TESIS DE LA UNIVERSIDAD DE ZARAGOZA

2022

68

Miguel Martín Pérez

Effectiveness of Similarity Digest Algorithms for Binary Code Similarity in Memory Forensic Analysis

Director/es

Rodríguez Hernández, Ricardo Julio

http://zaguan.unizar.es/collection/Tesis

ISSN 2254-7606

Prensas de la Universidad
Universidad Zaragoza

© Universidad de Zaragoza
Servicio de Publicaciones

ISSN 2254-7606

Universidad Zaragoza

Tesis Doctoral

EFFECTIVENESS OF SIMILARITY DIGEST ALGORITHMS FOR BINARY CODE SIMILARITY IN MEMORY FORENSIC ANALYSIS

Autor

Miguel Martín Pérez

Director/es

Rodríguez Hernández, Ricardo Julio

UNIVERSIDAD DE ZARAGOZA
Escuela de Doctorado

Programa de Doctorado en Ingeniería de Sistemas e Informática

2022

Repositorio de la Universidad de Zaragoza – Zaguan http://zaguan.unizar.es

Effectiveness of Similarity Digest Algorithms for Binary Code Similarity in Memory Forensic Analysis

Miguel Martín Pérez

Ph.D. DISSERTATION

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Advisors: Dr. Ricardo Julio Rodríguez Fernández

January 2022

Effectiveness of Similarity Digest Algorithms for Binary Code Similarity in Memory Forensic Analysis

Miguel Martín Pérez

Supervisor

Ricardo Julio Rodríguez Fernández	Universidad de Zaragoza, Spain

Dissertation Committee

Frank Breitinger	University of Lausanne (Switzerland)
Juan Caballero	IMDEA Software Institute (Spain)
Raquel Trillo-Lado	Universidad de Zaragoza (Spain)
Martina Lindorfer	Technische Universität Wien (Austria)
Juan Tapiador	Universidad Carlos III de Madrid (Spain)

International Reviewers

Davide Balzarotti	EURECOM (France)
Frank Breitinger	University of Lausanne (Switzerland)

Submitted in compliance with the requirements for the degree of Doctor of Philosophy in Computer Science and Systems Engineering with a Mention "International Doctor" at the Computer Science and Systems Engineering Department, *Escuela de Ingeniería y Arquitectura, Universidad de Zaragoza, Spain*. Jan 3rd, 2022.

Gracias al café porque, en las cantidades adecuadas, puedes hacer las cosas mal mucho más rápido y con más energía. No tengo pruebas, pero tampoco dudas, de que pocas sustancias han hecho tanto por la ciencia y la divulgación.

(RAMÓN NOGUERAS, *Por qué creemos en mierdas*)

Cada vez que tengo que enfrentarme a un momento en la vida en que otros pueden o tienen que juzgarme (una presentación, un examen, la escritura y defensa de una tesis), recuerdo y hago mía una frase que mi tía repite a menudo: *Deseadme suerte... porque si hay justicia, estoy jodida.*

Resumen

Hoy en día, cualquier organización que esté conectada a Internet es susceptible de sufrir incidentes de ciberseguridad y por tanto, debe contar con un plan de respuesta a incidentes. Este plan ayuda a prevenir, detectar, priorizar y gestionar los incidentes de ciberseguridad. Uno de los pasos para gestionar estos incidentes es la fase de eliminación, que se encarga de neutralizar la persistencia de los ataques, evaluar el alcance de los mismos e identificar el grado de compromiso. Uno de los puntos clave de esta fase es la identicación mediante triaje de la información que es relevante en el incidente. Esto suele hacerse comparando los elementos disponibles con información conocida, centrándose así en aquellos elementos que tienen relevancia para la investigación (llamados *evidencias*).

Este objetivo puede alcanzarse estudiando dos fuentes de información. Por un lado, mediante el análisis de los datos persistentes, como los datos de los discos duros o los dispositivos USB. Por otro lado, mediante el análisis de los datos volátiles, como los datos de la memoria RAM. A diferencia del análisis de datos persistentes, el análisis de datos volátiles permite determinar el alcance de algunos tipos de ataque que no guardan su código en dispositivos de persistencia o cuando los archivos ejecutables almacenados en el disco están cifrados; cuyo código sólo se muestra cuando está en la memoria y se está ejecutado.

Existe una limitación en el uso de hashes criptográficos, comúnmente utilizados en el caso de identificación de evidencias en datos persistentes, para identificar evidencias de memoria. Esta limitación se debe a que las evidencias nunca serán idénticas porque la ejecución modifica el contenido de la memoria constantemente. Además, es imposible adquirir la memoria más de una vez con todos los programas en el mismo punto de ejecución. Por lo tanto, los hashes son un método de identificación inválido para el triaje de memoria. Como solución a este problema, en esta tesis se propone el uso de algoritmos de similitud de *digest*, que miden la similitud entre dos entradas de manera aproximada.

Las principales aportaciones de esta tesis son tres. En primer lugar, se realiza un estudio del dominio del problema en el que se evalúa la gestión de la memoria y la modificación de la misma en ejecución. A continuación, se estudian los algoritmos de similitud de *digest*, desarrollando una clasificación de sus fases y de los ataques contra estos algoritmos, correlacionando las características de la primera clasificación con los ataques identificados. Por último, se proponen dos métodos de preprocesamiento del contenido de volcados de memoria para mejorar la identificación de los elementos de interés para el análisis.

Como conclusión, en esta tesis se muestra que la modificación de bytes dispersos afecta negativamente a los cálculos de similitud entre evidencias de memoria. Esta modificación se produce principalmente por el gestor de memoria del sistema operativo. Además, se muestra que las técnicas propuestas para preprocesar el contenido de volcados de memoria permiten mejorar el proceso de identificación de evidencias en memoria.

Preface

Today, any organization that is connected to the Internet is susceptible to cybersecurity incidents and therefore must have an incident response plan. This plan helps prevent, detect, prioritize, and manage cybersecurity incidents. One of the steps to manage an incident is the removal stage, which is responsible to eliminate the persistence of attacks, evaluate the scope of the attacks, and identify the point of compromise. The key point in this phase is identifying by triage the information that is relevant to the incident. This is usually done by comparing the available elements with known information, thus focusing on those elements that are relevant to the investigation (called *evidence*).

This goal can be achieved by studying two sources of information. On the one hand, by analyzing persistent data, such as data from hard drives or USB devices. On the other hand, by analyzing volatile data, such as data from RAM. Unlike the analysis of persistent data, the analysis of volatile data allows determining the scope of some types of attack that do not keep their code on persistence devices or when executable files stored on disk are encrypted; whose code is only shown when it is in memory and being executed.

There is a limitation in the use of cryptography hashes to identify memory evidences, commonly used to identify evidences in persistent data. This limitation is due to the fact that the evidences will never be identical because the execution constantly modifies the memory content. Furthermore, it is impossible to acquire memory more than once with all programs at the same point of execution. Therefore, hashes are an invalid identification method for memory triage. As a solution to this problem, this thesis proposes the use of *Similarity Digest Algorithms*, which measure the similarity between two inputs in an approximate manner.

The main contributions of this thesis are threefold. First, a study of the problem domain is carried out in which memory management and its modification in execution are evaluated. Then, the Similarity Digest Algorithms are studied, developing a classification of their phases and the attacks against these algorithms, correlating the characteristics of the first classification with the identified attacks. Finally, two methods of pre-processing the contents of memory dumps are proposed to improve the identification of the elements of interest for the analysis.

As a conclusion, this thesis shows that the modification of sparse bytes negatively affects the calculation of similarity between evidences from memory. This modification is produced mainly by the operating system memory manager. In addition, it is shown that the proposed preprocessing techniques allow to improve the process of identifying evidences in memory.

Contents

List of Figures v

List of Tables vii

List of Algorithms ix

I Context 1

1 Introduction 3
- 1.1 Motivation . . . 3
- 1.2 Contribution . . . 5
- 1.3 Structure of this Dissertation . . . 6

2 Background 7
- 2.1 The Windows PE Format . . . 7
- 2.2 Randomization of base addresses of executables and DLLs . . . 9
- 2.3 Windows Memory Management . . . 10
- 2.4 The Volatility Framework . . . 11

3 State of the Art 13
- 3.1 Memory Forensics . . . 13
- 3.2 Similarity Digest Algorithms . . . 15
- 3.3 Pre-processing of input data . . . 16

4 On the Effect of Paging and ASLR on Memory Forensics in Windows 17
- 4.1 Effect of the Windows Paging Mechanism . . . 17
  - 4.1.1 Description of Experiments . . . 17
  - 4.1.2 Discussion of Results . . . 19
- 4.2 Effect of ASLR . . . 21

II Similarity Digest Algorithms Classification and Attacks **27**

5 Classification of Similarity Digest Algorithms **29**
5.1 Similarity Digest Algorithms 29
5.2 Terminology, methodology, and background 30
5.2.1 Terminology 30
5.2.2 Methodology 31
5.2.3 The Bloom Filter 31
5.3 Proposed Classification Scheme 32
5.3.1 Phases of a Similarity Digest Algorithm 32
5.3.2 Feature Generation Phase 33
5.3.3 Feature Processing Phase 34
5.3.4 Feature Selection Phase 35
5.3.5 Digest Generation Phase 35
5.3.6 Features Deduplication Phase 36
5.3.7 Digest Comparison Phase 37
5.4 Classification of state-of-the-art SDA 37
5.5 Applicability of the Proposed Scheme 39
5.5.1 `dcfldd` 40
5.5.2 `Nilsimsa` 40
5.5.3 `ssdeep` 41
5.5.4 `md5bloom` 42
5.5.5 `MRS hash` 43
5.5.6 `SimHash` 44
5.5.7 `sdhash` 45
5.5.8 `MRSH-v2` 46
5.5.9 `mvHash-B` 47
5.5.10 `TLSH` 48
5.5.11 `saHash` 48
5.5.12 `LZJD` 49
5.5.13 `FbHash` 50

6 Attacks against Similarity Digest Algorithms **53**
6.1 Adversary model 53
6.2 Classification of SDA attacks 53
6.2.1 Attacks Against the Similarity Score 54
6.2.2 Attacks that bypass any of the phases of an SDA 57
6.3 Towards Building a Robust SDA 58

III Solutions to Minimize Memory-Related Problems **61**

7 Pre-Processing Methods for Normalizing the Variability of Modules **63**
7.1 Pre-Processing Methods 64
7.1.1 Method 1: GUIDED DE-RELOCATION 64
7.1.2 Method 2: LINEAR SWEEP DE-RELOCATION 67

8 Evaluation of Pre-Processing Methods **75**
8.1 The `Similarity Unrelocated Module` Tool 75
8.2 Experiments and Discussion 76

IV Conclusions **83**

9 Conclusions and Open Problems **85**
9.1 Thesis Summary 85
9.2 Detailed Contributions 86
9.3 Future Work and Open Problems 87

List of Figures

2.1 The Windows PE format. (source: https://en.wikipedia.org/wiki/Portable_Executable) . . . 8
2.2 Example of page tables of different processes sharing a dynamic library. 12

4.1 Resident pages of recoverable executable modules at the first (first and third row) and at the second observation moments (second and fourth), with memory workloads of 75%, 100%, and 125% (first, second, and third column, respectively). 20
4.2 Resident pages of recoverable shared dynamic library modules at the first (first and third row) and at the second observation moments (second and fourth), with memory workloads of 75%, 100%, and 125% (first, second, and third column, respectively). . . . 22
4.3 Similarity scores with respect to the ratio of dissimilar bytes. The similarity score of TLSH is normalized to be comparable with the other scores. 23
4.4 Minimum number of different bytes in a small page (4096 bytes) that drops similarity score. . . . 25

7.1 Sketch of the system model. The place where our proposed methods can take place has been highlighted. . . . 63
7.2 Example of selection of the longest sequence of instructions, per 15-byte slices. 73

8.1 Related comparison: similarity scores when none pre-processing method is applied (RAW *scenario*). . . . 78
8.2 Related comparison: similarity scores when the GUIDED DE-RELOCATION pre-processing method is applied (GUIDED DE-RELOCATION *scenario*). 79
8.3 Related comparison: similarity scores when the LINEAR SWEEP DE-RELOCATION pre-processing method is applied (LINEAR SWEEP DE-RELOCATION *scenario*). . . . 80
8.4 Unrelated comparison: similarity scores aggregated for all Windows OS and scenarios considered. . . . 81
8.5 Related comparison with cross pre-processing methods. 82

List of Tables

5.1 Proposed classification for state-of-the-art similarity digest algorithms; the last two columns are discussed in Section 5.4. 32
5.2 Categorization scheme for similarity digests algorithms. The possible values for each dimension/procedure are separated by semicolons. 34
5.3 Classification of similarity digest algorithms according to our proposed classification scheme (*feature generation* and *feature processing* phases). 38
5.4 Classification of similarity digest algorithms according to our proposed classification scheme (*feature selection* and *digest generation* phases). 38
5.5 Classification of similarity digest algorithms according to our proposed classification scheme (*feature deduplication* and *digest comparison* phases). 38

6.1 Sets of characteristics that allow attacks, grouped by attack type and algorithm. 54

7.1 Summary of the formal notation used in the pre-processing algorithms. 65

List of Algorithms

1 Guided De-relocation pre-processing method. 66
2 Linear Sweep De-relocation pre-processing method. 68

Part I

Context

Chapter 1

Introduction

This initial chapter contains the motivation for this dissertation, which is a brief description of the problems related to incident response, followed by a summary of the contributions of the thesis.

1.1 Motivation

Incident response aims to discover what happened in a security incident and, more importantly, preserve incident-related *evidence* that can then be used to take legal action, trying to respond to the known 6 W (*what*, *who*, *why*, *how*, *when*, and *where*) [Cichonski et al., 2012]. An important activity carried out during the incident response process is digital forensics, which in the event of a computer incident is performed on the computers or network where the incident occurred [Granc et al., 2006]. While computer forensics tries to find evidence on computers and digital storage media, network forensics deals with the acquisition and analysis of network traffic.

Computer forensics is divided into different branches, depending on the digital evidence that is analyzed. In particular, this dissertation focuses on memory forensics, which deals with the retrieval of digital evidence from computer memory rather than from computer storage media, as disk forensics does. Memory forensics is useful in scenarios where encrypted or remote storage is used, improving traditional forensic techniques more focused on non-volatile storage [Ligh et al., 2014]. For instance, memory forensics enables a forensic examiner to retrieve encryption keys or analyze malware that resides solely in RAM. In addition, triage in memory forensic is faster since the amount of data to be analyzed is less than in disk forensics. At the same time, it is more accurate as it only contains information related to running processes.

The memory of a computer system can be acquired by different means, which are highly dependent on the underlying operating system and the hardware architecture of the system. A recent and comprehensive review of the latest memory acquisition techniques is provided in [Latzo et al., 2019]. The memory acquisition process retrieves the current state of the system,

reflected as it is in memory, and dumps it into a snapshot file (known as a *memory dump*). This file is then taken off-site and analyzed with dedicated software for evidence (such as Volatility [Walters and Petroni, 2007], Rekall [Rekall, 2014], or Helix3, to name a few).

A memory dump contains data relevant to the analysis of the incident. In forensic terminology, these items are called *memory artifacts* and include items such as running processes, open files, logged in users, or open network connections at the time of memory acquisition. Additionally, a memory dump can also contain recently used artifacts that have been freed but not yet zeroed, such as residual IP packets, Ethernet frames, or other associated data structures [Beverly et al., 2011]. Many of these artifacts are more likely to reside in memory than on disk, due to their volatile nature.

The Windows operating system (Windows, for short) is still the most predominant target of observed malware families [FireEye, 2021]. For this reason, we focus on Windows in this dissertation. In Windows, an executable, shared dynamic library, or driver file that was loaded as part of the kernel or user-mode processes is named *image*, while the file is named an *image file*. Finally, both an image and a process are represented internally by a *module* [Microsoft Docs, 2017]. In what follows, we adhere to this terminology.

In the event of a malware-related security incident, it is likely that malicious modules exist in a memory dump, as the malicious image file and its dependent images were loaded into memory for execution. However, *to what extent can we trust the contents of a memory dump?* This content may be inaccurate due to the way the memory management subsystem works. This inaccuracy problem, called *page smearing*, is particularly visible when we acquire memory on a live system: while the operating system is running, the references to memory in the running processes are constantly updated and, therefore, memory inconsistencies can occur since the acquisition process is usually carried out non-atomically [Pagani et al., 2019]. Furthermore, the size image may be larger than the image file due to memory alignment issues, as the granularity of the memory subsystem operating system manager determines the minimum quantity of memory space that is allocated (for instance, 4 KiB in Windows, macOS, and GNU/Linux).

Additionally, some optimization or defense methods performed by the memory management subsystem and the operating system itself can also affect the data in a memory dump. For example, *page swapping*, which refers to copying pages from the process's virtual address space to the swap device (which is typically non-volatile secondary memory storage), or vice versa. In the same way, *on-demand paging* (also known as deferred paging) delays loading a page from disk until it is absolutely necessary. Finally, the *Address Space Layout Randomization defense* relocates the modules in a random base address each time they are loaded to memory. This security mechanism forces the image loader to adjust the module content on the basis of the new base address. These methods affect the contents of a memory dump since parts of memory are likely to be swapped or not loaded at the time of acquisition. Therefore, false negative results are likely to occur when looking for evidence of malware exclusively in memory.

At the same time, a memory forensic analyst can triage the list of processes and modules running at the acquisition time to rule out known modules or to focus on a few in particular.

Thus, they need to somehow identify the modules of interest. In disk forensics, cryptographic (one-way) hash 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 that have only been changed one bit) produce radically different outputs. Due to this property and page smearing, these crypto hash functions are unsuitable for identifying common modules that belong to the same image file, but in different executions.

Similarity Digest Algorithms (SDA[1]), which are a subtype of Approximate Matching Algorithms, have emerged in recent years as a prominent approach to overcome these limitations [Harichandran et al., 2016]. A similarity digest algorithm identifies similarities between two digital artifacts by providing a measure of similarity, typically in the range of $[0, 100]$. This similarity score allows the analyst to find out if the artifacts resemble each other or if an artifact is contained in another artifact [Breitinger et al., 2014a].

Over the past decade, many similarity digest algorithms have been released to the digital forensics community. However, there is a lack of a clear classification scheme, which makes comparison difficult. Therefore, this dissertation first discusses the relationship between approximate matching algorithms and similarity digest algorithms, which require an intermediate representation (e.g., a fingerprint, a similarity digest) that can be compared. Focusing on SDA, we present a classification scheme that makes easy to describe and compare these algorithms. The proposed classification scheme can be helpful for newcomers, practitioners, and experts to discuss approaches or identify limitations. In addition, we present a study of possible attacks on SDA, correlating each type of attack with the set of characteristics that allow the attack.

However, as mentioned above, there are some differences between modules that are mainly motivated by the work of the relocation process, as well as by the memory subsystem. These differences, in turn, negatively affect the similarity scores provided by the similarity digest algorithms (in some cases they even result in close to zero similarity). Therefore, this thesis proposes different approaches to minimize these differences as much as possible to affect the similarity score as little as possible.

1.2 Contribution

In summary, the contribution of this dissertation is threefold. First, a detailed analysis of how these paging issues affect the user-mode modules that reside in memory on a Windows system with different memory workloads. Then, we study the SDA and present a classification scheme that facilitates their description and comparison, as well as a set of attacks against them considering this classification. We also highlight the desirable properties that an SDA must have to be robust against these identified attacks.

[1] In this dissertation, we use the SDA interchangeably as a singular and plural acronym.

Finally, to minimize the adverse effect of memory behavior on the use of SDA, we propose two methods to process the input given to an SDA before calculating its similarity digest. Both pre-processing methods undo the work done by the relocation process, but in different ways: the method called GUIDED DE-RELOCATION relies on particular kernel-space structures that may be contained in the memory dump, while the LINEAR SWEEP DE-RELOCATION method performs a linear sweep disassembly of the binary code of an image.

In addition, we integrate both methods into a tool to facilitate the calculation of similarity digest in memory dumps. Our tool, dubbed `Similarity Unrelocated Module`, is a plugin for Volatility 2.6 that has been publicly released and licensed under GNU/GPLv3 for the sake of open science and to further research in this area.

1.3 Structure of this Dissertation

After explaining in this chapter the motivation of this thesis, the remaining document is structured as follows. Chapter 2 details key concepts that are relevant to the ideas behind this dissertation. Chapter 3 describes the state of the art in which the current work is contextualized. Chapter 4 characterizes the issues that arise from the Windows Memory Management and the Address Space Layout Randomization defense. These chapters form Part I.

Part II focuses on the Similarity Digest Algorithms. Chapter 5 proposes a classification of the phases and the characteristics of an SDA, applying it to most of the algorithms used in the area. Chapter 6 presents a classification of attacks against SDA, correlating them with the set of previously identified characteristics. This chapter also includes a description of the attacks that are possible and the characteristics that allow attacks to occur for each SDA considered in this dissertation.

Part III describes the proposed solutions to alleviate the problems referred to above and facilitate finding similarities between modules acquired from memory in a more effective way. Chapter 7 describes the two methods proposed in this thesis, while the details of our tool implementation and the experimental results that validate the proposed methods are described in Chapter 8.

Finally, Part IV closes this dissertation with Chapter 9, which summarizes the contributions and lists possible improvements and future work related to this research.

Chapter 2

Background

This chapter introduces some basic concepts that are necessary to follow the rest of this dissertation. We start by defining the Windows PE Format, which specifies the structure of Windows executable files. Next, we describe the base address randomization of executables and DLLs, which creates a private memory address space and maps images to a random location. Finally, we explain the Windows Memory Management.

2.1 The Windows PE Format

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

The Windows PE format is a data structure defined in the `WinNT.h` file of the Windows SDK, divided into different parts as sketched in Figure 2.1. First, there are *MS-DOS headers* for backward compatibility. These headers comprise the MS-DOS header and the MS-DOS stub, which is a code snippet to report that the binary program cannot be run in DOS mode. Next 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—which defines the characteristics of the program binary such as the architecture of the machine the PE file was compiled for, the endianness, if stripped, etc.—, and optionally the *PE optional header*.

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 binary program, such as its preferred virtual memory address and a structure called `DataDirectory`, which contains relevant data such as the export and import directories of the binary program, as well as its relocation table. The *image loader* uses the relocation table to change any instruc-

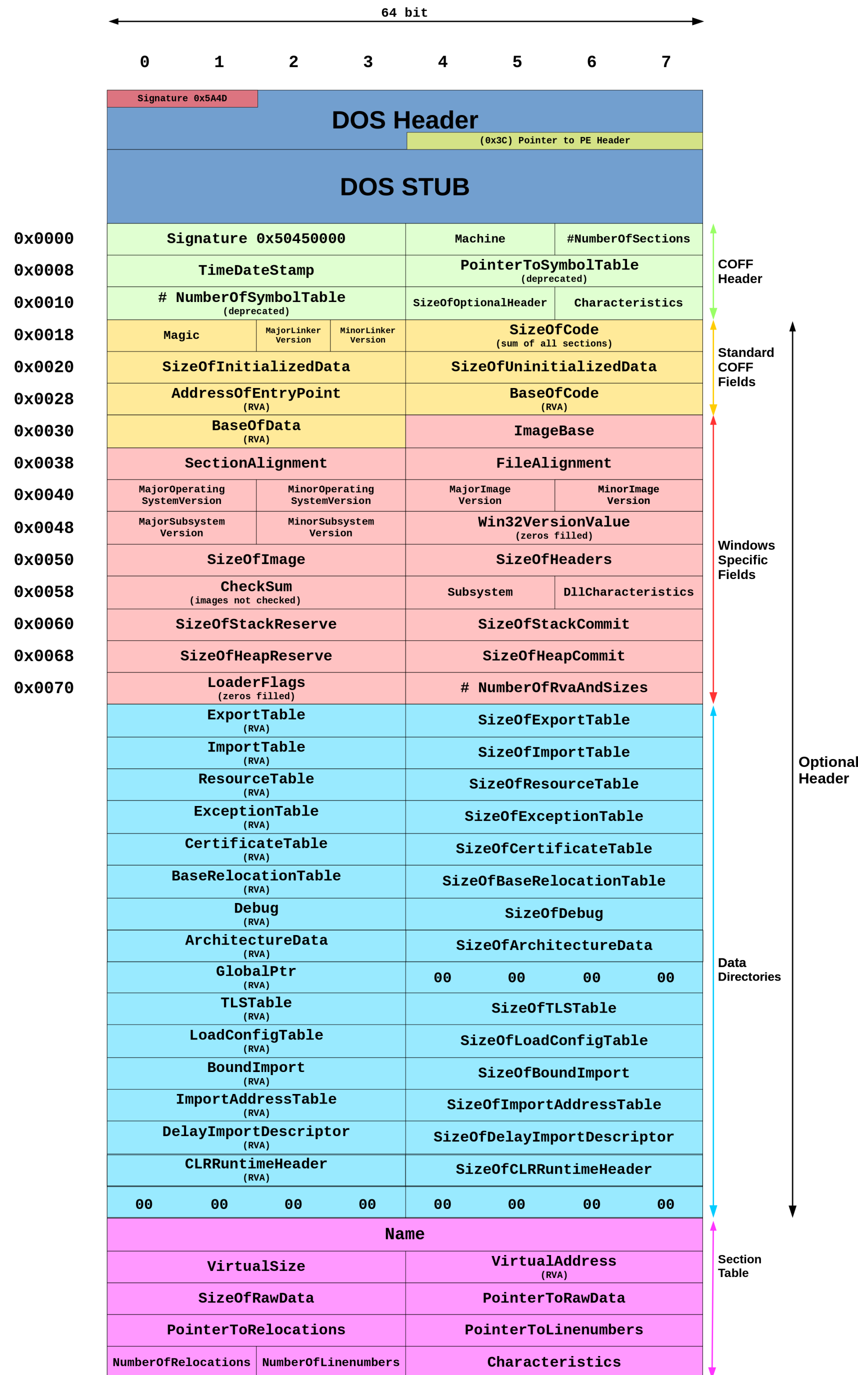


Figure 2.1: The Windows PE format.
(source: https://en.wikipedia.org/wiki/Portable_Executable)

tion or data references when the binary program is mapped to a virtual address other than its preferred one.

The *Section headers* appear after the PE/COFF headers. For each section contained within the binary program, a section header defines the size of the section in the binary file as well as in memory, in addition to other characteristics (if the section data is executable, readable, writable, etc.). Finally, the content of each section follows as a linear byte stream. The start and end limits of each section are defined in the section header itself.

2.2 Randomization of base addresses of executables and DLLs

To create a process in Windows, the system carries out several stages [Yosifovich et al., 2017]: converts and validates parameters and flags; opens the file image; creates a Windows executive process object to represent the process and performs various kernel-related initialization tasks, such as creating the initial process address space; creates the initial thread and its stack and context; performs subsystem-specific initialization; and starts execution of the initial thread, performing initialization in the context of the new process.

Most of the above work is done outside the kernel by the *image loader*, which is found in the user-mode system DLL *Ntdll.dll*. So it behaves like standard code, however it is the first part of the code to run in user mode as part of a new process. One of its functions is to parse the *Import Address Table* (IAT) to find all the DLLs that the process requires, loading and unloading the DLLs at runtime, and maintaining a list of all loaded modules. The IAT is an array of function pointers where the image loader writes the address of the imported function (that is, an external function whose code is provided by a DLL).

The user space where the executable, DLLs, stacks, and heaps are stored is created dynamically, as is the case in the kernel. Typically, the base address where these parts are stored is randomly assigned via the *Address Space Layout Randomization* mechanism (ASLR). The ASLR, combined with non-executable memory pages (in Windows, this protection is called *Data Execution Prevention*), make exploitation of a system through memory manipulation more difficult to achieve.

In Windows, ASLR starts at the image level with the first loaded module, the executable, for the process and its dependent DLLs. For executables, the system relocates the image by an offset by calculating a delta value each time an executable is loaded (if ASLR is enabled in the PE file). For DLLs, the system calculates the image offset globally for the current boot. This value is called *image bias* and is stored in a global memory state structure. In either case, the system has to adjust the code to correctly address the new code and data position. For this purpose, the compiler adds a new section to the image, called `.reloc`, which contains the position of all the required modifications [Microsoft Software Developer Network, 2019b]. Furthermore, all these positions are grouped by the page that they occupy when they are in memory. This structure is due to the lazy evaluation performed by Windows (it does not load a page in memory until it is accessed) [Pietrek, 2019].

2.3 Windows Memory Management

The virtual memory manager is a separate Windows process, primarily responsible for managing physical memory usage. To do this, it tracks each page of physical memory and ensures that when a thread (in the context of a process) reads/writes to addresses in its virtual memory space, it refers to the correct physical addresses thanks to the *page table entries* (PTE), which map a page of process virtual memory to a page of physical memory. The virtual manager will allocate a page of physical memory via a page table entry at the time a process accesses a committed or shareable page [Yosifovich et al., 2017]. As a consequence, any application program operates only with virtual addresses, avoiding the need to use physical addresses.

The memory unit by which Windows manages memory is called the *memory page* [Huffman, 2015]. A memory page defines a contiguous block of virtual memory of fixed length. Page sizes can be small or large. The small page size is 4 KiB, while the large page size ranges from 2 MiB (on x86 and x64 architectures) to 4 MiB (on ARM) [Yosifovich et al., 2017].

Since the virtual address space of the process may be larger than the physical memory on the machine, the Windows memory subsystem must maintain these page table entries to ensure that all reads/writes to virtual addresses refer to the correct physical addresses [Microsoft Docs, 2018a]. Likewise, when the memory required by running processes exceeds the available physical memory, it also sends some pages to disk that are later retrieved by returning them to physical memory when necessary (that is, when accessed).

A page of a virtual address space of a process can be in different states [Microsoft Docs, 2018b]: *free*, when the page has never been used or is no longer used (initial state of all pages of a process). A free page is not accessible for the process but can be reserved, committed, or simultaneously reserved and committed; *reserved*, when the process has reserved some memory pages within its virtual address space for future use. These pages are not yet accessible to the process, but their address range cannot be used by other memory allocation functions; *committed*, when the page has been allocated from RAM and paging files on disk, ready to be used by the process.

In addition, the processes keep track of the virtual addresses that are reserved or used in the process address space through the *Virtual Address Descriptor* (VAD) tree [Yosifovich et al., 2017]. The VAD tree is a self-balancing AVL tree that has a root (named Vadroot) and leafs (named Vadnodes) [Dolan-Gavitt, 2007]. The nodes are created by the virtual memory manager when a memory chunk is allocated and contain the information about this continuous memory region and additional extra information, such as the type of initial access allowed and the type of memory (reserved versus committed versus mapped). This structure is queried when a page is first accessed and the memory manager has to populate the PTE and commit a physical page.

The memory manager uses *lazy evaluation* to improve performance. This means that a physical page is not actually mapped to a virtual address until the virtual address is accessed for the first time, and PTEs are not created until they are queried for the first time. When a process tries to read/write an address in a page that does not reside in memory, a page fault exception is generated, which is detected and handled by the memory manager. There are two causes for

this exception: *i*) The page is accessed for the first time and there is no PTE. Then the memory manager has to query the VAD, create a PTE with this information, commit a physical page, and copy the content (if it exists); *ii*) the page has been paged to disk. In this case, there is a PTE for the virtual address, but the content of the page must be copied from disk to a new page in physical memory.

Shared Dynamic Libraries between Windows Processes

Shared memory is defined as memory that is present in more than one process virtual address space [Yosifovich et al., 2017]. This type of memory is commonly used when two processes use the same dynamic library, as illustrated in Figure 2.2: the library is loaded into physical memory only once, and the corresponding (physical) pages are shared between all processes that map that library. In particular, each process maintains the shared code and data pages of the shared libraries in its virtual address space. When a process loads a shared dynamic library into memory, its associated pages are allocated as copy-on-write. Furthermore, the process exclusively maintains references in its page table to the pages in the shared library that are strictly needed. Figure 2.2 illustrates this behavior between two processes of different file images (otherwise their code pages would also be shared). As shown, *Process A* uses code pages 0, 4, and 5, while *Process B* uses code pages 0, 5, and 6. Similarly, *Process B* maintains a private data page (highlighted in gray). As explained above, the pages are mapped as copy-on-write, that is, a private copy of a particular page will be made only if the process using that page modifies it.

2.4 The Volatility Framework

Volatility is an open-source memory forensics framework for memory investigations, incident responses, and digital forensics [Ligh et al., 2014]. It is implemented in Python but it also has compiled versions for Windows, Linux, and macOS. Additionally, it supports memory dumps from Windows, Linux, and macOS; providing a set of common (and specific) functions for processes, network, or kernel memory. Furthermore, the framework allows developers to contribute with their own plugins, making use of the functionalities provided by Volatility.

This framework provides a multitude of core plugins and functionalities [Volatility Foundation, 2020]. We only briefly describe the ones we use in this dissertation below. Regarding processes and modules, we can select the process by name or PID using the functions `get_proc_by_name()` and `get_proc_by_pid()`, respectively. The function `get_load_modules()` returns a list of load modules in a process, including the main program module. For each module, we can use the `vtop()` function to get the physical page associated with each virtual page that belongs to the module. When this function returns a null pointer means that the page does not reside in memory. The `dlldump` plugin extracts the content of a module from the user space of a process to a file. When a page does not reside in memory, it fills its space with zeros to maintain the alignment of the module content. With regard to files,

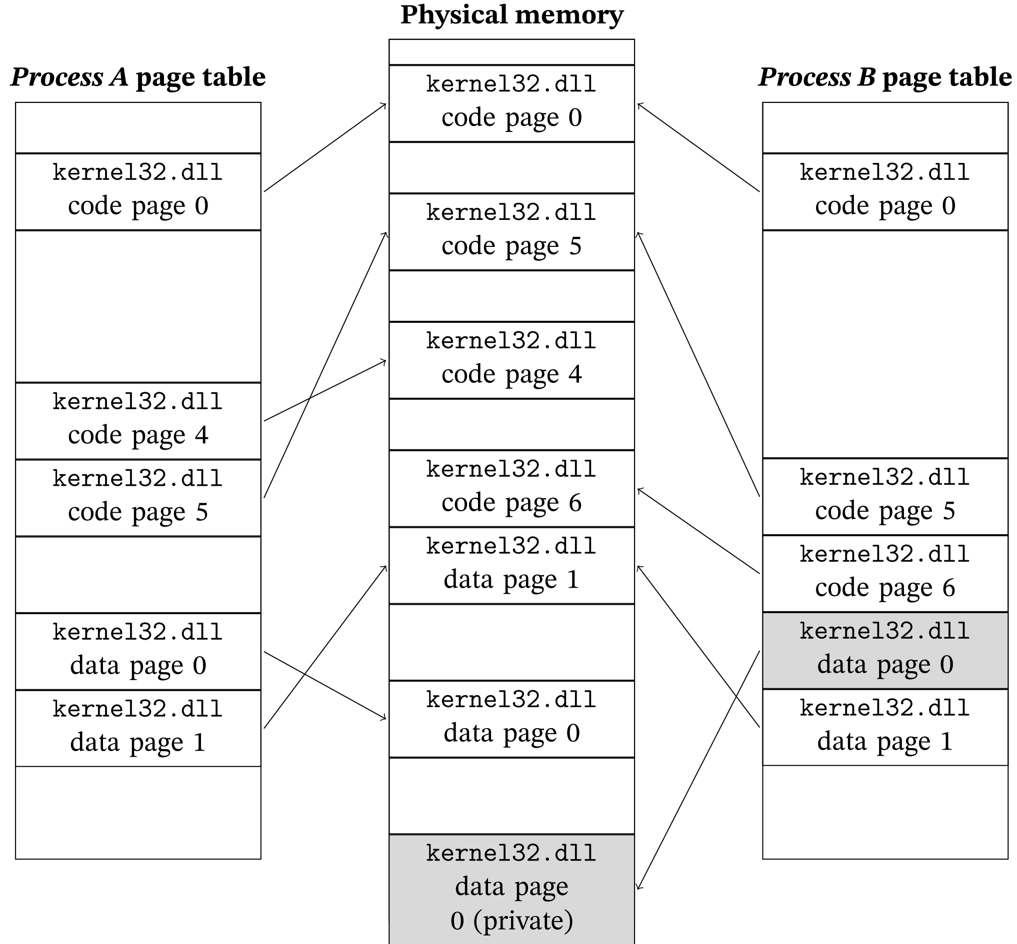


Figure 2.2: Example of page tables of different processes sharing a dynamic library.

the plugin `filescan` finds open files and returns a list of `FILE OBJECT`s in physical memory, which represent the memory mapped files in the kernel memory, acting as the logical interface between the kernel and user-mode processes and the corresponding file data stored on the physical disk [Uroz and Rodríguez, 2020].

The new plugins can use any core function. In addition, they can call other plugins, adjusting the initial parameters for them. The new plugins only need to define one function, `calculate()`, that is called by the Volatility framework to perform the analysis task of the plugin. Other functions can be overwritten with new functionality. For example, the writer function that formats the output can be modified for the new data structure and information generated by the plugins.

Chapter 3

State of the Art

This chapter discusses the state of the art by introducing works of the literature related to the topics covered by this thesis, namely: memory forensics, similarity digest algorithms, and pre-processing of input data. These works are introduced by briefly describing and comparing them with our work.

3.1 Memory Forensics

Forensic analysis of user-space memory has been approached in different ways. The work in [White et al., 2012] introduces an approach based on VAD [Dolan-Gavitt, 2007] that identifies all user allocations and then determines their purpose using kernel and user-space metadata sources. Based on an extensive analysis of the Windows XP SP3 32-bit and Windows 7 SP1 32-bit operating systems, the authors created two Volatility plugins to describe the content of allocations within user-space memory and to verify whether a virtual address of a process not described by a VAD is assigned to a page of physical memory. Paging is an important issue for this approach, as some metadata sources can be paged to disk, thus preventing extraction of their related metadata. This thesis complements this approach by providing insight into the internals of the Windows memory manager with regard to paging.

A utility dubbed `PageDumper` that captures traces of attacks based on runtime memory protection tampering in the Linux operating system is proposed in [Parida and Das, 2020]. Implemented as a kernel module, it helps analyze kernel and user-process address spaces, parsing page table entries in both kernel and user contexts. Rather, we focus on Windows and a post-mortem analysis of a complete memory dump. In any case, `PageDumper` can be a good complement to our solution when analyzing a Linux operating system memory dump.

With regard to malware detection in memory forensics, most works use Virtual Machine Introspection (VMI) techniques to avoid inaccuracy due to memory acquisition on live systems. The fundamental papers in this area are [Hay and Nance, 2008; Nance et al., 2009]. In [Dolan-Gavitt et al., 2011], the authors demonstrated that the memory forensic community can develop

tools using VMI and proceed much more quickly with memory analysis. In this regard, in [Tien et al., 2017] the authors introduce a VMI-based system on top of Xen that can detect malware in virtual machines using Volatility by comparing memory dumps acquired before and after executing a suspicious image file.

Other work focuses on using memory forensics as the basis for malware analysis. The differences between applying YARA signatures to disk or in-memory files and how these can be improved to effectively search for malware in memory are discussed in [Cohen, 2017]. YARA is a very popular open-source and multi-platform tool to identify and classify malware samples. In [Aghaeikheirabady et al., 2014], the effectiveness of different machine-learning classifiers is evaluated using information from VADs, registry hives, and other internal process structures such as EPROCESS. However, the software used to recover these memory artifacts is unclear. The work in [Duan et al., 2015] uses the prevalence of certain dynamic link libraries in processes contained in a memory dump as a characteristic of malicious behavior. The work in [Mosli et al., 2016] presents a machine learning model that uses some features (such as registry keys, imported shared libraries, and called operating system functions) extracted from the reports provided by Cuckoo Sandbox (a sandbox system for malware analysis) to obtain information about a memory dump. Similarly, the work in [Rathnayaka and Jamdagni, 2017] presents an analysis system composed of Cuckoo Sandbox and Volatility in which, as a final analysis step, the results obtained are compared with the results of VirusTotal. The work in [Case et al., 2020] introduces hooktracer_messagehooks, a Volatility plugin that helps analyze hooks in a Windows memory dump to determine if they are associated with a malicious keylogger or with benign software. Finally, the authors introduce a system in [Bozkir et al., 2021] that first uses the Procdump tool, a Microsoft command line tool, to dump processes from memory in Windows 10 version 1903 systems and then converts them into RGB images for classification using machine learning algorithms.

With regard to malware focused on hiding its presence, in [Block and Dewald, 2019] an approach is presented to discover executable pages despite the use of stealthy techniques so that they are not reported by current detection tools. The authors implement it in a plugin for the Rekall memory forensic framework and evaluate it against own implementations of different stealthy techniques, as well as against real-world malware samples. Instead of VAD, this approach relies on PTEs that are listed through paging structures to avoid certain (advanced) stealthy techniques. However, as before this approach does not work if the page tables are paged and the paging file is not provided. A similar work is [Balzarotti et al., 2015], which introduces different techniques that malware can adopt to hide its presence using GPU memory. This work is very interesting, since the malware that resides in that memory cannot leave a trace in the physical memory. The analysis of another type of memory instead of the physical memory, though, is beyond the scope of this dissertation.

3.2 Similarity Digest Algorithms

Prior works primarily focused on directly comparing two or more algorithms and mostly rely on the metrics such as runtime efficiency or precision and recall [Roussev, 2011; Lee and Atkison, 2017]. On the other hand, some researchers dedicated their time to inspecting implementations in all detail. For instance, Breitinger et al. [2012] ran various tests on `sdhash` and found that the implementation does not consider every byte for the similarity digest generation, which they denote by *coverage*. As a result, it is possible that two similarity digests are completely identical despite the artifacts been (slightly) different. Likewise, NIST SP 800-168 discusses various use cases as well as properties of special interest [Breitinger et al., 2014b].

However, there is not a clear, well-established way to provide direct comparisons between all existing algorithms. We aim at filling this gap in this dissertation. To the best of our knowledge, we are the first to establish a classification of SDA to facilitate the description and comparison of these algorithms.

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 this dissertation, 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, 2017b]. 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, 2014c; 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., 2014a]. 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, 2012b]. 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.

3.3 Pre-processing of input data

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 (presented in Chapter 7) are independent of the particular digest algorithm used to calculate the similarity, since our pre-processing inputs work on the input rather than on the inner working of the algorithms.

Finally, the authors in [White et al., 2013] 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 page, making sure that the page has been unmodified in the memory, the method uses the stored offsets to normalize the addresses within the page and to compare the computed hash on the page. Unlike their approach, our pre-processing methods do not need image files to identify the bytes affected by relocation. In addition, we are less conservative as we normalize the possible bytes affected by relocation (or ASLR) considering the 64-byte memory alignment.

Chapter 4

On the Effect of Paging and ASLR on Memory Forensics in Windows

In this chapter, we first study in detail and quantify the effect of paging in Windows user-space modules. Our work provides information on the operation of paging in Windows, which is a problem for the detection of malware through memory forensic analysis [Martín-Pérez and Rodríguez, 2021]. In addition, the effects of the ASLR defense in the similarity score for the most common SDA are also studied. We show how the variability introduced by this defense reduces the usability of these algorithms when the input is not pre-processed [Martín-Pérez et al., 2021a]. Apart from this assessment, we have also studied to what extent the similarity score of each similarity digest algorithm is affected when the bytes are changed.

4.1 Effect of the Windows Paging Mechanism

In this section, we first describe the experiments carried out to quantify and characterize the effect of paging on Windows and then discuss the results.

4.1.1 Description of Experiments

As an experimental scenario, we use a virtual machine with a base installation of Windows 10 64-bit version 19041 running on the VirtualBox 6.1.18 hypervisor with default paravirtualization and large paging disabled. Note that in the default paravirtualization, VirtualBox only supports paravirtualized clocks, APIC frequency reporting, guest debugging, guest crash reporting, and relaxed timer checks. Therefore, the behavior of the guest memory management unit is not affected by paravirtualization. Furthermore, we use a virtual machine to avoid the problem of page smearing, since we are interested in quantifying paging, which is affected by page swapping and on-demand paging. We consider two configurations of physical memory,

4GiB and 8GiB, with an Intel Core i7-6700 3.40GHz dual-core processor. The Internet has been disconnected after updating the machines.

As memory workloads, we consider 25%, 50%, 75%, 100%, 125%, and 150% of the total physical memory. We have developed a simple C tool to allocate the amount of memory needed to reach the specified percentage of memory used (that is, the tool allocates between 1GiB and 6GiB and between 2GiB and 12GiB, for the memory configurations of 4GiB and 8GiB, respectively). In particular, the tool allocates memory and writes a random byte every 4KiB to ensure that pages are constantly used and avoid their paging as much as possible. Note that this tool will consume a large chunk of memory and will leave less space for the pages of other user-space processes, regardless of the use of these pages (recall that both anonymous mappings and file mappings are backed by the system paging file [Chen, 2013]).

Under these conditions, the system memory has been acquired at various runtimes for each memory workload. First, we initialize the virtual machine and wait 5 minutes for the machine to reach a stable state. Immediately after, we pause the virtual machine and acquire the *initial* dump. Then we launch the memory allocation tool explained above and dump the memory every 15 seconds for one minute, pausing and resuming the execution of the virtual machine before and after memory acquisition. After the first minute, we continue to capture the memory every minute for an additional 4 minutes, also pausing and resuming the execution of the virtual machine between memory acquisitions. These memory dumps constitute the *first observation moment*. We then stop the memory allocation process, pause the execution of the virtual machine, and dump the memory using the same pattern: every 15 seconds for the first minute and every 1 minute for the next 4 minutes. These memory dumps are part of the *second observation moment*. Finally, we shut down and reboot the virtual machine to restart the dump process with another memory workload.

For each memory dump, we get the number of recoverable modules and how many resident pages are in each module. The process of memory acquisition and calculation of recoverable data has been replicated 10 times to increase the reliability of the evaluation. We finally took into account the average of the 10 independent repetitions for each recoverable module. To help us obtain the recoverable data from the modules, we have implemented a tool, dubbed `residentmem`, as a Volatility plugin released under GNU/GPL version 3 license and publicly available at [Martín-Pérez, 2021]. The plugin iterates through the processes contained in the memory dump and, for each process, checks the memory pages assigned to each module associated with that process through internal Volatility structures. As input, the plugin needs a memory dump. As output, it returns a list of the recoverable modules with the resident pages and the total number of pages of each module, the process to which the module belongs, the path where it is stored in the file system, and other information of interest related to the module (such as its version, base address, and its process identifier, among other information). In addition, it allows to obtain the specific list of physical pages associated to each resident page of a recoverable module.

The plugin analysis workflow is as follows. We first get the list of processes that were running at the time of memory acquisition through Volatility's internal structures. We then iterate

through this list, accessing its memory address space and validating with a 4096-byte step (the size of a small page) if each memory address is resident. This gives us the number of resident and total memory pages for each process and its related modules.

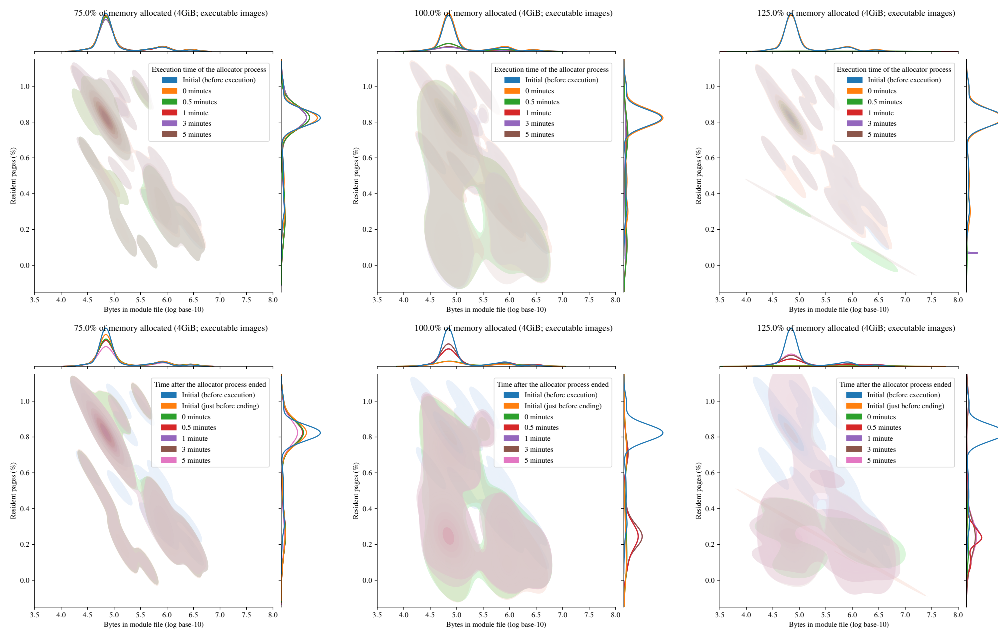
4.1.2 Discussion of Results

We only discuss the 75%, 100%, and 125% memory workloads because we have empirically observed that experiments below 75% and above 125% behave equal to 75% and 125%, respectively. Likewise, we do not plot all the instants of time because otherwise the graphs are overloaded and difficult to understand. For this reason, we only show the *Initial (before execution)* (before any interaction with the system), *0 minutes* (just after interacting with the system, that is, starting or stopping the memory allocation tool); and *0.5*, *1*, *3*, and *5 minutes*, a subset of the observed instants of time that accurately represent the complete behavior.

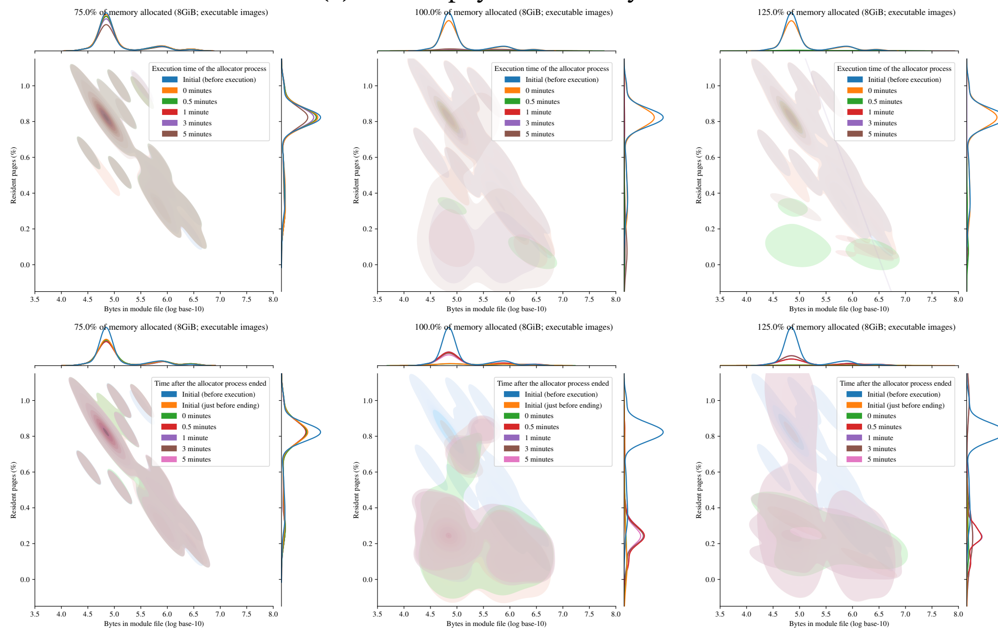
In addition, one more moment, *Initial (just before ending)*, has been incorporated for the graphs relative to the second observation moment to show how the system is just before interacting with it. This instant of time is actually the same as the *5 minutes* instant of the first observation moment. The graphs of both observation moments show *Initial (before execution)* to have a common reference that allows comparing the results.

Modules of executable image files. Figure 4.1 shows the resident pages of the recoverable modules of executable files under different memory workloads for 4GiB and 8GiB of physical memory (Figures 4.1a and 4.1b, respectively), for both observation moments. Each plot shows the distributions of two variables (the size of a module file in log-base 10, on the x-axis, and the percentage of resident pages, on the y-axis) through color intensity. The darker the region, the more data is in that region. The subplots at the top and right of the main plots show a smoothed version of the size and resident pages data frequencies, revealing the distribution of resident pages and module file sizes.

Looking at the first observation moment, the initial conditions show that almost 80% of the executable module pages reside in memory. With a memory workload of 75%, there are no significant changes to the resident pages because there is still enough free memory, regardless of the size of the physical memory. A slight reduction in recoverable modules is observed throughout the acquisition times, which may be motivated by the paging of unused modules, while the resident pages remain constant. Note that the colored areas are mostly identical. With regard to 100% memory workload, in *0.5 minutes* most modules are expelled and the number of resident pages for recoverable modules is drastically reduced. With the 125% memory workload, there is again a large reduction in recoverable modules. In this case, the graph shows two well-defined areas: an area that contains the first two moments of time and another area that contains the remaining rime moments on a diagonal below 60% resident pages. The 8GiB results show greater variability in this workload, as indicated by the larger color areas below the 60% diagonal.



(a) 4GiB of physical memory



(b) 8GiB of physical memory

Figure 4.1: Resident pages of recoverable executable modules at the first (first and third row) and at the second observation moments (second and fourth), with memory workloads of 75%, 100%, and 125% (first, second, and third column, respectively).

With regard to the second observation moment, the results are the same as in the first observation moment with a memory workload of 75%. With the 100% and 125% memory workloads, it is observed that the modules progressively come back to memory, but the ratio of resident pages for recoverable modules never goes above 25%. Significant increases in *0.5 minutes* and in *3 minutes* are seen for both memory configurations, although the number of recoverable modules is less for a 125% workload.

Modules of shared dynamic library image files. Figure 4.2 shows the distribution graphs of resident pages of recoverable modules of shared dynamic libraries under different memory workloads for 4GiB and 8GiB of physical memory (Figures 4.2a and 4.2b, respectively), for both observation moments. In this case, most modules only have 20% of their pages resident and are in the range 10^5 to 10^6 bytes. The maximum percentage of resident pages is 75%.

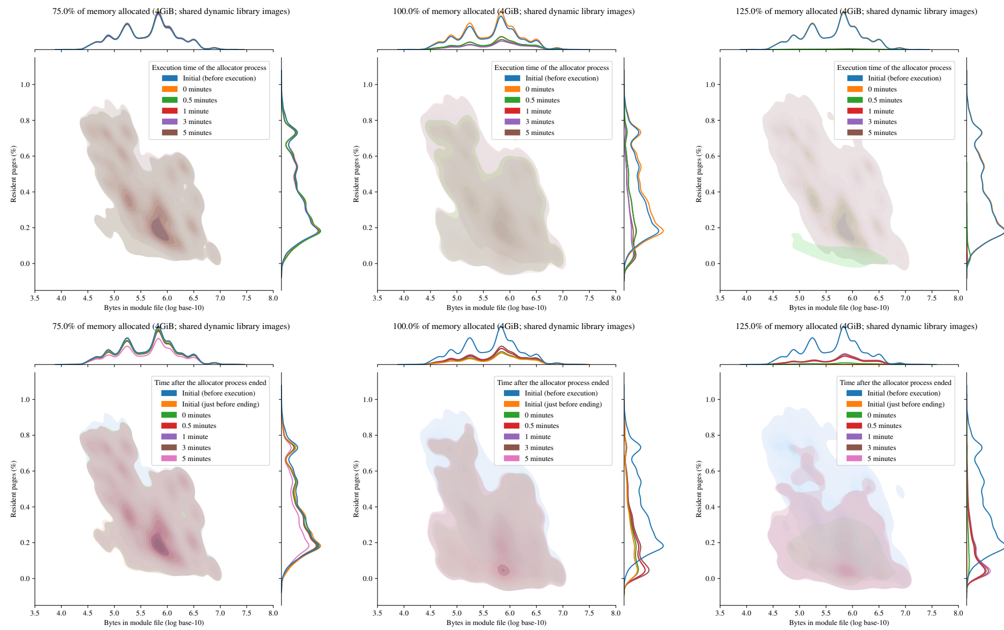
Regarding the first observation moment, no significant changes are observed with 75% of memory workload, similarly to the results of the type of module studied previously. A slight decrease in recoverable modules is observed, but with no effect on resident pages. With 100% of memory workload, the system begins expelling modules for any size in *0.5 minutes*. The number of recoverable modules is reduced, but the distribution shape is similar in both memory configurations. A more aggressive expelling of modules is observed in 8GiB of physical memory. In any case, most modules have only less than 5% of their pages resident. With 125%, the results are very similar. As before, there is a small colored area at the bottom of the graph, which indicates that the percentage of the resident pages is approaching 5% again.

With regard to the second observation moment, the results with a memory workload of 75% are very similar to the results of the first observation moment. With 100% of memory workload, the number of recoverable modules slowly increases, but the percentage of resident pages for most modules is still close to 5%. Similar behavior is observed with the memory workload of 125%, where few modules return to memory but the percentage of resident pages remains close to 5% for most of them.

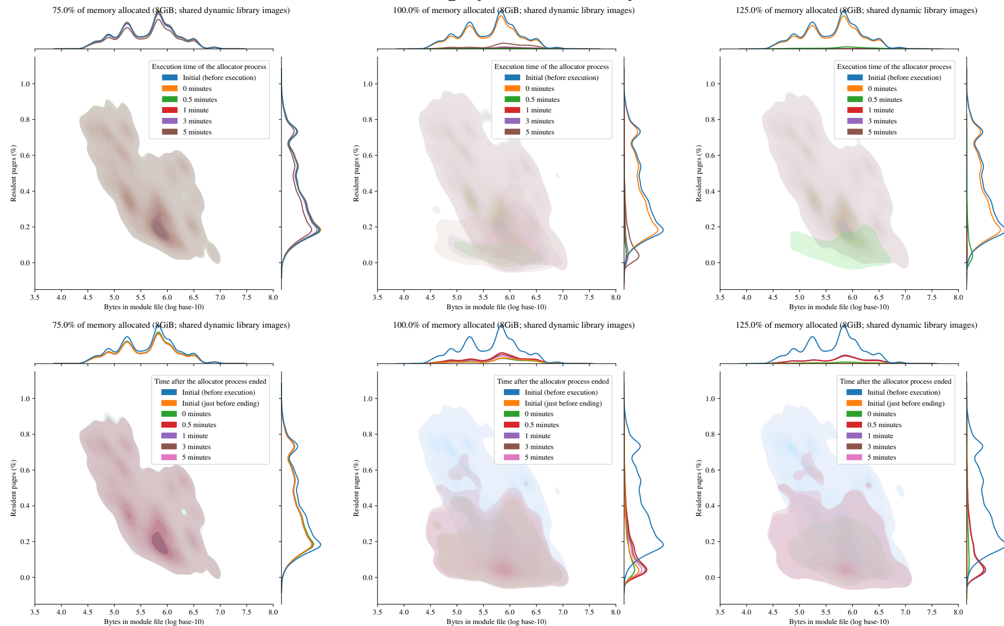
4.2 Effect of ASLR

The relocation process randomizes the memory segment locations where a binary program is mapped, including the 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 code and data relocation processes are performed in the Windows system libraries each time the OS is rebooted.

Note that these byte modifications can affect the similarity score of the bytewise similarity digest algorithms, as discussed in Chapter 5. To assess this possibility, we have evaluated the extent to which the relocation processes affect each of these bytewise similarity digest algorithms when comparing similarity between modules. In this regard, we have calculated the



(a) 4GiB of physical memory



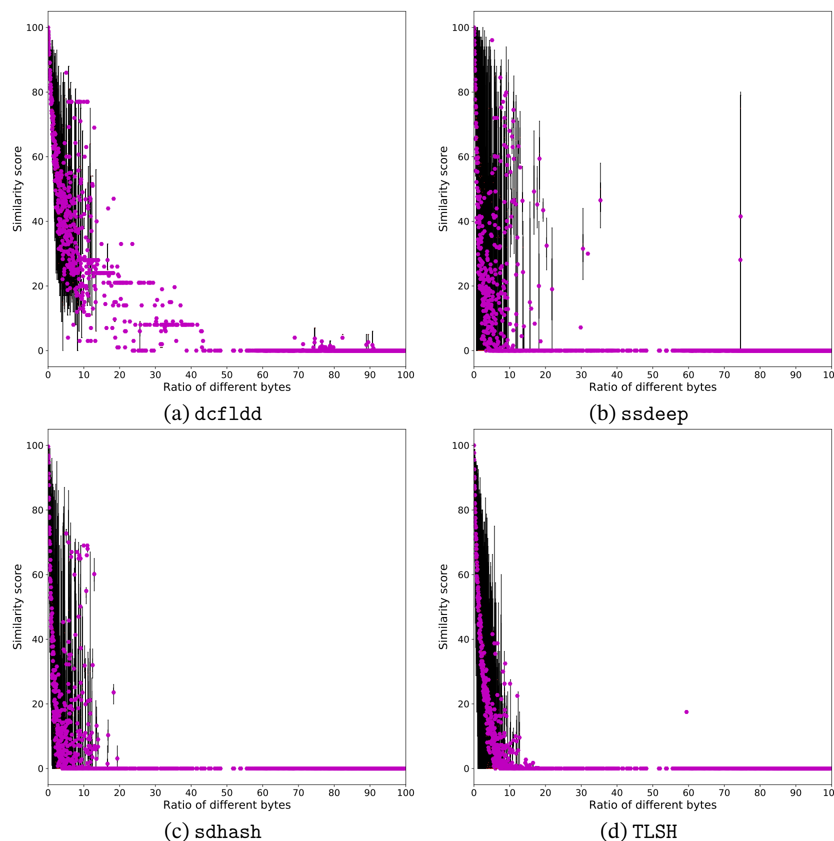
(b) 8GiB of physical memory

Figure 4.2: Resident pages of recoverable shared dynamic library modules at the first (first and third row) and at the second observation moments (second and fourth), with memory workloads of 75%, 100%, and 125% (first, second, and third column, respectively).

similarity score and the number of different bytes between pairs of pages. In particular, we have considered 868,673 comparisons over 44,398 valid pages of common modules extracted from 10 dumps (per machine) of virtual machines running three 32-bit Windows operating systems (specifically, Windows 7 6.1.7601, Windows 8.1 6.3.9600, and Windows 10 10.0.14393).

Figure 4.3 shows boxplots of the similarity scores for each algorithm with respect to the ratio of different bytes. The mean values of the boxplots are plotted with a dot. Our results indicate that more than 46% of the pages compared contain different bytes, and the similarity score of all the algorithms drops rapidly when the proportion of different bytes is between 1% and 10%. Note that the similarity score of `TLSH` is normalized to be comparable to the other scores.

Based on these results, we consider that **the bytewise similarity digest algorithms studied do not provide a good degree of confidence for a forensic analysis due to the similarity scores drop when the number of different bytes grows slightly**, as well as due to the high variability of the similarity score. As we have explained above, these byte differences in the images that come of the same image file are due to the relocation of the binary program.



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

Figure 4.3: Similarity scores with respect to the ratio of dissimilar bytes. The similarity score of `TLSH` is normalized to be comparable with the other scores.

Effect of Byte Changes on Similarity Score

As a final experiment, we evaluate the extent to which the similarity score of each similarity digest algorithm is affected when an arbitrary number of bytes is changed. In particular, we use 102214 pairs of resident pages retrieved from 8 typical modules on different executions. We calculate the similarity score for each pair with different similarity algorithms (specifically, `dcfldd`, `ssdeep`, `sdhash`, and `TLSH`). Next, we group the pages according to the similarity score. Finally, we look for the pair of pages with the fewest different bytes in each value of the similarity score. The results are shown in Figure 4.4.

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 more sensitive to byte changes while `TLSH` is the algorithm less sensitive.

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 stepwise function, mainly caused by the granularity of the algorithm: `ssdeep` generates 64 features as a maximum, and thus the number of bytes is 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 bytes affected by the relocation process causes this necessary condition to 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 pages, as `sdhash` does not select features with low entropy, as well as the number of selected features in the digest is 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 less than 16 features and thus they are incomparable, yielding to a similarity score of zero value [Roussev and Quates, 2013]. `TLSH` exhibits the most stable behavior, although it requires the fewest different bytes to provide a similarity score of zero value.

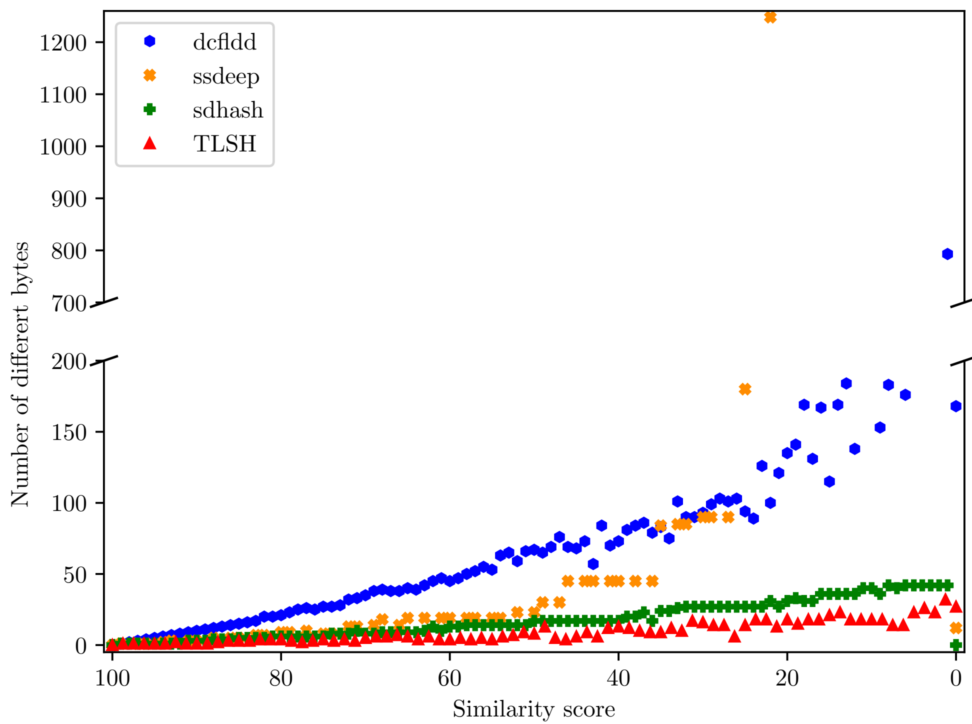


Figure 4.4: Minimum number of different bytes in a small page (4096 bytes) that drops similarity score.

Part II

Similarity Digest Algorithms Classification and Attacks

Chapter 5

Classification of Similarity Digest Algorithms

This chapter summarizes the main contribution of the dissertation related to the study of Similarity Digest Algorithms (SDA). The study proposes a novel classification about Similarity Digest Algorithms and new categories based on the approach of the state-of-the-art algorithms [Martín-Pérez et al., 2021b]. Finally, we apply the proposed classification to the current state-of-the-art SDA, as a way to validate it.

5.1 Similarity Digest Algorithms

Approximate matching algorithms identify similarities between two or more artifacts at three different levels of abstraction: *bytewise*, when the comparison is based on the raw sequence of bytes that make up digital artifacts; *syntactic*, when the internal structures of the digital artifacts under analysis are used instead of simply byte sequences; or *semantic*, when the comparison is based on contextual attributes to interpret digital artifacts and estimate their similarity.

In addition, these algorithms can directly compare artifacts (e.g., Levenshtein distance or Hamming distance), or they can first convert them to an intermediate representation (e.g., a fingerprint, hash, digest) which can then be compared. This latter case is often referred to as ***Similarity Digest Algorithms***. Other authors also call them fuzzy hashing or similarity hashing. The goal of these algorithms is to complement cryptographic hash functions by allowing the identification of *similar* objects rather than completely identical objects.

Similarity Digest Algorithms (SDA) transform digital artifacts into an intermediate representation to allow efficient (fast) identification of similar objects. These algorithms select features (sometimes also called *chunks* in literature), compress them, and merge them to form a final signature (*similarity digest*) that can be compared to another and produce a *similarity score*.

According to NIST SP 800-168, "*approximate matching is a promising technology designed to*

identify similarities between two digital artifacts" [Breitinger et al., 2014b]. SDA is a subtype of these algorithms characterized by comparing artifacts through the comparison of their digests. SDA gained a lot of popularity over the last decade with new algorithms developed and released to the digital forensic community.

SDA may be a feasible solution to do a initial triage of the artifacts of a memory dump to identify as much as possible their similarity among other artifacts. In the following sections we present a summary of the characteristics of these algorithms together with their strengths and weaknesses. The goal is to understand which algorithms have the most suitable feature set for memory forensics.

On the other hand, when algorithms are published (e.g., as part of a scientific publication), they are often compared with other algorithms to describe the benefits (and sometimes also the weaknesses) of the proposed approach. However, given the wide variety of algorithms and approaches, it is impossible to provide direct comparisons with all existing algorithms. To solve this, we present the first SDA classification that allows algorithm comparisons and an easier description. Therefore, we first review the existing literature to understand the techniques used by various algorithms and to become familiar with common terminology. Our findings allow us to develop a categorization that is largely based on the terminology proposed by NIST SP 800-168. We believe that this contribution helps newcomers, practitioners, and experts to better compare algorithms, understand their potential, as well as the characteristics and implications they may have for forensic investigations.

5.2 Terminology, methodology, and background

This section introduces the terminology related to SDA that we use in this dissertation. We also describe the methodology applied to find the literature and algorithms and to derive the classification scheme. Finally, we explain the Bloom filter, a data structure used by several algorithms to store features.

5.2.1 Terminology

Due to the novelty of this area, there is no consensus on terminology. Although all the works use a terminology that is consistent in itself, it is easy to find several articles that name the same concept with different terms or use the same term for different concepts. For example, *hash* is used in most articles to denote different concepts (e.g, hash as a digest or hash as a function). This section introduces the terminology and concepts that we follow in this and subsequent chapters, which is similar to NIST SP 800-168 [Breitinger et al., 2014b]:

Features are the basic characteristics that can be extracted from digital artifacts and allow the comparison between two or more objects. Sample features can be a single bit, byte sequences, or offsets in an object.

Mapping functions allow the processing of features, (e.g., to compress them or encrypt them using cryptographic hashing). Note that the NIST, as well as other literature, frequently use the term *compression functions* as this is the most common behavior of this function. However, we think that mapping function is more accurate as theoretically the function can also expand the feature. We refer to the features obtained as a result of the mapping functions as *processed features*.

Similarity digests are the final output of SDA and can be seen as an aggregation of processed features.

Similarity functions allow the comparison of two similarity digests, returning a *similarity score* that is often a numeric value. This value, although it often ranges from 0 to 100, is not necessarily a percentage value.

5.2.2 Methodology

To derive the classification scheme, we review the relevant literature, i.e., descriptions of the various algorithms, as well as secondary literature such as comparisons of algorithms, security evaluations, or suggested properties for approximate matching (e.g., as suggested by [Breitinger and Baier, 2012a]). Our starting point are articles that discuss properties and security features of algorithms. Thus, we start with more general/broader articles followed by articles describing specific algorithms and implementations. For each article, we extract relevant information (characteristics) describing behavior, features, and peculiarities of algorithms. Last, we try to align these characteristics as best as possible.

Table 5.1 shows the algorithms that we consider in this chapter, presenting their old and new classification that are discussed in Section 5.4 considering our proposed classification scheme. For each algorithm we also detail their main literature references.

5.2.3 The Bloom Filter

The Bloom filter is a probabilistic structure with a good space/time trade-off based on hash functions and used to represent sets of elements, introduced in 1970 by Bloom [Bloom, 1970]. This structure is widely used by SDA to store and compare sets of significant elements.

A *Bloom filter* (BF) is a representation of a set $S = \{s_1, s_2, ..., s_n\}$ of n elements from U. The filter consists of a vector of m bits, initially set to 0, and k independent sub-hash functions that support filtering operations. The sub-hash functions uniformly maps the elements of U to the range $[0, m-1]$. To insert an element s into the filter, the structure calculates the k sub-hash values of s and sets the k positions of m that are pointed by the k sub-hash values to 1, increasing the number n of elements within the filter by 1. To see if an element is inside a BF, it calculates the k sub-hash functions and checks the pointed positions. If all bits are 1, the element is inside the BF. However, there is a probability $P = (1-(1-1/m)^{kn})^k$ that a query

Algorithm	Reference	Previous classification	New classification
`dcfldd`	[Harbour, 2002]	Block-Based Hashing	Feature Sequence Hashing
`ssdeep`	[Kornblum, 2006]	Context Trigger Piecewise Hashing	Feature Sequence Hashing
`md5bloom`	[Roussev et al., 2006]	Block-Based Hashing	Feature Sequence Hashing
`MRS hash`	[Roussev et al., 2007]	Context Trigger Piecewise Hashing	Feature Sequence Hashing
`sdhash`	[Roussev, 2010]	Statistically-Improbable Features	Feature Sequence Hashing
`MRSH-v2`	[Breitinger and Baier, 2012b]	Context Trigger Piecewise Hashing	Feature Sequence Hashing
`SimHash`	[Sadowski and Levin, 2007]	Block-Based Rebuilding	Byte Sequence Existence
`mvHash-B`	[Breitinger et al., 2013]	Block-Based Rebuilding	Byte Sequence Existence
`LZJD`	[Raff and Nicholas, 2017]	(none)	Byte Sequence Existence
`Nilsimsa`	[Damiani et al., 2004]	Locality-Sensitive Hashing	Locality-Sensitive Hashing
`TLSH`	[Oliver et al., 2013]	Locality-Sensitive Hashing	Locality-Sensitive Hashing
`saHash`	[Breitinger et al., 2014d]	(none)	Locality-Sensitive Hashing
`FbHash`	[Chang et al., 2019]	(none)	Locality-Sensitive Hashing

Table 5.1: Proposed classification for state-of-the-art similarity digest algorithms; the last two columns are discussed in Section 5.4.

will receive a false positive because the marked bits are 1 due to other elements. In contrast, the probability of a false negative is 0.

To address a position in an m-bit vector, $log_2 m$ bits are needed. So, to add a new element it is necessary to generate $klog_2 m$ independent bits. Taking advantage of this fact, SDA implementing BF calculate only one hash with enough output bits and divided it into k, using them as sub-hash values. Since only one hash is computed, SDA performance is improved by reducing computation time.

At the same time, SDA limit the number of elements inside filters to keep the probability of false positive low, concatenating a new filter when the previous one reaches a fixed threshold.

5.3 Proposed Classification Scheme

To develop the classification scheme, we reviewed the most widely used algorithms and implementations, as well as some niche ones, summarized in Table 5.1. First, we describe the general phases of SDA, and then, we detail the procedures/dimensions and characteristics of each phase.

5.3.1 Phases of a Similarity Digest Algorithm

Similar to traditional hash functions, SDA have two main stages of work:

1. During the **artifact processing and digest generation phases**, an algorithm receives data (i.e., a sequence of bytes) as input, processes the input, and returns a similarity digest. In detail, there are several phases: First, the features are extracted from the input (*feature generation phase*), which are then processed (*feature processing*). Some al-

gorithms may have an optional phase to select the features that will make up the digest (*feature selection phase*). If this phase does not exist, all the features form the digest. In addition, some algorithms may have an optional phase to remove duplicate features (*features deduplication phase*). This phase can be associated with the next stage when it uses digests instead of features as input. Finally, the processed features are transformed to form the similarity digest (*digest generation phase*), which is the final result of the SDA. Note that the order of phases is not fixed, as each algorithm can implement the phases in a different order. For instance, one algorithm may first select features and then perform processing, while other algorithms may implement the generation phase in conjunction with the feature deduplication phase due to the underlying *storage structure* used.

2. The **digest comparison phase** is a separate phase that requires two similarity digests as input (obtained from the same SDA). Using some similarity metric, the digests are compared and a similarity score is returned. This score provides information about the similarity (or dissimilarity) between the digests.

These phases led to our classification scheme as summarized in Table 5.2 where we identified several dimensions/procedures for each phase, each with various characteristics.

5.3.2 Feature Generation Phase

To begin, SDA process the input at the byte level with the goal of identifying and extracting features. We distinguish between the following dimensions/procedures: The ***length*** of the feature can be a fixed length (*static*) or variable (*dynamic*). In the latter case, the features are generally identified with a ***splitting function*** that allows to identify the limits of the features. Currently, we differentiate between two different splitting functions. A *trigger* function compares its output to a predefined value to determine the feature boundaries. On the other hand, a *unique* function can be used to construct a set of unique features from the input using set theory[2]. When the feature size is static, an SDA may not need a splitting function to identify boundaries. These cases are indicated with *none*. An example of static features would be `dcfldd`, which uses 512 bytes. In contrast, `ssdeep` and `MRSH-v2` use trigger functions (rolling hashes) to identify features, resulting in features of variable size.

The ***intersection*** dimension in the feature generation phase can be *yes* (a byte can belong to more than one feature) or *no* (a byte can belong to exactly one feature). Some examples are `ssdeep` and `sdhash`, which have distinct and overlapping features, respectively.

The fourth dimension is ***cardinality***, which describes the number of features that SDA try to produce for a given input which can be *fixed* or *variable* where it often depends on the input length L. For instance, `ssdeep` has a maximum of 64 features (*fixed*) while `dcfldd` and `MRSH-v2` produce $L/512$ and $L/320$, respectively (*variable*). In other words, cardinality is strongly related to the length dimension of the feature.

[2] One may consider this as part of the feature selection phase. However, in this particular case the feature changes if it is in the set. Therefore, we placed it in feature generation.

	Phase	Dimension/ Procedure	Characteristic
Artifact processing and digest generation phases	FEATURE GENERATION	Length	*Static; Dynamic*
		Splitting function	*Trigger; Unique; None*
		Intersection	*Yes; No*
		Cardinality	*Fixed; Variable*
	FEATURE PROCESSING	Mapping function	*Hash; Encoding; Identifier; None*
		Bit reduction	*Ratio; None*
	FEATURE SELECTION	Selection function	*Minimum probability; Block matching; Block similarity; Minimum value; None*
		Domain	*Feature; Processed feature*
		Coverage	*Complete; Partial*
	DIGEST GENERATION	Digest size	*Fixed; Input dependent; Input dependent with max*
		Storage structure	*Processed feature concatenation; Set concatenation; Set; Counter*
		Order	*Absolute; Set-absolute; Processed feature–aware; None*
		Generation requirements	*Minimum features; Diversity; Document frequency; None*
	FEATURE DEDUPLICATION	Type	*Consecutive; In-scope; None*
		Occurrence phase	*Digest generation; Digest comparison*
	DIGEST COMPARISON	Method	*Common feature ratio*; *Weighted edit distance*; *Average Hamming distance*; *Jaccard index*; *Cosine similarity*
		Comparison requirements	*Minimum commonality; Minimum cardinality; Similar cardinality; Ordered cardinality; Equal feature parameter; None*
		Output score	*Binary value; Interval; Half-bounded*
		Score trend	*Ascending; Descending*
		Spatial sensitivity	*Partial; Total; None*

Table 5.2: Categorization scheme for similarity digests algorithms. The possible values for each dimension/procedure are separated by semicolons.

5.3.3 Feature Processing Phase

After identifying the features, there may be a processing phase which includes two dimensions. A ***mapping function*** (or compression function, as suggested by NIST) applies some form of mapping to the feature which can be hash, encoding, or identifier. We only found one algorithm, `saHash`, which does not apply any compression function since its feature size is already small. Most commonly, SDA use a *hash* to obtain pseudo-random processed features where cryptographic and non-cryptographic algorithms are used. For instance, `MRSH-v2` uses FNV-1a function [Fowler et al., 2011] to reduce features from 320 bytes to 64 bits, while `sdhash` relies on SHA-1 as mapping function. Instead of hash, the mapping function can be a more general *encoding* function to process the features. For instance, `mvHash-B` applies a majority voting step followed by Run Length Encoding to compress the sequences of the chosen identifiers. The

last approach we found is a feature *identifier* function (which works like a mapped feature) and generates feature identifiers of selected features from a fixed set. For instance, SimHash generates an identifier for each feature that matches one of its 16 fixed features.

The mapping function is usually followed by a ***bit reduction*** procedure that consists of selecting a few bytes (or bits) of the mapped feature. This bit reduction can be expressed as a *ratio* between output and input. For instance, the MRSH-v2 input to the bit reduction function is a 64-bit FNV-1a hash that results in 55-bit output. Hence, the ratio is 55/64. Similarly, the sdhash inputs are 160-bit SHA-1 hashes, while the outputs are 55 bits, resulting in a 55/160 ratio. We have used *none* when the SDA does not have a bit reduction procedure (that is, all the bits of the mapped feature are used). For completeness in Table 5.3, in these cases we have written the output size (in bits) in parenthesis.

5.3.4 Feature Selection Phase

This phase, carried out by some algorithms, selects specific features (e.g., the most exclusive) to build a digest. Its implementation can be distributed over several functions of different phases. As a theoretical example, an algorithm can implement a feature generation phase that does not produce features for long chunks of zeros. Then it can select some of the generated features for the digest. Hence, this phase should be considered as a transversal phase.

The ***selection function*** characterizes the main approach of this phase, which can be based on: *minimum probability* (i.e., the most unique), when the features with the least probability of occurring are selected since the most significant features are considered (e.g., sdhash); *block matching*, when a predefined set of blocks is used to find and replace the sequence of features with a sequence of block identifiers (e.g., SimHash); *block similarity*, when the most similar blocks are identified (e.g., mvHash-B); *minimum values*, when the algorithm selects a subset of processed features with the lowest values (e.g., LZJD).

The ***domain*** of these selection functions is *feature* or *processed feature*. For instance, sdhash selects features by their entropy and then calculates the processed features, which are finally added to the digest. On the contrary, LZJD first calculates the processed features and then selects which of them will form the digest.

Coverage reflects whether all bytes of the input are considered in the similarity digest. It can be *complete* (such as ssdeep or MRSH-v2) or *partial* (such as sdhash, where there may be gaps between features). In the latter case, it is important to understand that two inputs can produce a perfect SDA match but they may not be identical (i.e., they have different cryptographic hash values).

5.3.5 Digest Generation Phase

This phase includes the construction of a similarity digest. The ***digest size*** can be *fixed* when it is always the same length, regardless of the input length (e.g., FbHash or saHash); *input dependent*, when there is a correlation between input size and digest size (e.g., dcfldd concatenates

the MD5 hash for each 512-bytes feature); or *input dependent with max*, when size depends on input length but has a maximum length (e.g., `ssdeep` produces a digest between 32 and 64 characters, but may not reach its maximum for small inputs).

Regarding the ***storage structure***, it can be a *processed feature concatenation* (i.e., the processed features are simply concatenated, as `dcfldd` or `ssdeep` do) or it can be *set concatenation*, which describes the idea of adding processed features to a set with a maximum capacity. When the set reaches its capacity (that is, it is full), a new empty set is added. So far, all implementations of the set concatenation method use Bloom filters and the comparison process described by [Roussev et al., 2006; Roussev, 2010, 2012]. When the number of processed features is limited, they can be stored in a single *set* as in the case of `LZJD`. Finally, during the selection phase, some algorithms (e.g., `Nilsimsa` and `SimHash`) add features according to *counters*.

The storing ***order*** between features is also a characteristic of SDA. Possible values are: *absolute*, when the position of each feature is preserved (for instance, `dcfldd` or `ssdeep` store all features in order); *set-absolute*, when the generated digest maintains the order between different sets of features, but does not know the order of features within a particular set. For example, SDA that use set concatenation (based on Bloom filters) for feature storing have this characteristic value, as the Bloom Filters do not maintain the order (e.g., `md5bloom`); *processed feature–aware*, when the storage structure does not maintain the order of the features contained within, but the processed feature itself helps determine the order. Thus, the intersection between features allows to detect changes (i.e., the similarity score varies) when the input is rearranged (e.g., `TLSH` and `saHash`); or *none*, when the SDA does not consider the order between features (e.g., `SimHash` and `LZJD`).

Some algorithms have ***generation requirements*** to produce a digest. The requirements can be to achieve *minimum features*, to have sufficient *diversity* in the input, or to have a *document frequency* that contains the frequency of each feature in a training set. When SDA has no requirements, we use *none*. In this regard, some algorithms have a feedback phase, during which they make configuration adjustments and restart the process from the beginning when the generated digests do not meet the minimum expected requirements.

5.3.6 Features Deduplication Phase

The *features deduplication phase* is an optional phase implemented by some algorithms to eliminate duplicate or redundant processed features. The ***type*** can be *consecutive*, when several consecutive features are reduced to a short sequence; or *in-scope*, when identical features in the same scope are removed. When this phase does not exist, the type is *none*. This phase can take place (***occurrence phase***) at two different sites, either during the *digest generation* or during the *digest comparison* phases. For example, algorithms that use Bloom Filters deduplicate by design during the generation phase, since these filters behave as sets and thus do not store duplicate features.

5.3.7 Digest Comparison Phase

This phase allows two digests to be compared, resulting in a similarity score, i.e., how similar or different these two digests are. There are several **methods** to calculate the similarity between two digests: *common feature ratio*, which calculates the number of common elements divided by the maximum digest size; *weighted edit distance* (Damerau–Levenshtein distance), which is the edit distance between two sequences of features [Levenshtein, 1966; Damerau, 1964]; *average Hamming distance*, when the digests are composed of storage substructures and the similarity is measured by adding the Hamming distance between substructures [Hamming, 1950]); *Jaccard similarity* (similarity coefficient), which calculates the common elements between sets divided by the union of elements in both sets [Jaccard, 1908]; or *cosine similarity*, which is the sum of multiplying the normalized frequency of each pair of terms with each other [Salton and Buckley, 1988].

To return a similarity (or dissimilarity) score, some algorithms impose one or more ***comparison requirements***. When these requirements are not met, the phase can return a non-similarity value or a comparison error. Comparison of digests can require a *minimum commonality* among digests, a *minimum cardinality* of features in each digest, a *similar cardinality* of both inputs, *ordered cardinality* (where the cardinality of the first digest has to be greater/less than second one; usually ordered by the implementation), or *equal feature parameter* (when the feature generation process has some parameter that can dynamically change but this parameter needs to match between compared digests; e.g. trigger value or feature size).

The ***output score*** is the similarity score between two given digests and can be a *binary* value, i.e., inputs are similar or not (yes/no), an *interval* (usually between $[0, 1]$ or $[0, 100]$), or *half-bounded*. The latter case indicates that there is only one lower (or upper) limit. This is common when dissimilarity is measured such that 0 is (almost) identical, but there is no upper bound. Related to the output score is the ***score trend***, which can be *ascending* (the higher the output score, the greater the similarity of the digests) or *descending*, otherwise.

Another characteristic of the comparison function is its ***spatial sensitivity***, which expresses whether the function is sensitive to a different order of the same features. This sensitivity can be *total*, when the total order of the features is considered; or *partial*, when the order of features is only partially considered.

5.4 Classification of state-of-the-art SDA

Tables 5.3, 5.4 and 5.5 classify the state-of-the-art SDA that we have considered in this dissertation in chronological order. In total, we reviewed 13 algorithms released between 2002 and 2019. For readability, we have divided the results into three tables: Table 5.3 contains the *feature generation* and *feature processing* phases; Table 5.4 details *feature selection* and *digest generation* phases; and Table 5.5 describes *feature deduplication* and *digest comparison* phases.

Before discussing the existing algorithms regarding our classification, let us take a look at the existing classifications that are mainly based on the categories that the creators assigned

Algorithm	Feature generation				Feature Processing	
	Length	*Splitting Function*	*Intersection*	*Cardinality*	*Mapping Function*	*Bit Reduction*
`dcfldd`	Static (512)	None	No	Variable ($L/512$)	Hash	None (128)
`Nilsimsa`	Static (3)	None	Yes	Variable ($6L$)	Hash	None (8)
`ssdeep`	Dynamic ($L/64$)	Trigger	No	Fixed (64)	Hash	Ratio (6/32)
`md5bloom`	Static (512)	None	No	Variable ($L/512$)	Hash	Ratio (40/128)
`MRS hash`	Dynamic (256)	Trigger	No	Variable ($L/234$)	Hash	Ratio (44/128)
`SimHash`	Static (1)	None	Yes	Variable ($8L$)	Identifier	None (8)
`sdhash`	Static (64)	None	Yes	Variable (L)	Hash	Ratio (55/160)
`MRSH-v2`	Dynamic (320)	Trigger	No	Variable ($L/320$)	Hash	Ratio (55/64)
`mvHash-B`	Static (21, 51)	None	Yes	Variable (L)	Encoding	Ratio (1/32)
`TLSH`	Static (3)	None	Yes	Variable ($6L$)	Hash	None (8)
`saHash`	Static (1)	None	Yes	Variable ($2L$)	None	None (8)
`LZJD`	Dynamic ($1 + \log_{256} L$)	Unique	No	Variable ($L/(1 + \log_{256} L)$)	Hash	None (128)
`FbHash`	Static (7)	None	Yes	Variable (L)	Hash	None (64)

Table 5.3: Classification of similarity digest algorithms according to our proposed classification scheme (*feature generation* and *feature processing* phases).

Algorithm	Feature Selection			Digest generation			
	Selection Function	*Domain*	*Coverage*	*Digest Size*	*Storage Structure*	*Order*	*Generation requirements*
`dcfldd`	None	(n/a)	Complete	Input dependent	Processed feature concatenation	Absolute	None
`Nilsimsa`	None	(n/a)	Complete	Fixed	Counter	Processed feature–aware	None
`ssdeep`	None	(n/a)	Complete	Input dependent with max	Processed feature concatenation	Absolute	Minimum features
`md5bloom`	None	(n/a)	Complete	Input dependent	Set concatenation	Set-absolute	None
`MRS hash`	None	(n/a)	Complete	Input dependent	Set concatenation	Set-absolute	None
`SimHash`	Block matching	Feature	Partial	Fixed	Counter	None	None
`sdhash`	Minimum probability	Feature	Partial	Input dependent	Set concatenation	Set-absolute	Diversity
`MRSH-v2`	None	(n/a)	Complete	Input dependent	Set concatenation	Set-absolute	None
`mvHash-B`	Block similarity	Feature	Complete	Input dependent	Set concatenation	Set-absolute	Diversity
`TLSH`	None	(n/a)	Complete	Fixed	Counter	Processed feature–aware	Diversity
`saHash`	None	(n/a)	Complete	Fixed	Counter	Processed feature–aware	None
`LZJD`	Minimum value	Processed feature	Partial	Fixed	Set	None	None
`FbHash`	None	(n/a)	Complete	Fixed	Counter	Processed feature–aware	Document frequency

Table 5.4: Classification of similarity digest algorithms according to our proposed classification scheme (*feature selection* and *digest generation* phases).

Algorithm	Feature Deduplication		Digest comparison				
	Type	*Occurrence*	*Method*	*Comparison requirements*	*Output Score*	*Score Trend*	*Spatial Sensitivity*
`dcfldd`	None	(n/a)	Common feature ratio	None	Interval	Ascending	Total
`Nilsimsa`	None	(n/a)	Weighted edit distance	None	Interval	Ascending	Partial
`ssdeep`	Consecutive	Comparison	Weighted edit distance	Minimum commonality, Equal feature parameter	Interval	Ascending	Total
`md5bloom`	In-scope	Generation	Average Hamming distance	None	Interval	Ascending	Partial
`MRS hash`	In-scope	Generation	Average Hamming distance	None	Interval	Ascending	Partial
`SimHash`	None	(n/a)	Weighted edit distance	Similar cardinality	Half-bounded	Descending	None
`sdhash`	In-scope	Generation	Average Hamming distance	Ordered cardinality, Minimum cardinality, Minimum commonality	Interval	Ascending	Partial
`MRSH-v2`	In-scope	Generation	Average Hamming distance	Ordered cardinality, Minimum cardinality, Minimum commonality	Interval	Ascending	Partial
`mvHash-B`	In-scope	Generation	Average Hamming distance	Similar cardinality	Interval	Ascending	Partial
`TLSH`	None	(n/a)	Weighted edit distance	None	Half-bounded	Descending	None
`saHash`	None	(n/a)	Weighted edit distance	None	Binary	(n/a)	Total
`LZJD`	None	(n/a)	Jaccard similarity	None	Interval	Ascending	None
`FbHash`	None	(n/a)	Cosine similarity	None	Interval	Ascending	None

Table 5.5: Classification of similarity digest algorithms according to our proposed classification scheme (*feature deduplication* and *digest comparison* phases).

to their algorithms. Based on [Gayoso Martínez et al., 2014; Lee and Atkison, 2017; Moia and Henriques, 2017a], there are the following categories:

Block-Hased Hashing, which consists of algorithms that use cryptographic hashes, generating and storing features for each block of a fixed size;

Context Trigger Piecewise Hashing, which is made up of algorithms that divide the input into contexts, defined as a sliding window on the input bytes when the trigger function fires;

Statistically-Improbable Features, comprising algorithms that use a selection function based on statistically improbable features, as its name implies;

Block-Based Rebuilding, which consists of algorithms that choose blocks (randomly selected or preset) and generate the digests by selecting the blocks most similar to the input; and

Locality-Sensitive Hashing, which is made up of algorithms that map objects in buckets, grouping similar objects in the same bucket with high probability.

However, these categories only consider specific aspects of the algorithms, rather than a complete view of the complete behavior of the algorithms. This leads to a misunderstanding of how an algorithm works, which can eventually lead to wrong decisions when selecting an SDA for a specific purpose (for example, comparing algorithms with totally different behavior or even not comparing them with other similar algorithms). Therefore, we propose a new and simpler classification based on the complete behavior as follows:

Feature Sequence Hashing, which encompasses the algorithms that divide the input into features and map them, measuring similarity by feature sequences.

Byte Sequence Existence, this category comprises the algorithms that identify the existence (or similarity) of byte sequences (called *blocks*) in the input. The similarity score is calculated by comparing the number of common blocks between similarity digests.

Locality-Sensitive Hashing, this category is as in the previous classification. It is made up of algorithms that map objects in buckets, grouping similar objects in the same bucket with high probability.

5.5 Applicability of the Proposed Scheme

Finally, we validate our proposed classification scheme by applying it to all relevant algorithms (presented in chronological order) and discussing the new classification with respect to the previous one:

5.5.1 dcfldd

dcfldd is an improved version of the GNU dd Unix program [Harbour, 2002]. Developed by Nicholas Harbour in 2002, it ensures copy integrity by ensuring the integrity of the blocks. It has been used to measure the similarity between artifacts by counting how many blocks are equal. Next, we describe dcfldd using our classification scheme.

dcfldd generates features (FEATURE GENERATION) by dividing the input into disjoint 512-byte blocks. This means that the **splitting function** is *none* because the features have a *static* **length** with *no* **intersection**. **Cardinality** is *variable* because it depends on the input length ($L/512$).

In the next phase, FEATURE PROCESSING, dcfldd calculates (**mapping function**) a cryptographic *hash* for each feature, allowing different algorithms to be used: MD5, SHA1, SHA2. All processed features are considered in the digest, so there is no FEATURE SELECTION phase. Therefore, the **selection function** is *none* and **coverage** is *complete*.

In the DIGEST GENERATION phase, the **storage structure** is a *processed features concatenation* where the cryptographic hashes are concatenated in **order** of occurrence (*absolute*) into a string, separating the processed features by a colon. The **digest size** is *input dependent*. This phase does not have **generation requirements** (*none*). For this algorithm, there is no FEATURE DEDUPLICATION phase (**type**: *none*).

For the DIGEST COMPARISON phase, dcfldd uses the *common features ratio* **method**, which considers the number of equal features in the same position within the digest to calculate similarity. For this comparison approach, the **spatial sensitivity** is *total*, the **output score** is within an *interval* with an *ascending* **score trend**, and the **comparison requirements** are *none*.

dcfldd is considered as a *Block-Base Hashing* because it divides the data into fixed-size blocks and hashes them [Gayoso Martínez et al., 2014; Moia and Henriques, 2017a]. According to our proposed classification, it is a *Feature Sequence Hashing* as it measures similarity by feature sequences.

5.5.2 Nilsimsa

Nilsimsa is an anti-spam algorithm originally proposed in 2001 and reviewed by Ernesto Damiani et al. in 2004 [Damiani et al., 2004]. The most significant differences between both works are in the generation digest phase, as explained below, and on the threshold to identify the same message, which is beyond the scope of this dissertation.

In the FEATURE GENERATION phase, Nilsimsa generates all possible trigrams (3 bytes) within a 5-bytes sliding window as features. Therefore, the features have a *static* **length** so the algorithm **splitting function** is *none*. There is **intersection** (*yes*) between features. **Cardinality** is *variable* ($6L$), depending on input size.

In the next phase, FEATURE PROCESSING, the trigrams are processed by a *hash* function (**mapping function**) to obtain a byte identifier that is used as the processed feature. Hence,

the **bit reduction** is *none*.

This algorithm does not rule out features. So in the FEATURE SELECTION phase, the **selection function** is *none* and the **coverage** is *complete*.

To build the digest, in the DIGEST GENERATION phase, `Nilsimsa` adds the processed features in 256 *counters* (**storage structure**). In the original implementation of `Nilsimsa`, it yields a value of 1 for each counter whose cardinality is above the mean of all, otherwise it yields a value of 0. In contrast, in the revised version the relative frequency of each counter is compared with the average counter frequency of a large collection. Then, a value of 1 bit is obtained for each counter if the ratio of the counter is greater than the average of the counter ratios. In both cases, the digest is a sequence of 32 values of 1 and 0, each of which represents a counter. Therefore, the **digest size** is *fixed* and **generation requirements** to build a digest are *none*. Regarding the **order**, the sequence of features generated from the sliding window allows to identify if two arbitrary byte sequences are swapped between similar inputs, implementing a *processed feature–aware*. Since this algorithm creates the digest based on the frequency of the features, it cannot do any **type** (*none*) of FEATURE DEDUPLICATION.

To compare two digests, DIGEST COMPARISON phase, `Nilsimsa` implements a *weighted edit distance* as a comparison **method** where it counts how many bits are equal in the same positions minus 128—this value is because each bit represents one counter and the average of equal bits between a pair of random inputs is 128. The **output score** is an *interval* with an *ascending* **score trend**. The **comparison requirements** are *none* and **spatial sensitivity** is *partial*, since a random swap of byte sequences can cause a variation in the sequence of generated features due to the overlap between features.

Our classification is the same as that given by the authors in [Damiani et al., 2004], *Locality-Sensitive Hashing*.

5.5.3 `ssdeep`

`ssdeep` was developed in 2006 by Jesse Kornblum [Kornblum, 2006]. This is the first algorithm that proposed to dynamically split the input based on the content of the input itself.

In the FEATURE GENERATION phase, `ssdeep` uses Adler32 checksum function [Deutsch and Gailly, 1996] (*trigger*) as a **splitting function** to divide the input into approximately 64 distinct features (*dynamic* **length**, *no* **intersection**, and *fixed* **cardinality**). The rolling hash function is configured with an initial block size $b = 3 \cdot 2^{\lfloor \log_2(\frac{n}{64 \cdot 3}) \rfloor}$, where n is input size. In addition, the remaining input after the 63rd feature is considered the last feature.

In the next phase, FEATURE PROCESSING, features are processed and *hashed* with the FNV algorithm [Fowler et al., 2011] (**mapping function**). The least relevant 6 bits of the 32-bit FNV output are selected (*ratio* **bit reduction**) to generate a base64-encoded character.

With regard to the FEATURES SELECTION phase, all the processed features belong to the digest (*complete* **coverage**), so the feature selection phase is skipped (**selection function** is *none*).

The DIGEST GENERATION phase creates a digest that is a *processed feature concatenation* (**storage structure**), maintaining the *absolute* **order**. This algorithm has a **generation requirement** that is *minimum features*: if the generated features are less than 32, `ssdeep` does not produce a digest and restarts the entire process, dynamically readjusting the block size to produce twice as many features ($b = b/2$). On the other hand, the **digest size** is *input dependent with max* because the generation phase produces a maximum of 64 features. As we explain below, the comparison process requires digests obtained from the same trigger value b. Therefore, the `ssdeep` digest contains two feature sequences for block size b and $2b$ to maximize the likelihood of comparison.

`ssdeep` implements the FEATURE DEDUPLICATION phase within the *comparison* phase (**occurrence**). It reduces the sequences of *consecutive* (**type**) repeated features to three. The DIGEST COMPARISON phase assesses the necessary **comparison requirements**: *i*) *equal feature parameter*: the same block size is required to compare two sequences of features; *ii*) *minimum commonality*: there are at least 7 consecutive common features among the digests for comparison. Finally, a *weighted edit distance* **method** between features sequences is used to calculate the similarity score, which is in an *interval* $[0, 100]$ (**output score**) where 0 indicates no similarity while 100 indicates identical inputs. That is, it has an *ascending* **score trend**. Feature concatenation and edit weight distance provide *total* **spatial sensitivity**.

`ssdeep` is classified as *Context Trigger Piecewise Hashing* due to the novel method used in the feature generation phase. However, under our consideration, a classification as *Feature Sequence Hashing* is more general and accurate.

5.5.4 `md5bloom`

In 2006, Vassil Roussev et al. explored the use of Bloom filters [Bloom, 1970] to add and search for hash information. They implemented their proposal in `md5bloom`, defined as "*a Bloom filter manipulation tool that can be incorporated into forensics practice*" [Roussev et al., 2006]. The tool requires hashes as input and the authors propose a method to compare sets of hashes, so the tool can be considered as a SDA.

In the FEATURE GENERATION phase, `md5bloom` splits inputs into disjoint features (*no* **intersection**) of 512 bytes (*static* **length**). The **splitting function** is *none* and the **cardinality** is *variable* ($L/512$).

In the next phase, FEATURE PROCESSING, the **mapping function** is the MD5 *hash* algorithm. Regarding **bit reduction**, the algorithm only considers 40 bits of the 128 bits of MD5 output (*ratio*). On the other hand, the FEATURE SELECTION phase has *complete* **coverage** because the **selection function** is *none*.

Then, in the DIGEST GENERATION phase, the processed features are added to a Bloom filter until the maximum capacity is reached, at which point an empty Bloom filter is added to the end of the digest. Therefore, a digest is a sequence of Bloom filters whose **digest size** is *input dependent* and the **storage structure** is a *set concatenation*, while the **order** between features

contained in different sets is preserved by concatenating the Bloom filters (*set-absolute*). This phase has no **generation requirements** (*none*).

The FEATURE DEDUPLICATION phase is integrated with the digest *generation* phase (**occurrence**) because the Bloom filters discard duplicated features that belong to the same filter (*in-scope* **type**).

As we have mentioned above, `md5bloom` is the first approach that proposes the use of Bloom filters to store and compare features. Therefore, its COMPARISON DIGEST phase is a bit weak. The comparison **method** measures the number of matching bits between filters using an *average Hamming distance*. This number of common bits allows the probability of their matching by chance to be calculated. The ratio between matching bits and the number of bits set to 1 works in both filters compared as an **output score** of *interval* with an *ascending* **score trend**. The **spatial sensitivity** inherits its characteristic from the digest order, and it is *partial* as Bloom filters do not maintain the order between features, so only when the features belong to different sets, they can cause a variation in the similarity score. This algorithm has no **comparison requirements** (*none*).

`md5bloom` is considered a *Block-Base Hashing* because it divides the data into fixed-size blocks and converts them to hashes. However, we consider it a *Feature Sequence Hashing* because of the concatenation of all processed features in Bloom filters.

5.5.5 `MRS hash`

In 2007, Vassil Rousev et al. proposed *Multi-resolution similarity hashing* (`MRS hash`) [Roussev et al., 2007]. In their study, they compared the performance of various algorithms for different functionalities, e.g, splitting function or mapping function, with the results of their algorithm.

They finally implemented the following characteristics: as FEATURE GENERATION, `MRS hash` uses a *trigger* function as a **splitting function** to split the input into multiples features. Specifically, `MRS hash` uses *djb2*—a polynomial hash—in a 7-byte sliding window, which compares the generated value with a trigger value ($t = 8$), configured to produce features of approximately 256 bytes, *dynamic* **length**. To ensure that the features are not too small, a minimum feature size is entered, which is $1/4$ of the expected average byte sequence size. This means that after matching the output of the trigger function, 64 bytes of the input are added directly to the next feature. So, there is *not* **intersection** between the features and the **cardinality** is *variable* ($L/256$).

In the next phase, FEATURE PROCESSING, `MRS hash` calculates the MD5 of the features—*hash* **mapping function**— and selects 4 sets of 11 bits as hash values for the Bloom filter. Hence, the **bit reduction** *ratio* is $44/128$. Regarding the FEATURE SELECTION phase, this algorithm adds all the generated features to the digest, so the **selection function** is *none* and the **coverage** is *complete*.

In the DIGEST GENERATION phase, `MRS hash` uses a sequence of Bloom filters (*set concatenation*) as the **storage structure**. Therefore, the **digest size** is *input dependent*, the **order**

is *set-absolute*, and the **generation requirements** are *none*. The FEATURE DEDUPLICATION phase is integrated into the digest *generation* phase (**occurrence**) by the behavior of the Bloom filters, which deduplicate the repeated features *in-scope* (**type**) of a Bloom filter.

For the last phase, DIGEST COMPARISON, V. Rousev et al. proposed a different comparison function than in their previous work, `md5bloom`. In this work, the authors calculate a similarity score called Z-*score* for a pair of Bloom filters based on the number of zero bits within the filters and the number of zero bits in the inner product, which is a kind of Hamming distance. To measure similarity between two digests, they average the maximum values of comparing each Bloom filter from one digest with each filter from the other digest, a kind of *average Hamming distance* **method**. The **comparison requirements** are *none* for this comparison function, while its **output score** is an *interval* with *ascending* **score trend** and the **spatial sensitivity** is *partial*.

`MRS hash` is considered as a *Context Trigger Piecewise Hashing* algorithm by the trigger function of the feature generation phase. However, we believe that it is more accurate to consider it as *Feature Sequence Hashing* for its behavior.

5.5.6 `SimHash`

In 2007, Sadowski and Levin published `SimHash` [Sadowski and Levin, 2007], which is based on the idea of counting the occurrences of certain binary strings within a file.

In the FEATURE GENERATION phase, this algorithm considers any byte with a difference of one bit from the input. Hence, the **length** is *static* (1 byte) without **splitting function** (*none*), there is **intersection** (*yes*), and the **cardinality** is *variable* ($8L$).

The FEATURE SELECTION phase selects *features* (**domain**) by *block matching* to a set of 16 fixed blocks (**selection function**). Any selected feature cannot share bits with any other selected feature. The **coverage** of this approach is *partial*.

The FEATURE PROCESSING phase then generates an *identifier* for each selected feature (**mapping function**). The **bit reduction** of the identifier is *none*.

The DIGEST GENERATION phase adds the identifiers in *counters* (**storage structure**), creating a *fixed* **digest size** that does not consider any order between the features (*none* **order**). The **generation requirements** are *none* for this process.

Regarding the FEATURE DEDUPLICATION phase, this algorithm does not implement any type of deduplication (*none* **type**).

Finally, the DIGEST COMPARISON phase implements a *weighted Hamming distance* (**method**) between the digests with a *similar cardinality* (**comparison requirements**). The **output score** starts at 0 when the inputs are very similar or the same without a clear upper limit (*half-bounded*), so the similarity **score trend** is *descending*. The **spatial sensitivity** is *none* due to aggregation in counters.

`SimHash` is classified as a *Block-Based Rebuilding*. Under our consideration the algorithm is

of type *Byte Sequence Existence* since it only verifies the existence of the sequence of bytes, as the name of the category itself indicates.

5.5.7 sdhash

In 2010, Vassil Roussev proposed a new approach based on *Statistically-Improbable Features*, which is based on selecting features that are less likely to occur in other objects by chance [Roussev, 2010]. This approach was implemented in sdhash.

In the FEATURE GENERATION phase, sdhash extracts overlapping 64-byte blocks as features, which differ by one byte. This means that features have a *static* **length**, so a **splitting function** is not necessary (*none*). There is (*yes*) **intersection** due to overlap between features. **Cardinality** ($L - 63$) is *variable*, depending on the input size.

Then, the FEATURE SELECTION phase takes place, in which a subset of *features* (**domain**) is chosen based on the *minimum probability* (**selection function**). In particular, this algorithm obtains a preceding rank value for each feature based on its entropy and empirical observation of the frequency of appearance of each entropy value. Features with extremely high or low entropy are directly discarded, which means that the **coverage** is *partial*. After ranking the features, the most popular features (popularity ≥ 16) are selected to compose the digest. They earn a popularity point each time they are the feature with the leftmost preceding rank value in a 64-feature sliding window.

Next, sdhash performs the PROCESSING FEATURES phase, processing the selected features with the SHA-1 *hash* (**mapping function**) and using 55 bits of the 160-bit SHA-1 hash value (*ratio* **bit reduction**).

The processed features are then inserted into Bloom filters (*set concatenation* **storage structure**), beginning the DIGEST GENERATION phase. When a filter is full, a new filter is created and added to the end of the digest (the **digest size** is *input dependent*). The filters are concatenated in the order of creation, so we consider that the digest has a *set-absolute* **order**. In addition, since the selection phase discards the features with extremely high or low entropy, sdhash has the **generation requirement** of input *diversity* to produce the features. Finally, the FEATURE DEDUPLICATION phase is implemented in the *generation* phase (**occurrence**). If a processed feature is already included *in-scope* of the current filter (**type**), it is discarded.

Regarding the DIGEST COMPARISON phase, sdhash calculates the similarity based on the similarity between the Bloom filters, considering only the greatest similarity between each pair of filters (*average Hamming distance* **method**). It has three **comparison requirements**: *i*) it requires an *ordered cardinality*, where the first Bloom filter has elements less than or equal to the second. The comparison function solves this requirement, by sorting the Bloom filters before the comparison; *ii*) it requires a *minimum cardinality* per filter (16 features) to calculate the similarity between two filters; and *iii*) it requires a *minimum commonality* (more than 30% of the maximum possible common bytes minus the common bytes by chance) between filters. When any of these requirements is not fulfilled, the similarity score is zero. The similarity between the Bloom Filters that share between 30% and 100% of common bytes is assigned in

the range of 0 to 100. Therefore, the similarity score is an interval $[0, 100]$, where 0 indicates no similarity while 100 indicates identical inputs (*interval* **output score** and *ascending* **score trend**). Finally, Bloom filters do not maintain any order in the elements contained in the filter, so the algorithm cannot know if two sets of features maintain the same order (*partial* **spatial sensitivity**).

`sdhash` is classified as a *Statistically-Improbable Features* algorithm for its proposal to select features. However, we believe it should be *Feature Sequence Hashing* due to its general behavior.

5.5.8 `MRSH-v2`

In 2012, Frank Breitinger and Harld Baier conducted a study on the security properties of Similarity Digest Algorithms. As a result, they presented `MRSH-v2`, a new version of `MRS hash` that meets their proposed security properties [Breitinger and Baier, 2012b].

In the GENERATION FEATURES phase, `MRSH-v2` uses a *trigger* function as a **splitting function**. Specifically, it goes back to the `ssdeep` original rolling hash because its performance is superior. The trigger function is able to calculate the next output based on the previous calculation, removing the effect of the first byte and adding the effect of a new byte. `MRSH-v2` maintains the minimum feature size of its predecessor. Thus, the feature **length** is *dynamic* with an expected length of 320 bytes with *no* **intersection** and *variable* **cardinality** ($L/320$).

In the next phase, FEATURE PROCESSING, it processes the features with the FNV-1a *hash* function (**mapping function**). Then 55 bits out of 64 are considered to make up the processed feature which defines the **bit reduction** *ratio*.

Regarding the FEATURE SELECTION phase, this algorithm selects all the generated features to build the digest. Hence, the **selection function** is *none* and the **coverage** is *complete*.

To build a digest, in the DIGEST GENERATION phase, `MRSH-v2` uses Bloom filters (*set concatenation*) as **storage structure** with *none* **generation requirements**. Like all algorithms that use Bloom filters, the **digest size** is *input dependent* and the **order** is *set-absolute*. Likewise, the FEATURE DEDUPLICATION phase is implemented by the Bloom filter (**occurrence** in *generation*), eliminating duplicate features that belong to the same filter (*in-scope* **type**).

In the last phase, the DIGEST COMPARISON phase, `MRSH-v2` uses the comparison **method** proposed by Roussev for `sdhash`, the *average Hamming distance* between Bloom Filters, with the same three **comparison requirements**: *i*) the *ordered cardinality* resolved by the implementation; *ii*) the *minimum cardinality* of features (8 features); and *iii*) the *minimum commonality*, 30% to begin to reflect any similarity. The **output score** is in an *interval* of *ascending* **score trend** and the **spatial sensitivity** is *partial* due to the Bloom filters.

This algorithm is considered a *Context Trigger Piecewise Hashing* by the *trigger* function. However, we consider it to be a *Feature Sequence Hashing* due to the concatenation of all the processed features in the Bloom filters.

5.5.9 `mvHash-B`

Frank Breitinger et al. published `mvHash-B` in 2013. As the authors state, "*the algorithm is based on the idea of majority voting in conjunction with run length encoding to compress the input data and uses Bloom filters to represent the fingerprint*"/digest [Breitinger et al., 2013].

In the first phase, FEATURE GENERATION, the algorithm divides the inputs into *statics* features whose **length** are 21 or 51 bytes, depending on the type of input. Therefore, the **splitting function** is not required (*none*). There is (*yes*) **intersection** between features and the **cardinality** is L as a feature is produced for each byte of the input.

As a second step, there is a non-strict FEATURE SELECTION phase that returns an identifier from a fixed set based on *block similarity* (**selection function**). In particular, the algorithm implements a majority voting approach to return a byte of ones or zeros when the number of ones is over or below a threshold t. This process is repeated for all *features* (**domain**). We affirm that it is not strictly a selection phase because it generates an identifier for all features from the previous phase, so the **coverage** is *complete*. This phase can be viewed as a standard feature selection phase, establishing two thresholds that creates a gap of unselected features between them.

The sequence of identifiers is then processed in the FEATURE PROCESSING phase, *encoding* (**mapping function**) the sequence with a Run Length Encoding (RLE) algorithm—*"RLE simply counts the amount of identical consecutive bytes and returns this number"* [Breitinger et al., 2013]. After that, each value is reduced modulo 2 to 1 bit. Hence, the **bit reduction** is a *ratio* of one bit per 32-bit integer, 1/32.

In the next phase, DIGEST GENERATION, `mvHash-B` groups 11-bit sequences from a sliding window with 2-bit steps. These groups are added to a *set concatenation* (Bloom filters) that is used as the **storage structure**. The **digest size** is *input dependent* and the **order** is *set-absolute*. The **generation requirement** for this approach is *diversity* to generate 11 identifier sequences and obtain at least 1 element for the Bloom filter. As in the previous cases where the storage structure is a concatenation of BF, this algorithm implements a FEATURE DEDUPLICATION phase in the *generation* phase (**occurrence**) in which duplicate features are discarded when they are stored in the same BF (*in-scope* **type**).

Finally, in the DIGEST COMPARISON phase, the algorithm makes a comparison of all with all of the Bloom filters—computing the Hamming distance between filters—and averages the lowest distance score (*average Hamming distance* **method**). The **comparison requirement** is a *similar cardinality*, less than 4 Bloom filters apart. The **output score** generated by this approach is an *interval* with *ascending* **score trend** that has *partial* **spatial sensitivity** due to the concatenation of the Bloom filters.

Gayoso et al. [Gayoso Martínez et al., 2014] consider this algorithm as a *Block-Based Rebuilding*. However, we believe it is a *Byte Sequence Existence* algorithm because it calculates the similarity between each feature and a fixed set and produces identifiers. As we explained above, there are only 2 fixed sequences and the feature selection phase always returns an iden-

tifier, but it can be adjusted by thresholds.

5.5.10 `TLSH`

In 2013, Oliver et al. published `TLSH`, a similarity digest algorithm based on LSH [Oliver et al., 2013].

As a FEATURE GENERATION phase, this algorithm generates all possible trigrams without repetition from a 5-bytes sliding window. This means that the **length** of the features is *static*, 3 bytes; there is (*yes*) **intersection** between features, but **splitting function** is not necessary (*none*), and the **cardinality** is *variable* (6*L*).

Then, in the FEATURE PROCESSING phase, the trigrams/features are hashed (*hash* **mapping function**) with the Pearson hash, producing a byte that is directly used as an identifier/processed feature in the following phases. The **bit reduction** is *none* in this case.

Regarding the FEATURE SELECTION phase, the algorithm considers all features, so there is no **selection function** (*none*) and the **coverage** is *complete*.

In the DIGEST GENERATION phase, feature identifiers are aggregated across 128 counters to calculate their quartiles. The digest consists of the logarithm of the input size and the ratio between quartiles as the heading, while the identifier of the quartiles for all the counters is the body. Therefore, the **digest size** is *fixed* and uses *counters* as **storage structure**. The **order** between the input sequences is maintained by the intersection between the features and the section of all features, *processed feature–aware*. The algorithm has a **generation requirement**: the input must have enough *diversity* to ensure that the quartile values are different. Furthermore, it does not implement a FEATURE DEDUPLICATION phase (*none* **type**) because it calculates the similarity between two inputs by comparing the ratio of equal features.

For the DIGEST COMPARISON PHASE, `TLSH` uses a *weighted edit distance* as the comparison **method**. This method compares the quartile of each pair of counters and adds a value based on how many quartiles there are between them. For this algorithm, the **comparison requirements** are *none* while it has a *half-bound* **output score** and *descending* **score trend** with no **spatial sensitivity** (*none*).

In both cases (the previous classification and our proposal), `TLSH` is a *Locality-Sensitive Hashing* algorithm.

5.5.11 `saHash`

In 2014, Breitinger et al. presented `saHash`, an SDA based on a modular design and operating in linear time [Breitinger et al., 2014d].

The FEATURE GENERATION phase for `saHash` divides the inputs into bytes to generate features. Therefore, the feature **length** is *static* and the **splitting function** is *none*. In addition, it considers transition between bytes, making a four-bit circular shift to the left of the input

and producing one-byte features of the new input (transition features), so there is (*yes*) **intersection** between features. This generation of double features makes a *variable* **cardinality** ($2L$).

Since the features are already one-byte and all are considered for the digest, there are no FEATURES PROCESSING (*none* **mapping function** and *none* **bit reduction**) or FEATURES SELECTION (*none* **selection function** and *complete* **coverage**) phases.

In the DIGEST GENERATION phase, four sub-hashes functions process the features. The first function simply returns the input length. The second function returns the frequency of each feature. The third returns the frequency of the transition features (4-bit circular left shift). The last function returns the standard deviation of each byte frequency. These four results are concatenated to build the digest, generating a *fixed* **digest size** formed by the *counters* (**storage structure**). The **order** between features is maintained by the last two sub-hashes (*processed feature-aware*). For this phase, there are no **generation requirements** (*none*). As before, there is no FEATURE DEDUPLICATION phase (*none* **type**) because the algorithm considers the frequency of the features.

In the COMPARISON DIGEST phase, there is a comparison function for each sub-hash, all based on the *weight edit distance* (**method**). Their results are aggregated to generate a pair of values. These values are compared with two thresholds to determine whether or not there is similarity. Hence, the **output score** is a *binary value*. In addition, the **spatial sensitivity** is *total* because the digest reflects any changes to the input. The **comparison requirements** are *none* for this algorithm.

There is no pre-classification for this algorithm, but we consider it to be a *Locality-Sensitive Hash* due to the use of the byte frequency and its transition.

5.5.12 LZJD

In 2017, Edward Raff and Charles Nicholas presented a new metric, Lempel-Ziv Jaccard Distance (LZJD), to calculate distances between sequences of bytes. Their approach is based on the Normalized Compression Distance (NCD) [Raff and Nicholas, 2017].

The FEATURE GENERATION phase of LZJD converts a sequence of bytes into a set of subsequences of bytes, creating a set of *unique* sequences (**splitting function**). The idea is that, starting with an empty set and a one character subsequence, it adds all the subsequences to the set where the current subsequence does not belong to the set, starting a new subsequence with the next character. If the current subsequence belongs to the set, a character is added to the current subsequence and its membership is checked again. This approach generates features with *dynamic* **length**, ($1 + \log_{256} L$), with *no* **intersection** and *variable* **cardinality** ($L/(1 + \log_{256} L)$).

The features are then processed in the FEATURE PROCESSING phase. Each feature is processed with the MD5 *hash* function (**mapping function**). For this algorithm, the **bit reduction** is *none*. As a FEATURE SELECTION phase, 1000 *processed features* (**domain**) with the *min-*

imum value (**selection function**) are selected to construct the digest and therefore the **coverage** is *partial*.

In the DIGEST GENERATION phase, the *set* (**storage structure**) of the 1000 features produces a *fixed* **digest size**, while the **order** is *none*. The **generation requirements** are also *none*.

The FEATURE DEDUPLICATION phase is not required (**type** *none*) for this algorithm due to the selection of unique features and then the computation of cryptographic hashes without bit reduction. Therefore, the processed features in the digest must be unique. This method has the same collision probability as the MD5 hash algorithm.

For DIGEST COMPARISON, `LZJD` uses the *Jaccard similarity* **method** to measure the similarity between pairs of sets/digests. The **comparison requirements** are *none*, while the comparison method produces a similarity score in an *interval* (**output score**) with an *ascending* **score trend**. Since the digest is an unordered set, the **spatial sensitivity** of the comparison is *none* too.

There is no pre-classification for this algorithm. We consider it to be a *Byte Sequence Existence* algorithm because it measures the similarity between two inputs by the number of equal sequences within both.

5.5.13 `FbHash`

Donghoon Chang et al. published `FbHash` in 2019 [Chang et al., 2019], which is based on the Term Frequency - Inverse Document Frequency (TF-IDF) numerical statistic.

The GENERATION FEATURE phase of this algorithm splits the input into 7-byte features (*static* **length**) with (*yes*) **intersection**. There is one byte difference between consecutive features. This schema does not need (*none*) **splitting function** and the **cardinality** is *variable* (L).

In the next phase, FEATURE PROCESSING, the features are processed with a rolling hash (*hash* **mapping function**) that produces a 64-bit unsigned integer. The rolling hash needs a parameter n, which is a large prime number $n \in (2^{52}, 2^{64})$ to limit the output value below n. This algorithm does not (*none*) make **bit reduction**, since it uses all bits as identifier of the processed feature.

In the FEATURE SELECTION phase, there is no **selection function** (*none*) so the **coverage** is *complete*.

To produce a digest, the DIGEST GENERATION phase has a **generation requirement**, a *document frequency* that stores how often each feature is unique in a set of training documents. With the frequency of the input features and in the document frequency, `FbHash` calculates a weighted frequency of each feature, using an array of *counters* as the **storage structure**. The **digest size** is *fixed* by the n parameter of the feature processing phase because it sets the number of different features. Finally, the **order** between the features is maintained by overlap between them, *processed feature-aware*.

This algorithm does not implement the FEATURE DEDUPLICATION phase, so the **type** is *none*.

In the DIGEST COMPARISON phase, the similarity between digest is calculated by the *cosine similarity* of the digest. The **comparison requirements** are *none* for this algorithm, while it produces an *interval* **output score** with *ascending* **score trend**. Finally, the **spatial sensitivity** is *none* because the algorithm only considers the frequency of the features and not their position.

Neither the authors nor the related literature have proposed a classification for the `FbHash` algorithm. We have classified it as a *Locality-Sensitive Hashing* algorithm because it splits the input into many small features and aggregates the processed values into a small set of counters.

Chapter 6

Attacks against Similarity Digest Algorithms

This chapter summarizes the main contributions of this dissertation related to the classification of attacks against SDA according to their purpose [Martín-Pérez et al., 2021b]. Furthermore, for each algorithm considered, we study the characteristics that make them vulnerable to each type of attack, explaining a known attack or a theoretical attack for each possible case. In particular, we are interested in knowing how robust the algorithms are against attacks, finding the characteristics that make them more vulnerable to manipulation by adversaries. We first define our adversary model and then introduce the classification of attacks against SDA.

6.1 Adversary model

We assume an intelligent adversary who knows the processes and techniques used by the SDA that she wants to attack. Therefore, the adversary can classify the SDA according to our classification scheme, either by reverse engineering, by performing a source code analysis, or by reviewing the available literature/documentation of the algorithm.

Regarding the attack scenarios, for the sake of simplicity and without loss of generality we assume that the lowest value of the SDA similarity score (value of 0) indicates that there is no similarity, while the highest value (value of 100) indicates perfect similarity (that is, the similarity score trend is trending upward).

6.2 Classification of SDA attacks

As mentioned before, we study the possible attacks against SDA and the characteristics of the algorithms that facilitate these attacks. We do not claim that this section is complete, as other

Algorithm	Attacks against the similarity score		Attacks that bypass any of the phases of an SDA	
	Similarity Reduction	*Similarity Emulation*	*Bypassing the Digest Generation Phase*	*Bypassing the Digest Comparison Phase*
dcfldd	**Length**: *Static* **Intersection**: *No*	None	None	None
Nilsimsa	**Intersection**: *Yes*	None	None	None
ssdeep	**Comparison requirements**: *Minimum commonality*	**Cardinality**: *Fixed* **Splitting function**: *Trigger* **Bit reduction**: Low *Ratio*	**Generation requirements**: *Minimum features*	**Type**: *Consecutive* **Occurrence**: *Comparison* **Comparison requirements**: *Minimum commonality*
md5bloom	**Length**: *Static* **Intersection**: *No*	None	None	None
MRS hash	**Storage structure**: *Set concatenation*	None	None	None
SimHash	**Selection function**: *Block matching*	**Coverage**: *Partial*	None	None
sdhash	**Storage structure**: *Set concatenation*	**Coverage**: *Partial* **Comparison requirement**: *Ordered cardinality*	**Generation requirement**: *Diversity*	**Comparison requirement**: *Minimum cardinality*
MRSH-v2	**Storage structure**: *Set concatenation*	None	**Comparison requirement**: *Minimum cardinality*	**Comparison requirement**: *Minimum cardinality*
mvHash-B	**Selection function**: *Block similarity*	None	**Generation requirement**: *Diversity*	None
TLSH	**Intersection**: *Yes*	None	**Generation requirement**: *Diversity*	None
saHash	None	None	None	None
LZJD	**Selection Function**: *Minimum value*	**Coverage**: *Partial*	None	None
FbHash	**Intersection**: *Yes*	None	None	None

Table 6.1: Sets of characteristics that allow attacks, grouped by attack type and algorithm.

attacks may exist. However, as far as we know, this is the first methodological approach to defining attacks against SDA.

We distinguish two types of attacks: *Attacks against the similarity score* and *Attacks that bypass any of the phases of an SDA*. In Table 6.1, we summarize the set of characteristics that allow the most feasible attack for each algorithm. We explain these attacks in more detail below.

6.2.1 Attacks Against the Similarity Score

The goal of these attacks is to affect the similarity score of a crafted artifact when it is compared to an original artifact. An adversary can construct an input that generates a digest that returns a certain value when compared to a target digest. We distinguish two subtypes of this attack: *similarity reduction* and *similarity emulation*.

Similarity Reduction

These attacks aim to minimize the similarity score between two inputs. Consequently, the adversary is interested in creating a new digital artifact (*crafted artifact*) from a given artifact knowing that they will eventually be compared.

The attack that affects SDA the most is the modification of random bytes. This attack takes advantage of *hash* or *identifier* **mapping functions**. For SDA with these characteristics, modifying a byte of a feature makes the processed feature different due to the avalanche effect of the hash function (meaning that despite how similar two inputs are, their outputs will differ by approximately 50% of the bits) or due to the inconsistency of the feature with its corresponding identifier. Thus, the similarity is reduced to zero when enough bytes are modified. As an exam-

ple, [Oliver et al., 2014b] showed that random changes in the input can reduce the similarity score. In particular, they evaluated these attacks against `ssdeep`, `sdhash`, and `TLSH`.

One of the most trivial methods to achieve the above attack is to add or delete a byte at the beginning of the input, when the algorithm has *static* **length** and *no* **intersection** characteristics. By doing this, all subsequente features are modified (because all offsets are shifted). This kind of attack, for example, was described in [Baier and Breitinger, 2011] against `dcfldd`, although it is also viable against `md5bloom`.

On the other hand, when SDA have **intersection** (*yes*) between features, random byte modification is more effective because a simple modification can affect multiple processed features. For example, the authors of `Nilsimsa` state that the similarity between two related input drops under the required threshold by randomly modifying 20% of the input, because each single-byte modification changes 24 processed features [Damiani et al., 2004]. `TLSH` is also affected by this weakness because it is based on `Nilsimsa` and has the same feature generation and feature processing phases.

In theory, it is possible to optimize random modification when the comparison phase has a **comparison requirement** of *minimum commonality*. An attacker only needs to ensure that this requirement is not met. Baier and Breitinger [2011] demonstrate this attack for `ssdeep`. Since the algorithm needs 7 consecutive common features, the attacker must simply modify 1 of the 7 features to reduce the similarity score to zero. Recall that the trigger function of `ssdeep` generates up to 64 features, and thus, it is sufficient to modify at most $\lfloor 64/7 \rfloor = 9$ bytes.

Similarly, an SDA that **selects features** based on *block matching* or *minimum values* can be defeated by finding these particular sets of blocks or features and modifying them accordingly. For instance, `SimHash` counts the number of 16 blocks of 1 byte [Sadowski and Levin, 2007]. If an adversary looks for these blocks and modifies them in the crafted input, the similarity score is reduced. Likewise, `LZJD` stores the lowest 1000 hashes of the features [Raff and Nicholas, 2017]. Hence, an adversary who detects the input characteristics used to form these features can modify them to reduce the similarity between objects.

SDA that use *set concatenation* as the **storage structure** are currently based on Bloom filters. While this is a space-efficient probabilistic data structure, it has a problem that is often denoted as shifting, meaning that half of the features from one filter are shifted to the next filter. For instance, consider two Bloom filters, BF_1 and BF_2, with a maximum capacity of 10 features and containing 10 features each. If an adversary adds 5 features to BF_1 (e.g., adds data to the beginning of the artifact), their overlap (similarity) drops to 50. This attack works well for `MRS hash` or `MRSH-v2`. In addition, when the algorithm also has the **comparison requirement** of *minimum commonality*, then the similarity result is worse. This is the case of `sdhash`, for which [Breitinger and Baier, 2012a] showed that this attack reduces the similarity to approximately 28.

Last, `mvHash-B` is the only SDA that uses a Run Length Encoding algorithm (*encoding* **mapping function**), which returns the number of consecutive equal identifiers. It then performs modulo 2 of each RLE element, getting the lowest **bit reduction** *ratio* among all SDA considered in this thesis—processed features are 1 bit in size. These characteristics make `mvHash-B`

vulnerable to similarity reduction attacks. As demonstrated by [Singh et al., 2016], an intelligent adversary can modify features to produce different identifiers. Changing an identifier will cause modifications to the encoding sequence, dividing the quantity of each affected number into three elements or moving one identifier between consecutive RLE elements. In the first case, two new processed features are added after the original. In the second case, two processed features are modified. Both cases produce a significant change that modifies several BF elements. Repeating the attack at different input positions, Singh et al. [2016] manage to reduce the similarity score to 4 by modifying only 3% of the input.

Similarity Emulation

The opposite attack to the one presented above is *similarity emulation*, where an adversary is interested in creating a new digital artifact that produces a high similarity score compared to a known artifact.

When a SDA has a *fixed* **cardinality**, a *trigger* function such as a **splitting function** and a low **bit reduction** *ratio*, an adversary can create an input that generates several small features with the beginning of the input, which then generate an arbitrary sequence of processed features. For instance, [Baier and Breitinger, 2011] showed that it is possible for `ssdeep` to include trigger sequences (i.e., features) in an EXIF image data. Consequently, the authors were able to manipulate an image to match any similarity digest.

Partial **coverage** allows an adversary to modify the gaps taking care that these modifications do not alter the feature generation and selection phases. For `sdhash`, according to [Breitinger et al., 2012], up to 20% of the content of an input can be modified without influencing the generated digest. This claim was later demonstrated by [Chang et al., 2015]. `SimHash` is also vulnerable to this attack: an adversary can modify any byte of the input if these modified bytes do not match with any of the fixed blocks. Similarly, in `LZJD` any data can be altered after the last selected feature, as long as it does not generate a processed feature with a value less than the maximum of the selected processed features.

There is also a more sophisticated attack against `sdhash` based on *partial* **coverage** and *ordered cardinality* (**comparison requirements**). To compare two digest, `sdhash` swaps the input digest to fulfill the ordered cardinality ($|d_1| < |d_2|$) if necessary. It then calculates the average similarity of the maximum score between each filter in the smallest digest against each filters in the other. Therefore, if one digest has only one filter, the comparison method only compares the maximum similarity of this filter to each filter in the second digest. Therefore, an adversary can obtain the maximum similarity by creating a new artifact that copies the first part of a known artifact, where there are features that make up the first filter, and then by adding content not selectable by the selection function. For example, to create an artifact similar to an application file, it can start with the same content of the legitimate program file, and then fill the remaining content with short sequences of instructions that end in a jump assembler instruction. The jump chains the current sequence with the next, creating spaces between the sequences. The content of the gaps must ensure that the selection function discards them. Low

entropy gaps are viable, but high entropy gaps avoid highlighting the attack sequence. Finally, the **comparison requirement** of *minimum cardinality* in a filter is not necessary, but it does make it easy to create an emulated digest.

6.2.2 Attacks that bypass any of the phases of an SDA

The objective of these attacks is to affect the operation of the SDAs by affecting some of their phases, either to avoid the generation or comparison of digests.

Bypassing the Digest Generation Phase

The goal of the adversary is to complicate the generation of the digest, i.e., to modify an input so that the algorithm cannot generate a similarity digest due to the lack of necessary conditions. These attacks can take advantage of the previous phases to obtain the necessary conditions to avoid the digest generation.

SDA that need a *minimum features* as a **generation requirement** are vulnerable to this type of attack. This is possible because the *trigger* function (**splitting function**) can be manipulated by means of a forged input to produce insufficient features to generate the digest. For example, `ssdeep` can be attacked with an input that intentionally avoids byte sequences matching the value of the trigger function. As a result, only a few features are generated and therefore `ssdeep` cannot create the digest. Note that the algorithms can implement countermeasures such as adjusting the splitting function appropriately.

Other algorithms need input *diversity* as a **generation requirement**. This requirement can also be exploited by an intelligent adversary. For instance, `sdhash` discards all features and does not generate a digest if the entropy of the input features is too high or too low. Likewise, `mvHash-B` compares the features with two blocks and keeps the identifier of the most similar block. The generation of digest is based on how many consecutive identifiers of a given type are found. Hence, if all the features are similar to a single block, there is only one sequence of identifiers, which is insufficient to generate the digest. This attack can require a lot of changes and therefore may be impractical for real world scenarios.

Finally, `TLSH` is a special case of *diversity* as a **generation requirement**. It has no explicit requirement of diversity but calculates, as part of the digest, the ratio between quartiles of the counters (q_1/q_3 and q_2/q_3). So, if the diversity in the input is too low, and adding all the identifiers in 25% of the counters or less it is obtained that q3 is equal to 0, division by zero are generated that prevent the generation of digests.

Bypassing the Digest Comparison Phase

In this case, we contemplate an adversary with the aim of hindering the digest comparison process. To achieve this, the adversary constructs an input so that the generated similarity digest cannot be compared or, if comparable, the similarity score means dissimilarity.

Algorithms that have a feature deduplication phase of the **type** *consecutive* with the **occurrence** in the digest *comparison* phase and also need a *minimum commonality* as a **comparison requirement** to compare digests are vulnerable to this type of attack. An intelligent adversary can create an input so that the digest, calculated after deduplication, has fewer common elements than expected, and therefore the digest is incomparable. For instance, `ssdeep` can be attacked with an initial sequence of 63 times a small feature that matches the trigger function, followed by any content. The digest generated in this case is a sequence of a processed feature repeated 63 times and a random trailing character. At comparison time, the deduplication phase will replace the 63 characters with 3 characters, generating a 4-item digest that will never meet the minimum commonality comparison requirement.

Likewise, if SDA have a *minimum cardinality* as a **comparison requirement**, an adversary can create an input so that the algorithm produces a similarity digest with fewer items than is necessary for comparison. For instance, `sdhash` needs 16 features per Bloom Filter to compare two filters [Roussev and Quates, 2013]. However, because its selection function (feature selection phase) is based on entropy, an adversary can manipulate an input to have extremely low or high entropy so that features are discarded. If the input generates a similarity digest with only one filter and fewer than 16 features, the comparison is impossible. A similar attack is possible against `MRSH-v2`: an adversary can create an input to avoid the match of the trigger function, generating so few features that the sets do not meet the comparison requirements.

6.3 Towards Building a Robust SDA

This section describes the characteristics that we consider desirable to build a robust SDA, where robust is defined with respect to *resilience against attacks*. This section can be viewed as a *guideline to building a robust similarity digest algorithm*. Note that we are not focusing on perfect error rates or runtime efficiency.

For the FEATURE GENERATION PHASE, we consider that the **length** must be *static* and with (*yes*) **intersections**—the overlap between features is desirable because it allows detecting the interchange of features (processed feature–aware order). This last characteristic is not necessary if the selection phase has complete coverage—and with *variable* **cardinality**.

Algorithms that use a dynamic feature size need a splitting function that divides the input in some way. However, this function is susceptible to being attacked by manipulating the sequence of features. Therefore, it should be avoided.

Regarding the FEATURE PROCESSING PHASE, a **mapping function** that generates similar outputs for similar inputs is preferable. For example, the *block similarity* function used by `mvHash-B` as the selection function is a good choice. **Bit reduction** is acceptable if the output provided by the mapping function output is large enough. However, the final amount of bits used must be resistant to collision attacks and must preserve the similarity relationship between the outputs.

In contrast, using of a hash mapping function implies that a small change in one feature

(e.g., modifying just one byte) causes a totally different processed feature, whereas low bit reduction rate can allow an attacker to create an artifact with a desirable digest (e.g., similarity emulation against `ssdeep`).

In the FEATURE SELECTION PHASE, *complete* **coverage** is absolutely necessary to avoid uncovered gaps that can be used to hide data and still get good similarity scores, whereas a **selection function** can be a good option to reduce the digest size when the feature generation phase has intersection.

For the DIGEST GENERATION PHASE, the desirable characteristics are *input dependent* **size** with *absolute* **order** and no **generation requirements** (*none*). The requirements in the digest generation phase impose conditions in which an adversary can focus to hamper the digest generation process. Similarly, a FEATURE DEDUPLICATION PHASE is undesirable because it removes information.

Regarding the DIGEST COMPARISON PHASE, having **comparison requirements** makes it easier for an adversary to obstruct the comparison process, as explained before. Finally, a comparison function with *total* **spatial sensitivity** is also desirable, since it allows to identify modifications in the order of the features.

Discussion about digest size. The desired **digest size** is difficult to answer. As stated by Breitinger et al. [2014b], a fixed-size digest is preferable. However, our study of SDA revealed that it is difficult to design a robust algorithm with this characteristic. We have found three SDA that have this characteristic, but they all have some limitations. For instance, using features with dynamic length and a fixed-size digest has trouble comparing inputs with very different sizes (for example, `ssdeep`). Likewise, the selection of a limited cardinality of features such as the representation of the whole (as `LZJD` does) implies a partial coverage, leaving gaps that can be exploited by an attacker inserting arbitrary content. Finally, counting features in a limited set of counters (as `TLSH` and `Nilsimsa` do) causes loss of order between features, allowing reorganization of the input without affecting the similarity score. This problem is partially solved by considering feature intersection (processed feature–aware). The latter case is the solution least vulnerable to attacks, but we still believe that maintaining order between features with an input-dependent digest is more valuable than having better performance for using a fixed size digest. More research is needed in this regard.

Part III

Solutions to Minimize Memory-Related Problems

Chapter 7

Pre-Processing Methods for Normalizing the Variability of Modules

In this chapter, we describe our proposed solutions to mitigate the side effects produced by ASLR, which causes an unintended reduction of the similarity between modules. As shown in Section 4.2, an easy way to reduce the similarity of any SDA is to modify sparse bytes. In fact, this is exactly what ASLR does as a side effect.

To mitigate this behavior and to be able to use SDA to identify running modules, we propose to normalize all the affected bytes, setting them to zero. In particular, we present two solutions: *i*) GUIDED DE-RELOCATION, a method that attempts to retrieve the information that has been used by image loader to relocate the module; and *ii*) LINEAR SWEEP DE-RELOCATION, another method that identifies modified bytes when the information required by the previous method is inaccessible. This method searches the longest sequence of plausible instructions to find the affected addresses. Both methods are published in [Martín-Pérez et al., 2021a].

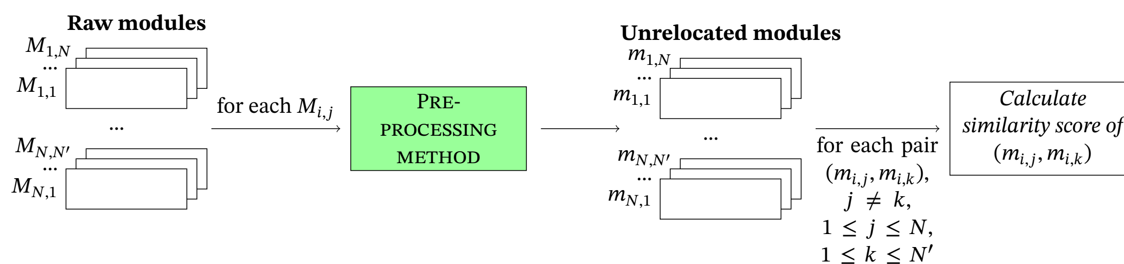


Figure 7.1: Sketch of the system model. The place where our proposed methods can take place has been highlighted.

Figure 7.1 sketches the system model of our problem. The raw modules $\{M_{i,1}, ..., M_{i,N}\}$ from a memory dump, obtained in a forensically sound manner, are pre-processed with any of the pre-processing methods proposed in this work. The extracted modules $\{m_{i,j}\}$ have been appropriately derelocated and thus the similarity scores of each pair $(m_{i,j}, m_{i,k}), j \neq k, 1 \leq j \leq N, 1 \leq k \leq N', N \neq N'$ are more accurate than when the pre-processing method is not applied. We have highlighted in the figure where the pre-processing methods that we propose can take place.

7.1 Pre-Processing Methods

To formally present the algorithms of the pre-processing methods, we have adopted the notation described in Table 7.1. Below, we explain in detail how our proposed pre-processing methods work.

7.1.1 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 each 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/linker. It contains the necessary information to allow the image loader to make any adjustment needed in the program binary code and data of the application due to relocation.

The information of the `.reloc` section is divided into blocks, where each block represents the adjustments needed for a 4K page. Each block contains an `IMAGE_BASE_RELOCATION` structure, which contains the 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 unnecessary to the normal execution of the process, because the kernel is the one who copies pages into memory and does the relocation in the pages when the process requires them. The `.reloc` section is rarely accessed by the process and thus it is rarely copied into the user-space. Luckily, a memory dump may contain other elements rather than the image file that represent the image file and contain a `.reloc` section, such as `File Objects` [Microsoft Corporation, 2019]. 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].

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 with 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, $0 \le i \le 14$, $b_0 = b$ (see further explanation and example in Section 7.1.2).
memoperand(i)	Returns the memory operand of an assembly instruction i.
non-visited_map(s)	Returns an array initialised as non-visited of size s.
visit($\mathcal{V}, \mathcal{S}$)	Mark in the array $\mathcal{V}$ the bytes of the set $\mathcal{S}$ as visited.

Table 7.1: Summary of the formal notation used in the pre-processing algorithms.

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  |   |   B ← blocks(f)
8  |   |   foreach b ∈ B do
9  |   |   |   a ← rvaddress(b)
10 |   |   |   E ← entries(b)
11 |   |   |   foreach e ∈ E do
12 |   |   |   |   derelocate(m, a_m + offset(e))
13 |   |   |   end
14 |   |   end
15 |   |   U = U ⋃{m}
16 |   end
17 end
```

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

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 image loader has relocated it). 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 $\mathcal{D}$ (line 2). In this regard, we have used the Volatility plugin `filescan`, which finds `File Object` structures in physical memory using pool tag scanning [Schuster, 2009a,c,b]. Next, we iterate for each module m in $\mathcal{M}_\mathcal{D}$ (line 3)—the list of modules of $\mathcal{M}_\mathcal{D}$ is obtained using the Volatility plugin `dlllist`. 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 image file in virtual memory) (line 4). Then, we walk through the PE structure looking for each PE field which is a memory address pointing within $\mathcal{A}$ (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 (line 6). If so, the de-relocation process in m is performed, using the information given by the `.reloc` section of f (lines 7–14). First, the set of blocks contained in the `.reloc` section is retrieved and stored in $\mathcal{B}$ (line 7). Then, for each block $b \in \mathcal{B}$ (line 8) the RVA of its page is taken as a (line 9). The set of entries of the block b is stored in $\mathcal{E}$ (line 10). Next, for each entry block $e \in \mathcal{E}$ (line 11), the memory address a plus the offset of the entry e is unrelocated in the module m (line 12). 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 15).

7.1.2 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 file objects of a memory dump. In addition, it may happen that the physical page into which the `.reloc` section was mapped has been out swapped 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 $\mathcal{D} = \langle \mathcal{M}_{\mathcal{D}} \rangle$, where $\mathcal{M}_{\mathcal{D}} = \{m_1, ..., m_M\}$ is the set of modules contained in $\mathcal{D}$. Line 1 initializes $\mathcal{U}$ with empty set. Then, we iterate for each module m that can be retrieved from $\mathcal{M}_{\mathcal{D}}$ (line 2). We use again the plugin `dlllist` of Volatility. In line 3, we first initialize an array $\mathcal{V}$ with length size m as *non-visited*. Then we identify all pages of 4096 bytes swapped out from memory by means of the Volatility framework. In particular, we retrieve the memory address space of each module and then check whether the first byte of each page is valid. A page is valid if it resides in the memory. Each byte of the outswapped page is tagged as *visited* (line 4). Then, in line 5 we retrieve the range $\mathcal{A}$ of virtual memory addresses of m, obtaining its base address and its image size.

This algorithm works in two phases. In the first phase, it processes all the structured data of the PE of m, walking through the PE structure and tagging each byte within the PE structure as visited byte (lines 6 and 7). 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 8). 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 9 to the end). We first retrieve the memory space $\mathcal{C} \subset \mathcal{A}$ into which the code section of m is mapped (line 9). 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.

```
    Input: A memory dump D = ⟨M_D⟩
    Output: Set of unrelocated modules U
 1  U = ∅
 2  foreach m ∈ M_D do
 3  |   V = non-visited_map(length(m))
 4  |   visit(V, empty(m))
 5  |   A ← memory_range(m)
    |   /* Phase 1: structured data processing */
 6  |   P ← PEheader_datadir(m)
 7  |   visit(V, P)
 8  |   ∀p ∈ P : points_to(p) ∈ A ∴ derelocate(m, p)
    |   /* Phase 2: unstructured data processing */
 9  |   C ← section_code(m)
    |   /* Tag lookup tables */
10  |   L ← lookup_tables(C)
11  |   visit(V, L)
12  |   ∀l ∈ L : points_to(l) ∈ A ∴ derelocate(m, l)
13  |   if 32bit_image(m) then
    |   |   /* Tag strings */
14  |   |   S ← strings_padding(C)
15  |   |   visit(V, S)
    |   |   /* Tag byte patterns */
16  |   |   B ← byte_patterns(C)
17  |   |   ∀p ∈ B, U ← subsequent(p) : a = points_to(U) ∈ A∴derelocate(m, a)
18  |   |   visit(V, B)
    |   |   /* Process the rest of bytes in C */
19  |   |   while ∃b ∈ C|b = non-visited do
20  |   |   |   J ← build_sequences(b)
21  |   |   |   I = {I ∈ J : ∀S_i ∈ J, S_i ≠ I, |S_i| < |I|}
22  |   |   |   ∀i ∈ I : a = memoperand(i) ∈ A∴derelocate(m, a)
23  |   |   |   visit(V, J)
24  |   |   end
25  |   end
26  |   U = U ⋃{m}
27  end
```

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

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 10 and 11). For each entry of these tables, the de-relocation process takes place if the entry targets to $\mathcal{C}$ (line 12). 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, although zeroing two bytes instead of six.

In addition, if m is a 32-bit image file (line 13), a little more work is needed (line 14 to line 25). We first aim to identify known byte patterns (lines 14 to 18). 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 14). Each byte of the identified strings is tagged as a visited byte (line 15). 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 16). For each 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 (line 17). As before, each pattern byte identified and the subsequent addresses are also tagged as visited bytes (line 18).

Finally, the last part of the algorithm (line 19 to 24) iterates while exist bytes $b \in \mathcal{C}$ that they were not visited by any of the aforementioned processes. For each non-visited byte (line 19), we build sequences of valid assembly instruction $\mathcal{I}$ (line 20). 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 . 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 each 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, 2020] to obtain the valid sequences of instructions.

Then, we select the longest byte sequence of valid assembly instructions $I \in \mathcal{I}$ (line 21). We iterate in each instruction in this sequence, tagging each 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 22). This iteration finishes marking each byte of the instructions stored in $\mathcal{I}$ as visited (line 23).

The iteration process of the algorithm ends adding the modified module m to the set of unrelocated modules $\mathcal{U}$ (line 26). 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 each operation of Algorithm 2 that works on $\mathcal{C}$ is iterating in each 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).

Detailed example

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:

Listing 7.1: Code example

```
0x1000:     FC                    CLD
0x1001:     FEFF                  ???
0x1003:     FFFF                  ???
0x1005:     E8 39000000           CALL 0x1043
0x100a:     8B45 08               MOV EAX,DWORD PTR SS:[EBP+0x8]
0x100d:     E8 A487FFFF           CALL KernelBa.752917F0
0x1012:     C2 0C00               RETN 0xC
0x1015:     90                    NOP
0x1016:     FE                    ???
0x1017:     FFFF                  ???
0x1019:     FF00                  INC DWORD PTR DS:[EAX]
0x101b:     0000                  ADD BYTE PTR DS:[EAX],AL
0x101d:     00CC                  ADD AH,CL
0x101f:     FFFF                  ???
```

We first get a slice of 15 bytes, starting at byte FC. Figure 7.2 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 (for the sake of simplicity, 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 at 0x1002 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 at 0x1005 is reached. The valid sequence of assembly instructions is:

```
0x1005:     E8 39000000         CALL     0x1043
0x100a:     8B45 08             MOV      EAX,DWORD PTR SS:[EBP+0x8]
0x100d:     E8 A487FFFF         CALL     KernelBa.752917F0
0x1012:     C2 0C00             RETN     0xC
0x1015:     90                  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` because they are the first bytes of instructions which we have considered in the current sequence. However, since the next instruction in the sequence is out of the current slice, the `length` vector is no longer updated, but the list of instruction addresses is stored and checked for all instructions found. Starting the disassembly in byte 39, the sequence of valid instructions is:

```
0x1006:     3900                CMP      DWORD PTR [RAX], EAX
0x1008:     0000                ADD      BYTE PTR [RAX], AL
0x100a:     8B4508              MOV      EAX,DWORD PTR SS:[EBP+0x8]
0x100d:     E8 A487FFFF         CALL     KernelBa.752917F0
0x1012:     C2 0C00             RETN     0xC
0x1015:     90                  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 `0x100a` 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` at `0x1007`, obtaining the following sequence:

```
0x1007:     0000                ADD      BYTE PTR [RAX], AL
0x1009:     008B 4508E8A4       ADD      BYTE PTR [RBX - 0x5b17f7bb], CL
0x100f:     87FF                XCHG     EDI, EDI
0x1011:     FFC2                INC      EDX
0x1013:     0C00                OR       AL, 0
0x1015:     90                  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:     45          INC     EBP
0x100c:     08E8        OR      R8B, R13B
0x100e:     A4          MOVSB   BYTE PTR [RDI], BYTE PTR [RSI]
0x100f:     87FF        XCHG    EDI, EDI
0x1011:     FFC2        INC     EDX
0x1013:     0C00        OR      AL, 0
0x1015:     90          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 considered bytes 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 instructions 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` at `0x1016` which follows the `NOP` assembly instruction (22 bytes after the previous slice), since all the previous bytes are now marked as visited.

slice	FC	FE	FF	FF	FF	E8	39	00	00	00	8B	45	08	E8	A4
length	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Initial condition

slice	FC	FE	FF	FF	FF	E8	39	00	00	00	8B	45	08	E8	A4
length	1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0

After processing the sequence of instructions starting at FC

slice	FC	FE	FF	FF	FF	E8	39	00	00	00	8B	45	08	E8	A4
length	1	-1	-1	-1	-1	0	0	0	0	0	0	0	0	0	0

After processing the sequence of instructions starting at FF

slice	FC	FE	FF	FF	FF	E8	39	00	00	00	8B	45	08	E8	A4
length	1	-1	-1	-1	-1	17	0	0	0	0	-1	0	0	-1	0

After processing the sequence of instructions starting at E8

slice	FC	FE	FF	FF	FF	E8	39	00	00	00	8B	45	08	E8	A4
length	1	-1	-1	-1	-1	17	-1	0	-1	0	-1	0	0	-1	0

After processing the sequence of instructions starting at 39

slice	FC	FE	FF	FF	FF	E8	39	00	00	00	8B	45	08	E8	A4
length	1	-1	-1	-1	-1	17	-1	-1	-1	-1	-1	0	0	-1	0

After processing the sequence of instructions starting at 00

slice	FC	FE	FF	FF	FF	E8	39	00	00	00	8B	45	08	E8	A4
length	1	-1	-1	-1	-1	17	-1	-1	-1	-1	-1	-1	-1	-1	-1

After processing the sequence of instructions starting at 45

Figure 7.2: Example of selection of the longest sequence of instructions, per 15-byte slices.

Chapter 8

Evaluation of Pre-Processing Methods

This chapter introduces our Volatility plugin, `Similarity Unrelocated Module`, which implements the methods presented in the previous chapter. We also describe and discuss the experiments performed to validate both pre-processing methods. The tool is publicly accessible under GNU/GPLv3 license in [Martín-Pérez, 2020] and was published together with the results of the experiments in [Martín-Pérez et al., 2021a].

8.1 The `Similarity Unrelocated Module` Tool

We have implemented both pre-processing methods presented in Chapter 7 in a plugin for the Volatility memory analysis framework [Walters, 2007]. Our plugin, dubbed `Similarity Unrelocated Module` (`SUM`), is an improvement of the previous tool introduced in [Rodríguez et al., 2018]. We have released the code of `SUM` under the GNU GPLv3 license in [Martín-Pérez, 2020].

Unlike the previous tool, `SUM` generates a similarity digest for each resident page of each module that is retrieved from a memory dump, while the comparison is an array of similarity scores per page. A forensic examiner can choose to pre-process each module with the GUIDED DE-RELOCATION method only (when the `.reloc` section is retrievable), with the LINEAR SWEEP DE-RELOCATION method only (applicable in all cases), or with both (`SUM` first tries to retrieve the `.reloc` section to apply the GUIDED DE-RELOCATION method; the LINEAR SWEEP DE-RELOCATION method is applied if the previous one fails). By default, `SUM` does not apply any pre-processing method.

The plugin also supports the use of more than one similarity digest algorithm at a time, the selection of only specific sections of the modules for comparison of similarity, and the selection of processes by PID or processes and shared libraries by name. The actual implementation of `SUM` has integrated the most relevant SDA in the literature (namely, `dcfldd`, `ssdeep`, `sdhash`,

and TLSH). Nevertheless, the tool has a modular design that allows new SDA to be easily added.

8.2 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).

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 modules 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 each 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 page of each module (this usually contains the PE header since the header size is commonly less than 4KiB) plus the pages containing the code section (that is, a subset of the pages of each module).

The similarity of the modules is computed as an aggregate similarity score of pairs of pages of the modules that are comparable. As similarity digest algorithms, we use dcfldd, ssdeep, sdhash, and TLSH (described in Section 5.5). Since the score provided by TLSH has a half-bound output and works in a reverse mode (descending score trend), 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 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: $\texttt{TLSH}_{norm}(x) = \begin{cases} 100\left(1 - \frac{x}{t}\right) & \text{, if } x \leq t \\ 0 & \text{, otherwise} \end{cases}$

Related Comparison

In this case, we compare resident pages in the same relative offset, using the same pre-processing method. We have considered here all memory dumps. In total, we have 16710 resident pages from 38800 total pages in 32-bit scenarios, and a similar number of resident pages (namely, 16831 resident pages) from 42350 total pages in 64-bit scenarios. On average, the number of resident pages obtained in the memory dumps is around 40%, a similar result to that shown in the Section 4.1.2.

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 also that we compare pages versus pages instead of 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 8.1 shows the aggregated similarity scores considering the sixty memory dumps, for each 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 each 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 modules 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.

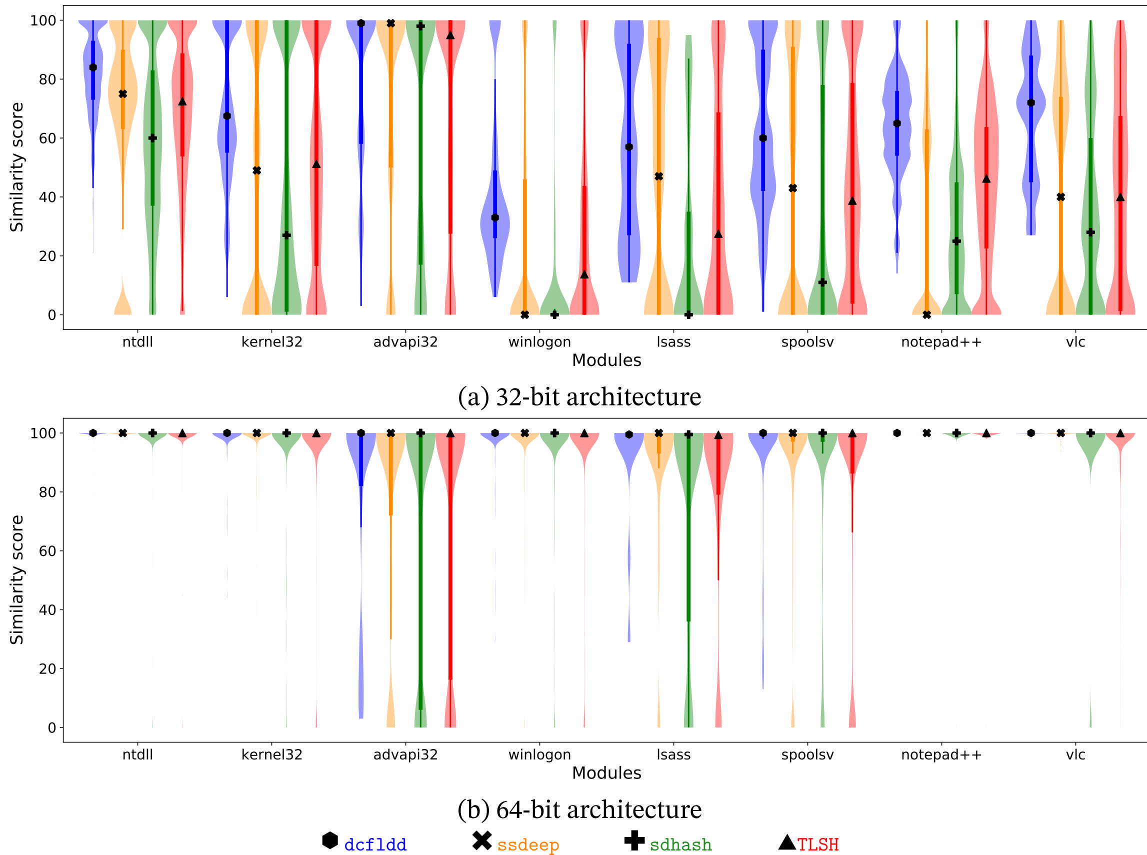


Figure 8.1: Related comparison: similarity scores when none pre-processing method is applied (RAW *scenario*).

In the case of 32-bit architecture, the similarity scores are more dispersed 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, the results on both architectures improve by applying our pre-processing methods.

GUIDED DE-RELOCATION **scenario.** Figure 8.2 shows the results of the similarity score when the GUIDED DE-RELOCATION pre-processing method is applied in the modules of each 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 pages where it was mapped were not present in some of the memory dumps of 32-bit Windows OS machines.

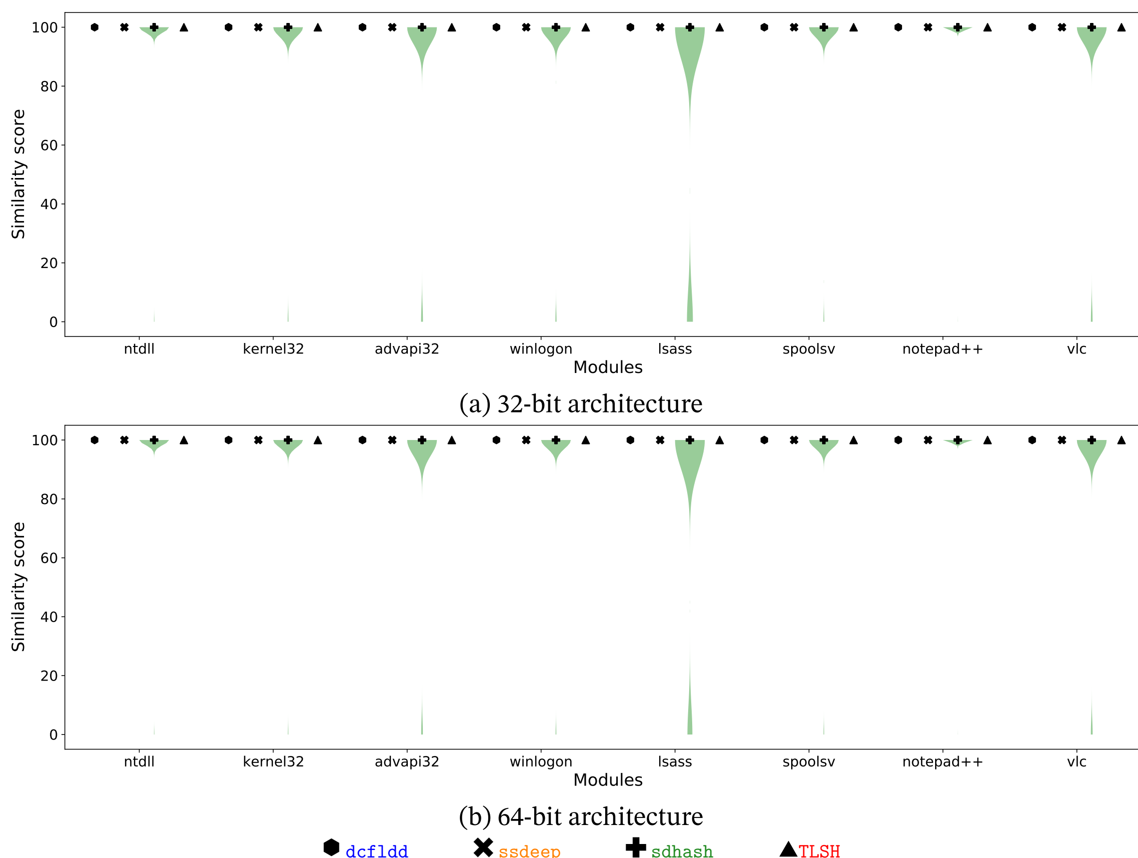


Figure 8.2: Related comparison: similarity scores when the GUIDED DE-RELOCATION pre-processing method is applied (GUIDED DE-RELOCATION *scenario*).

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 each 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 each algorithm and each 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 5.5, `sdhash` has a selection function based on minimum probability. In addition, the algorithm requires a minimum cardinality of features, at least 16, 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 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

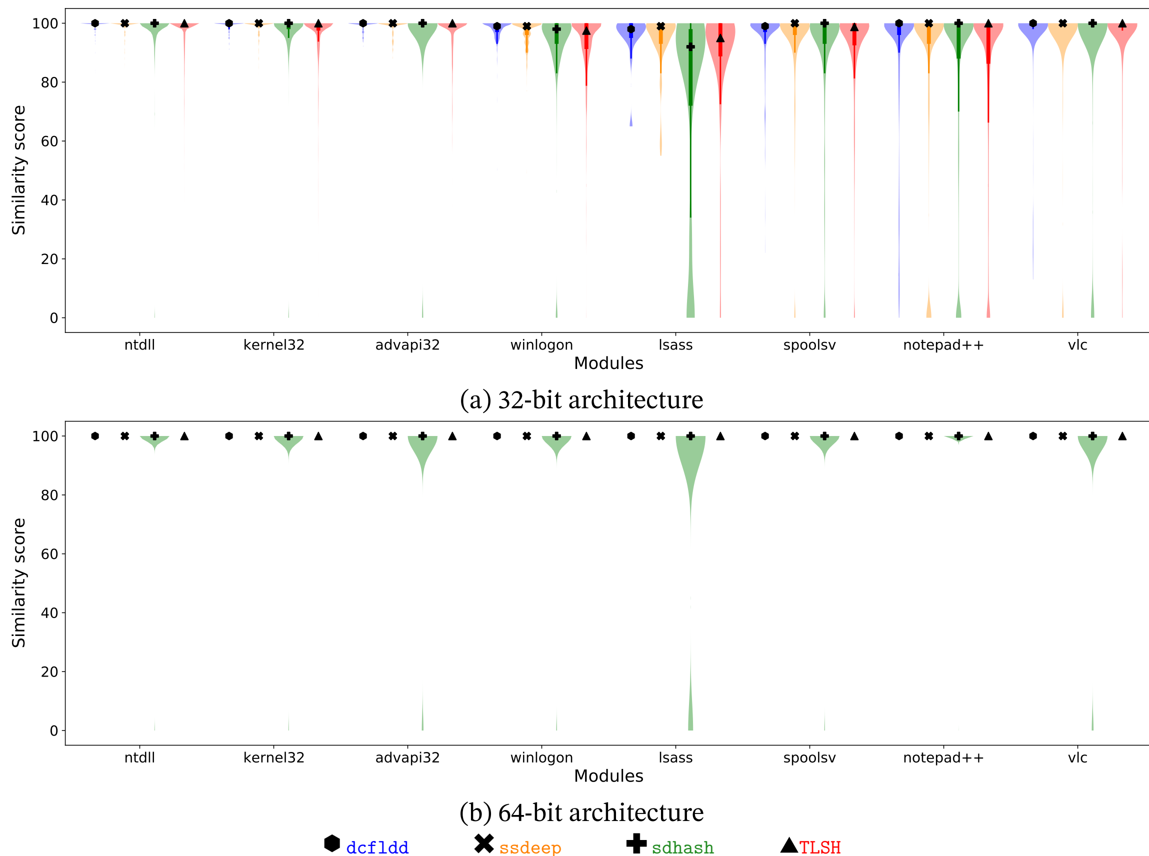


Figure 8.3: Related comparison: similarity scores when the LINEAR SWEEP DE-RELOCATION pre-processing method is applied (LINEAR SWEEP DE-RELOCATION *scenario*).

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 pages. However, the percentage of these pages is quite low (only 180 out of the 2327720 comparisons). Furthermore, almost all of them occurred in the first 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.

LINEAR SWEEP DE-RELOCATION **scenario.** Last, Figure 8.3 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 each 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

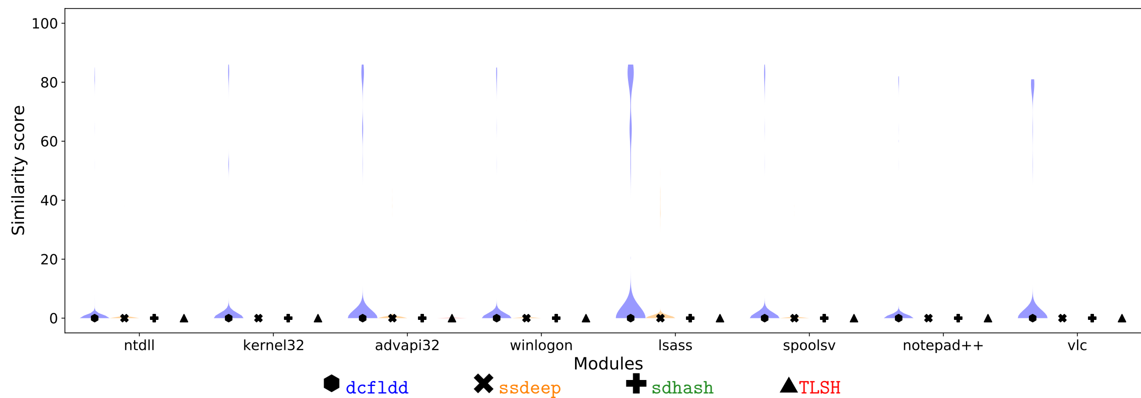


Figure 8.4: Unrelated comparison: similarity scores aggregated for all Windows OS and scenarios considered.

we are now considering all the memory dumps of 32-bit Windows OS, unlike the previous scenario in which we discarded the memory modules whose `.reloc` sections are unrecoverable. Thus, comparing these similarity scores with those shown in Figure 8.1a, 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 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.

Unrelated Comparison

In this section, we compare resident 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 8.4 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. Thus, the similarity score of the end padding bytes of pages yields a non-zero value.

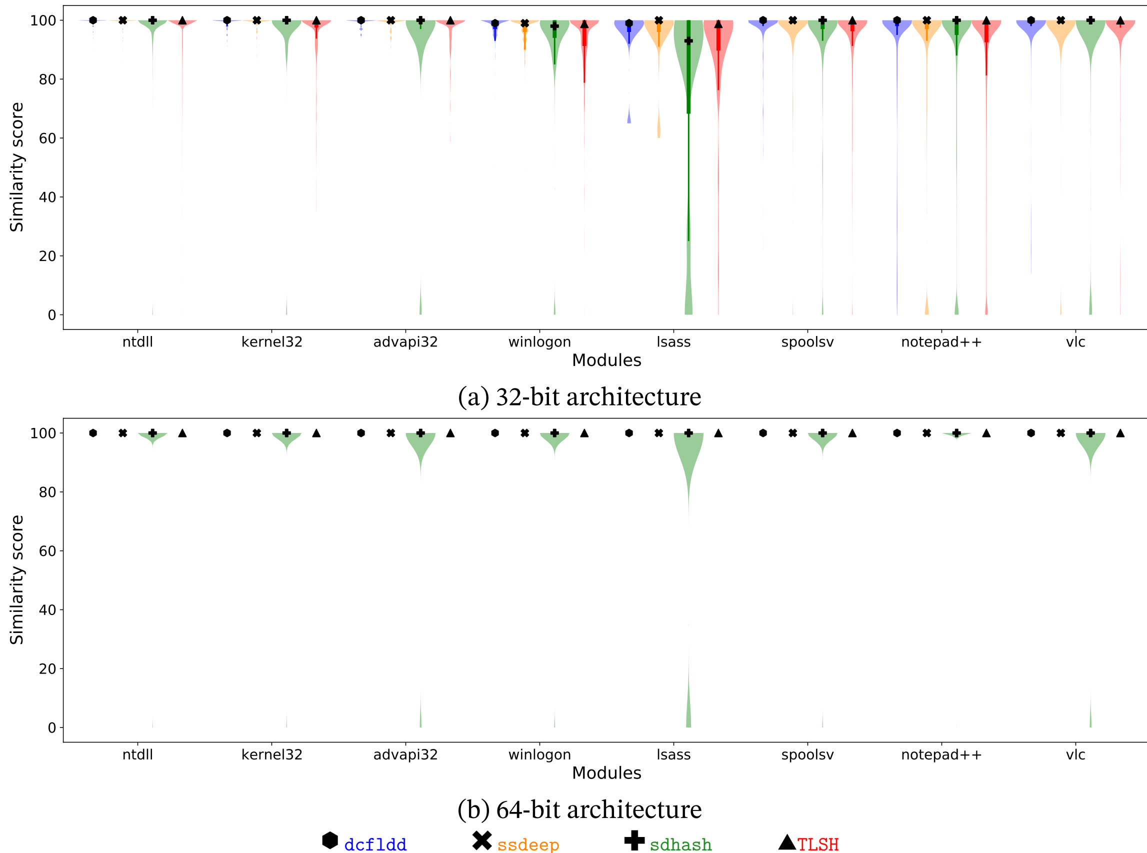


Figure 8.5: Related comparison with cross pre-processing methods.

Related Comparison with Cross Pre-processing Methods

As with the first comparison method, we now compare resident 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 this experiment, we performed 160198 and 221967 comparisons in 32-bit and 64-bit architectures, respectively.

Figure 8.5 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.

Part IV

Conclusions

Chapter 9

Conclusions and Open Problems

This final chapter presents the summary of this dissertation, detailing its contribution and describing the future work and open problems that can be investigated in the future.

9.1 Thesis Summary

The problems of automatic identification of the contents of a RAM memory of a possibly compromised system are caused by several factors, mainly related to the memory management performed by the operating system. These problems make the content of the memory dumps inaccurate and incomplete, making it difficult to detect potential malware in these dumps.

On the one hand, on-demand paging and page swapping mean that not all module pages are in memory, and therefore not in the dump either. Also, when the pages have not been used for a long period, the system swaps them to the page file. These problems have been described in more depth in Section 4.1. As we have shown, pagination behaves differently depending on the type of module. Almost 80% of the executable module pages and 20% of the shared dynamic library module pages reside on a system with typical memory workload. However, these values are drastically lowered when memory is required by the operating system, as most modules are ejected from memory and thus because unrecoverable in a memory dump.

On the other hand, the relocation to a random base address performed by the ASLR defense forces the image loader to adjust the contents of the modules based on the assigned base address. As explained in Section 4.2, these adjustments cause scattered modifications across the pages, making it difficult to check the similarity between modules with traditional techniques such as cryptographic hashes.

As a possible solution to identify memory content, we have studied the Similarity Digest Algorithms (SDA). These algorithms allow identification by approximate match rather than exact match like cryptographic hashes do. Chapters 5 and 6 of this dissertation are dedicated to studying these algorithms from a formal point of view. In Chapter 5, we have studied the most used SDA, as well as some niche ones, and created a classification scheme, proposing a

classification based on the global behavior of the algorithm, unlike previous classifications. In Chapter 6, we have discussed the limitations of the SDA, defining the possible attacks and the set of desirable properties that a robust SDA must meet.

As a conclusion of this part, the general behavior of most SDA is similar. They divide the input into features to create a feature identifier, which is typically generated using a hash function. The processed features are then aggregated or added in a storage structure, creating a similarity digest. Finally, the digests are compared to generate a similarity score. With regard to the attacks, we have found that sometimes they are possible due to just one characteristic, but other times the attack needs to chain several weak characteristics. In our proposal for a set of robust characteristics, we avoid the selection of critical features as well as weak characteristics.

Finally, Chapters 7 and 8 are devoted to ways to improve the similarity score between memory dump modules. In particular, we have proposed two preprocessing methods to find the ASLR modified bytes and normalize them by setting them to zero. Both methods are detailed in Chapter 7. The first method, GUIDED DE-RELOCATION, looks for the information of these bytes using structures of the operating system itself and the structure of the executable files themselves. This method scans memory for a particular section of the executable file for each loaded module. If found, the method uses this information to normalize the affected bytes. We have developed a second method when this information is unrecoverable, LINEAR SWEEP DE-RELOCATION, which attempts to identify the affected bytes by looking for the longest sequence of disassembled instructions. This method considers that a subsequence of bytes is affected by the relocation process when it encounters any instruction in this sequence that represents an address within the memory space of the module itself. In Chapter 8, we describe the details of the implementation of the proposed methods in a Volatility plugin, dubbed `SUM`. This tool is publicly available and released under GNU/GPLv3 license. In addition, the experiments that validate our proposals are explained and analyzed.

As a conclusion of this part and based on the experimental results, we conclude that SDA are suitable algorithms for identifying modules of memory dumps, despite the memory problems discussed in this dissertation. However, preprocessing the memory dumps is a necessary step to reduce variability in similarity results. Regarding the presented methods, the GUIDED DE-RELOCATION method has better results than the LINEAR SWEEP DE-RELOCATION method. This result was as expected, since the first method uses information that allows it to know exactly the bytes affected by the relocation process.

9.2 Detailed Contributions

This section summarizes the contributions of this dissertation:

- *Detailed analysis of paging issues in modules that reside in memory.* We have studied how the memory management affects user-mode modules on Windows systems with different memory workloads. Only 80% of the executable pages and 20% of shared library pages

reside in memory, while the workload is below 75% of the total physical memory. When the load value increases, the percentage of resident pages decreases up to 20% and 5%, respectively.

- *A tool to extract the number of resident pages.* We have developed a Volatility plugin, dubbed `residentmem`, which lets us know which pages of each module are in a memory dump. Therefore, it provides forensic analysis with information and the amount of binary data that cannot be analyzed correctly.

- *Categorization for Similarity Digest Algorithms.* Our identification of the different phases of SDA and their characteristics can allow the community to better discuss and compare existing algorithms. In addition, our classification groups the SDA by their main behavior, not only by one of their phases behavior as other proposals do.

- *Description of possible attacks.* We have described the possible attacks against the SDA. Some of them are published and we have only correlated with the set of characteristics that allow the attack to exist. Other attacks are novel and are based on the knowledge gained from our previous classification.

- *Guidelines to build a robust SDA against the attacks studied.* We also provide insights on the desirable properties to build a robust algorithm, selecting the set of characteristics that theoretically prevent tampering the similarity.

- *Two pre-processing methods that normalize the relocation performed by Windows.* We have developed two pre-processing methods to find and normalize the bytes affected by relocation. The first method, GUIDED DE-RELOCATION, relies on finding the `.reloc` section from image files. The second method, LINEAR SWEEP DE-RELOCATION, identifies the affected bytes by looking for the longest sequence of instructions and identifying the operands that point to the memory space of the module.

- *A tool to normalize the relocation effects in modules from Windows dumps.* We have developed a Volatility plugin, dubbed `Similarity Unrelocation Module`, that implements the presented pre-processing methods and allows a forensic analyzer to apply them to a memory dump, generate the digests, and compare the content with other previous calculated digests.

9.3 Future Work and Open Problems

Despite the contributions of this dissertation, there are still several open problems that are planned for future work:

- *Deeper analysis of the effects of relocation.* In this dissertation, we have found that the relocation produced by the ASLR defense negatively affects the similarity of the comparisons. We plan to do a specific study of how these modifications affect memory and relate them to the result of pre-processing methods to improve them.
- *Development of proofs of concept for the theoretical attacks against the Similarity Digest Algorithms.* Our goal is to validate our definition of potential attacks against SDA, developing proof of concept of these attacks. We hope these proof of concepts help identify new algorithm limitations, as well as improve the guidance for building a robust SDA.
- *Development of a robust SDA algorithm.* We plan to develop an SDA following the set of characteristics that we have proposed in the guidance to obtain a robust SDA algorithm.
- *Improve the recoverable content of memory dumps by considering page files.* Due to page swapping, memory content is incomplete. Therefore, we plan to consider page files during the analysis process to increase the number of pages retrieved.
- *Improve the* GUIDED DE-RELOCATION *pre-processing method using page files.* We plan to improve the scanning process of this method by considering page files so that we can retrieve the `.reloc` section of modules when their corresponding file objects are not in memory.
- *Improve the* LINEAR SWEEP DE-RELOCATION *pre-processing method using a recursive disassembly algorithm.* We plan to improve the accuracy of this method by incorporating a new phase after recognizing the PE structure. This new phase will implement a recursive disassembler that starts at true instruction addresses retrieved from the memory dump. These addresses can be retrieved via return addresses on the stack or function addresses obtained from the symbol table.

In addition to all the future work mentioned above, we aim to apply all the findings of this dissertation to other domains, such as memory forensics of mobile phones or IoT devices, as well as to extend this work to other desktop operating systems such as macOS and GNU/Linux.

Relevant Publications Related to this Dissertation

Martín-Pérez, M. and Rodríguez, R. J. (2021). Quantifying Paging on Recoverable Data from Windows User-Space Modules. Accepted for publication. To appear in *12th International Conference on Digital Forensics and Cyber Crime (ICDF2C)*. [Online; https://webdiis.unizar.es/~ricardo/files/papers/MR-ICDF2C-21.pdf].

Martín-Pérez, M., Rodríguez, R. J., and Balzarotti, D. (2021a). Pre-processing memory dumps to improve similarity score of Windows modules. Computers & Security, 101:102119.

Martín-Pérez, M., Rodríguez, R. J., and Breitinger, F. (2021b). Bringing Order to Approximate Matching: Classification and Attacks on Similarity Digest Algorithms. *Forensic Science International: Digital Investigation*, 36:301120.

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

Martín-Pérez, M. and Rodríguez, R. J. (2021). Residentmem Volatility Plugin. [Online; https://github.com/reverseame/residentmem]. Accessed on May, 20, 2020.

Bibliography

Aghaeikheirabady, M., Farshchi, S. M. R., and Shirazi, H. (2014). A new approach to malware detection by comparative analysis of data structures in a memory image. In *2014 International Congress on Technology, Communication and Knowledge (ICTCK)*, pages 1–4.

Baier, H. and Breitinger, F. (2011). Security Aspects of Piecewise Hashing in Computer Forensics. In *Proceedings of the 2011 Sixth International Conference on IT Security Incident Management and IT Forensics*, IMF '11, pages 21–36, USA. IEEE Computer Society.

Balzarotti, D., Di Pietro, R., and Villani, A. (2015). The impact of GPU-assisted malware on memory forensics: A case study. *Digital Investigation*, 14:S16 – S24. The Proceedings of the Fifteenth Annual DFRWS Conference.

Beverly, R., Garfinkel, S., and Cardwell, G. (2011). Forensic carving of network packets and associated data structures. *Digital Investigation*, 8:S78–S89. The Proceedings of the Eleventh Annual DFRWS Conference.

Block, F. and Dewald, A. (2019). Windows Memory Forensics: Detecting (Un)Intentionally Hidden Injected Code by Examining Page Table Entries. *Digital Investigation*, 29:S3–S12.

Bloom, B. H. (1970). Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426.

Bozkir, A. S., Tahillioglu, E., Aydos, M., and Kara, I. (2021). Catch them alive: A malware detection approach through memory forensics, manifold learning and computer vision. *Computers & Security*, 103:102166.

Breitinger, F., Astebøl, K. P., Baier, H., and Busch, C. (2013). mvHash-B - A New Approach for Similarity Preserving Hashing. In *2013 Seventh International Conference on IT Security Incident Management and IT Forensics*, pages 33–44.

Breitinger, F. and Baggili, I. (2014). File detection on network traffic using approximate matching. *Journal of Digital Forensics, Security and Law*, 9(2):15.

Breitinger, F. and Baier, H. (2012a). Properties of a similarity preserving hash function and their realization in sdhash. In *2012 Information Security for South Africa*, pages 1–8. IEEE.

Breitinger, F. and Baier, H. (2012b). Similarity Preserving Hashing: Eligible Properties and a New Algorithm MRSH-v2. In Rogers, M. and Seigfried-Spellar, K. C., editors, *Digital Forensics and Cyber Crime*, pages 167–182, Berlin, Heidelberg. Springer Berlin Heidelberg.

Breitinger, F., Baier, H., and Beckingham, J. (2012). Security and implementation analysis of the similarity digest sdhash. In *First International Baltic Conference on Network Security & Forensics (NeSeFo)*, pages 1–16.

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

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

Breitinger, F. and Roussev, V. (2014). Automated evaluation of approximate matching algorithms on real data. *Digital Investigation*, 11:S10–S17. Proceedings of the First Annual DFRWS Europe.

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

Breitinger, F., Stivaktakis, G., and Roussev, V. (2014c). Evaluating detection error trade-offs for bytewise approximate matching algorithms. *Digital Investigation*, 11(2):81–89.

Breitinger, F., Ziroff, G., Lange, S., and Baier, H. (2014d). Similarity Hashing Based on Levenshtein Distances. In Peterson, G. and Shenoi, S., editors, *Advances in Digital Forensics X. IFIP Advances in Information and Communication Technology, vol 433.*, pages 133–147. Springer, Berlin, Heidelberg.

Capstone (2020). Capstone – The Ultimate Disassembler. Online, http://www.capstone-engine.org/. Accessed on July 10, 2020.

Case, A., Maggio, R. D., Firoz-Ul-Amin, M., Jalalzai, M. M., Ali-Gombe, A., Sun, M., and Richard, G. G. (2020). Hooktracer: Automatic Detection and Analysis of Keystroke Loggers Using Memory Forensics. *Computers & Security*, 96:101872.

Chang, D., Ghosh, M., Sanadhya, S. K., Singh, M., and White, D. R. (2019). FbHash: A New Similarity Hashing Scheme for Digital Forensics. *Digital Investigation*, 29:S113 – S123.

Chang, D., Kr. Sanadhya, S., Singh, M., and Verma, R. (2015). A Collision Attack on Sdhash Similarity Hashing. In *Proceedings of the 10th International Conference on Systematic Approaches to Digital Forensic Engineering*, pages 36–46.

Chen, R. (2013). The source of much confusion: “backed by the system paging file”. [Online; https://devblogs.microsoft.com/oldnewthing/20130301-00/?p=5093]. Accessed on April 24, 2021.

Cichonski, P., Millar, T., Grance, T., and 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).

Cohen, M. (2017). Scanning memory with Yara. *Digital Investigation*, 20:34–43.

Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176.

Damiani, E., di Vimercati, S. D. C., Paraboschi, S., and Samarati, P. (2004). An Open Digest-based Technique for Spam Detection*. In *Proceedings of the 2004 International Workshop on Security in Parallel and Distributed Systems*, pages 559–564.

Deutsch, P. and Gailly, J.-L. (1996). Zlib compressed data format specification version 3.3. Technical report, RFC 1950, May. https://www.hjp.at/doc/rfc/rfc1950.html.

Dolan-Gavitt, B. (2007). The VAD tree: A process-eye view of physical memory. *Digital Investigation*, 4:62–64.

Dolan-Gavitt, B., Payne, B., and Lee, W. (2011). Leveraging Forensic Tools for Virtual Machine Introspection. Technical report, Georgia Institute of Technology.

Duan, Y., Fu, X., Luo, B., Wang, Z., Shi, J., and Du, X. (2015). Detective: Automatically identify and analyze malware processes in forensic scenarios via DLLs. In *2015 IEEE International Conference on Communications (ICC)*, pages 5691–5696.

FireEye (2021). M-Trends Report 2021. [Online, https://www.fireeye.com/content/dam/collateral/en/rpt-mtrends-2021.pdf.]. Accessed on September 20, 2021.

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

Gayoso Martínez, V., Hernández Álvarez, F., and Hernández Encinas, L. (2014). State of the Art in Similarity Preserving Hashing Functions. In *Proceedings of the 2014 International Conference on Security and Management (SAM’14)*, Worldcomp 2014, pages 139–145.

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

Granc, T., Chevalier, S., Scarfone, K. K., and Dang, H. (2006). Guide to Integrating Forensic Techniques into Incident Response. techreport 800-86, National Institute of Standards and Technology (NIST). Special Publication (NIST SP).

Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160.

Harbour, N. (2002). Dcfldd version 1.0. [Online; http://dcfldd.sourceforge.net/]. Accessed on October 3, 2020.

Harichandran, V. S., Breitinger, F., and Baggili, I. (2016). Bytewise Approximate Matching: the Good, the Bad, and the Unknown. *Journal of Digital Forensics, Security and Law*, 11(2).

Hay, B. and Nance, K. (2008). Forensics Examination of Volatile System Data Using Virtual Introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82.

Hintze, J. L. and Nelson, R. D. (1998). Violin Plots: A Box Plot-Density Trace Synergism. *The American Statistician*, 52(2):181–184.

Huffman, C. (2015). Chapter 4 - Process memory. In Huffman, C., editor, *Windows Performance Analysis Field Guide*, pages 93 – 127. Syngress, Boston.

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.

Jaccard, P. (1908). Nouvelles recherches sur la distribution florale. *Bull. Soc. Vaud. Sci. Nat.*, 44:223–270.

Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97. The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).

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

Lee, A. and Atkison, T. (2017). A Comparison of Fuzzy Hashes: Evaluation, Guidelines, and Future Suggestions. In *Proceedings of the SouthEast Conference*, ACM SE '17, pages 18–25, New York, NY, USA. Association for Computing Machinery.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710.

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

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

Microsoft Corporation (2019). 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 Docs (2017). Modules. [Online; https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/modules]. Accessed on February 15, 2020.

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

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

Microsoft Software Developer Network (2019a). PE Format. [Online; https://docs.microsoft.com/en-us/windows/win32/debug/pe-format]. Accessed on June 3, 2021.

Microsoft Software Developer Network (2019b). PE Format. [Online; https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#the-reloc-section-image-only]. Accessed on June 3, 2021.

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

Moia, V. H. G. and Henriques, M. A. A. (2017a). Similarity Digest Search: A Survey and Comparative Analysis of Strategies to Perform Known File Filtering Using Approximate Matching. *Security and Communication Networks*, 2017:1–17.

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

Mosli, R., Li, R., Yuan, B., and Pan, Y. (2016). Automated malware detection using artifacts in forensic memory images. In *2016 IEEE Symposium on Technologies for Homeland Security (HST)*, pages 1–6.

Nance, K., Bishop, M., and Hay, B. (2009). Investigating the Implications of Virtual Machine Introspection for Digital Forensics. In *2009 International Conference on Availability, Reliability and Security*, pages 1024–1029.

Oliver, J., Cheng, C., and Chen, Y. (2013). TLSH – A Locality Sensitive Hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13. IEEE.

Oliver, J., Forman, S., and Cheng, C. (2014a). Using randomization to attack similarity digests. In *International Conference on Applications and Techniques in Information Security*, pages 199–210. Springer.

Oliver, J., Forman, S., and Cheng, C. (2014b). Using Randomization to Attack Similarity Digests. In Batten, L., Li, G., Niu, W., and Warren, M., editors, *Applications and Techniques in Information Security*, pages 199–210, Berlin, Heidelberg. Springer Berlin Heidelberg.

Pagani, F., Dell'Amico, M., and 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*, pages 354–365.

Pagani, F., Fedorov, O., and Balzarotti, D. (2019). Introducing the Temporal Dimension to Memory Forensics. *ACM Trans. Priv. Secur.*, 22(2):9:1–9:21.

Parida, T. and Das, S. (2020). PageDumper: a mechanism to collect page table manipulation information at run-time. *International Journal of Information Security*.

Pietrek, M. (2019). An In-Depth Look into the Win32 Portable Executable File Format, Part 2. [online; https://docs.microsoft.com/en-us/archive/msdn-magazine/2002/march/inside-windows-an-in-depth-look-into-the-win32-portable-executable-file-format-part-2].

Raff, E. and Nicholas, C. (2017). An Alternative to NCD for Large Sequences, Lempel-Ziv Jaccard Distance. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1007–1015, New York, NY, USA. Association for Computing Machinery.

Rathnayaka, C. and Jamdagni, A. (2017). An Efficient Approach for Advanced Malware Analysis Using Memory Forensic Technique. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 1145–1150.

Rekall (2014). The Rekall memory forensic framework. [Online; http://www.rekall-forensic.com/]. Accessed on April 15, 2021.

Rodríguez, R. J., Martín-Pérez, M., and 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)*, pages 313–318.

Roussev, V. (2010). Data Fingerprinting with Similarity Digests. In Chow, K.-P. and Shenoi, S., editors, *Advances in Digital Forensics VI*, pages 207–226, Berlin, Heidelberg. Springer Berlin Heidelberg.

Roussev, V. (2011). An evaluation of forensic similarity hashes. *Digital Investigation*, 8:S34–S41. The Proceedings of the Eleventh Annual DFRWS Conference.

Roussev, V. (2012). Managing Terabyte-Scale Investigations with Similarity Digests. In Peterson, G. and Shenoi, S., editors, *Advances in Digital Forensics VIII*, pages 19–34, Berlin, Heidelberg. Springer Berlin Heidelberg.

Roussev, V., Chen, Y., Bourg, T., and Richard, G. G. (2006). md5bloom: Forensic filesystem hashing revisited. *Digital Investigation*, 3:82–90. The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).

Roussev, V. and Quates, C. (2013). sdhash 3.4. [Online; https://github.com/sdhash/sdhash/blob/master/sdbf/sdbf_defines.h#L58]. Accessed on 11 Mar, 2020.

Roussev, V., Richard, G. G., and Marziale, L. (2007). Multi-resolution similarity hashing. *Digital Investigation*, 4:105 – 113.

Sadowski, C. and Levin, G. (2007). SimHash: Hash-based Similarity Detection. Technical report, University of California, Santa Cruz.

Salton, G. and Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523.

Schuster, A. (2009a). filescan. https://github.com/volatilityfoundation/volatility/wiki/Command-Reference#filescan.

Schuster, A. (2009b). Linking File Objects to Processes. https://computer.forensikblog.de/en/2009/04/linking-file-objects-to-processes.html.

Schuster, A. (2009c). Scanning for File Objects. https://computer.forensikblog.de/en/2009/04/scanning-for-file-objects.html.

Singh, M., Chang, D., and Sanadhya, S. K. (2016). Security Analysis of MVhash-B Similarity Hashing. *Journal of Digital Forensics, Security and Law*, 11(2):21–34.

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

Tien, C., Liao, J., Chang, S., and Kuo, S. (2017). Memory forensics using virtual machine introspection for Malware analysis. In *2017 IEEE Conference on Dependable and Secure Computing*, pages 518–519.

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

Uroz, D. and Rodríguez, R. J. (2020). On Challenges in Verifying Trusted Executable Files in Memory Forensics. *Forensic Science International: Digital Investigation*, 32:300917.

Volatility Foundation (2020). Volatility Command Reference. [online; https://github.com/volatilityfoundation/volatility/wiki/Command-Reference]. Accessed on Oct, 2020.

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

Walters, A. and Petroni, N. (2007). Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process. In *BlackHat DC*.

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

White, A., Schatz, B., and Foo, E. (2012). Surveying the user space through user allocations. *Digital Investigation*, 9:S3–S12. The Proceedings of the Twelfth Annual DFRWS Conference.

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

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