https://docs.microsoft.com/en-us/windows/win32/memory/page-state]. Accessed on February 15, 2020.

Microsoft Software Developer Network, 2019. PE Format. [Online; https://docs.microsoft.com/en-us/windows/win32/debug/pe-format]. Accessed on June 3, 2020.

Minkov, E., Cohen, W.W., 2008. Learning Graph Walk Based Similarity Measures for Parsed Text, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, USA. pp. 907–916.

Moia, V.H.G., Breitinger, F., Henriques, M.A.A., 2020. The impact of excluding common blocks for approximate matching. Computers & Security 89, 101676.

Moia, V.H.G., Henriques, M.A.A., 2017. Towards a new approximate matching function for digital forensics investigations, in: X DCA/FEEC/University of Campinas (UNICAMP) Workshop (EADCA), p. 4.

Nia, M.A., Bahrak, B., Kargahi, M., Fabian, B., 2019. Detecting new generations of threats using attribute-based attack graphs. IET Information Security 13, 293–303.

Oliver, J., Cheng, C., Chen, Y., 2013. TLSH–a locality sensitive hash, in: 2013 Fourth Cybercrime and Trustworthy Computing Workshop, IEEE. pp. 7–13.

Oliver, J., Forman, S., Cheng, C., 2014. Using randomization to attack similarity digests, in: International Conference on Applications and Techniques in Information Security, Springer. pp. 199–210.

Pagani, F., Dell'Amico, M., Balzarotti, D., 2018. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis, in: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, pp. 354–365.

Rekall, 2014. The Rekall memory forensic framework. [Online; http://www.rekall-forensic.com/]. Accessed on April 15, 2020.

Rodríguez, R.J., Martín-Pérez, M., Abadía, I., 2018. A Tool to Compute Approximation Matching between Windows Processes, in: Proceedings of the 2018 6th International Symposium on Digital Forensic and Security (ISDFS), pp. 313–318.

Roussev, V., 2010. Data fingerprinting with similarity digests, in: IFIP International Conference on Digital Forensics, Springer. pp. 207–226.

Roussev, V., Richard, G., Marziale, L., 2008. Class-aware similarity hashing for data classification, in: IFIP International Conference on Digital Forensics, Springer. pp. 101–113.

Szekeres, L., Payer, M., Wei, T., Song, D., 2013. SoK: Eternal War in Memory, in: 2013 IEEE Symposium on Security and Privacy, pp. 48–62.

Upchurch, J., Zhou, X., 2015. Variant: a malware similarity testing framework, in: 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), IEEE. pp. 31–39.

Uroz, D., Rodríguez, R.J., 2020. On Challenges in Verifying Trusted Executable Files in Memory Forensics. Digital Investigation Accepted for publication. To appear.

Vassil Roussev, C.Q., 2013. sdhash 3.4. [online; https://github.com/sdhash/sdhash/blob/master/sdbf/sdbf_defines.h#L58]. Accessed on Mar, 2020.

Wallace, B., 2015. Optimizing ssDeep for use at scale. Virus Bulletin. Cited Nov .

Walters, A., 2007. The Volatility framework: Volatile memory artifact extraction utility framework.

Walters, A., Petroni, N., 2007. Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process, in: BlackHat DC.

Webster, A.F., Tavares, S.E., 1986. On the Design of S-Boxes, in: Williams, H.C. (Ed.), Advances in Cryptology — CRYPTO '85 Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 523–534.

White, A., Schatz, B., Foo, E., 2013. Integrity verification of user space code. Digital Investigation 10, S59–S68.

Yosifovich, P., Ionescu, A., Russinovich, M.E., Solomon, D.A., 2017. Windows Internals, Part 1: System architecture, processes,threads, memory management,and more. 7th ed., Microsoft Press, Redmond, WA, USA.