Mobile Networks and Applications (2020) 25:248–258
https://doi.org/10.1007/s11036-019-01441-1

Disk Cluster Allocation Behavior in Windows and NTFS

Martin Karresand[1,2] · Stefan Axelsson[1,3] · Geir Olav Dyrkolbotn[1]

Published online: 19 December 2019

Abstract
The allocation algorithm of a file system has a huge impact on almost all aspects of digital forensics, because it determines where data is placed on storage media. Yet there is only basic information available on the allocation algorithm of the currently most widely spread file system; NTFS. We have therefore studied the NTFS allocation algorithm and its behavior empirically. To do that we used two virtual machines running Windows 7 and 10 on NTFS formatted fixed size virtual hard disks, the first being 64 GiB and the latter 1 TiB in size. Files of different sizes were written to disk using two writing strategies and the $Bitmap files were manipulated to emulate file system fragmentation. Our results show that files written as one large block are allocated areas of decreasing size when the files are fragmented. The decrease in size is seen not only within files, but also between them. Hence a file having smaller fragments than another file is written after the file having larger fragments. We also found that a file written as a stream gets the opposite allocation behavior, i. e. its fragments are increasing in size as the file is written. The first allocated unit of a stream written file is always very small and hence easy to identify. The results of the experiment are of importance to the digital forensics field and will help improve the efficiency of for example file carving and timestamp verification.

Keywords Digital forensics · File carving · Allocation algorithm · NTFS

1 Introduction

File carving [17, 18] and timestamps are two key concepts in digital forensics. Unfortunately both concepts are laden with complex operations and a large amount of uncertainty regarding the correctness of the results. In file carving the digital forensic investigator has to find and categorize a large amount of (fragmented) data and then put the fragments back into the original file again without the help of a file system. The same applies to timestamps, where a timeline must be constructed, often only with data from different log files and the file system meta data at hand, two sources that are easy to manipulate. In both cases a digital forensic investigator would be helped by the extra information hidden in the storage structure of data, its allocation layout.

The file carving process and the extraction of timestamps are both dependent on the behavior of the combination of file system and operating system (OS), which govern the placement of data on hard disks through an allocation algorithm. Knowledge of the layout pattern of data on disk is crucial to the forensic investigator when doing file carving. However, the actual behavior of the allocation algorithm is not known. For example the general assumption used as a basis for different forensic file carving tools is that the data to be carved is stored contiguously and is only mildly fragmented.

To improve the timestamp information used in investigations several propositions have been made to also use the behavior of the allocation algorithm as a source [25, 26]. However, we have not found any previous work that study the exact behavior of the allocation algorithm of for example New Technology File System (NTFS). There is a small number of main allocation algorithm concepts used in all modern OSs, but the exact behavior of the different

The research leading to these results has received funding from the Research Council of Norway programme IKTPLUSS, under the research and development project Ars Forensica grant agreement 248094/O70.

✉ Martin Karresand
martin@filecarving.net

1 Department of Information Security and Communication Technology, Norwegian University of Science and Technology, Gjovik, Norway

2 Division of Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance, Swedish Defence Research Agency, Linkoping, Sweden

3 School of Information Technology, Halmstad University, Halmstad, Sweden

implementations of the algorithms are not known, at least not for the closed source OS variants. Windows using NTFS is for example said to use a best fit allocation strategy, but that information is getting dated and is also based on the Linux implementation of the NTFS driver [1]. We therefore have studied the allocation behavior of two modern versions of Windows (7 and 10) in combination with NTFS to empirically reverse engineer the allocation algorithm(s) used. The data source is based on writing files of different sizes to disk using different writing strategies (writing the file as one large block at once or as a continuous stream of data). The experiment was done using two virtual machines having fixed size virtual hard disks of different sizes, one 64 GiB and one 1 TiB.

The experiment is part of the future work presented in two earlier articles [5, 11] where we develop a framework to create maps of user data placement on hard disks. The maps give the probability of finding unique user data at different Logical Block Addressing (LBA) positions in Windows NTFS partitions and can for example be used for triage, planning of hard disk investigations and to enable different areas to be prioritized in file carving processes. Knowledge of the actual behavior of the allocation algorithm of different file systems will enhance the precision of the maps.

Our work is based on an empirical evaluation of the behavior of the allocation algorithm of Windows NTFS, using the hypothesis that a best fit allocation strategy is used. Hence we do not reverse engineer any code or expose any secret information. Likewise the reported results are not detailed enough to enable someone to recreate the Microsoft Windows NTFS allocation algorithm or driver. However, the presented results are of great value to the digital forensics community and we therefore make them public as a help in the global fight against digital crime.

The rest of this paper is organized as follows: The remaining parts of Section 1 presents background on file allocation algorithms and related work. In Section 2 we describe how the experiment was implemented. Section 3 presents the results of the experiment and in Section 4 we discuss the effects and implications of our result to the research field of file carving and other relevant areas within and related to digital forensics. Section 5 concludes the work and presents ideas of future work to be done.

1.1 Background

The theory of file system construction is for example described by Silberschatz et. al [20], Stallings [21] and Tanenbaum and Bos [23]. A file system keeps track of data stored on secondary storage and is organized in different ways. However, all implementations share some common properties: the addressing of the physical storage is abstracted by the file system into logical addresses and the logical storage position of written data is determined by an allocation algorithm.

During the history of computer systems different methods of allocating disk space have been in use. The main methods are contiguous, linked and indexed allocation [20, 21, 23]. The indexed allocation strategy, where the addresses of the data blocks are held in an index separated from a file's data, is currently the most popular. The indexed allocation strategy does not suffer from external fragmentation (unallocated holes being to small to be filled with new data), but heavily used storage media can still lead to fragmentation of files and require regular defragmentation. There is also a risk of disk space being wasted when using indexed allocation, especially for small files requiring a full index meta data block to hold just a few index posts.

There are also a number of algorithms used for handling the free space that is to be populated by new files. Silberschatz et. al [20] presents three algorithms; *first fit*, *best fit* and *worst fit*, Stallings [21] mentions *nearest fit* and Tanenbaum and Bos [23] add *next fit* and *quick fit*. Together they give the following free space allocation algorithms:

First fit starts the search for free space at the beginning of the file system and has the requirement of the space being large enough to hold the entire file. If there is no space large enough to hold the entire file, a fragment is written at the found position and the search continues.

Next fit (Stallings [21, p. 544] calls this *nearest fit*) uses the same principle as *first fit* of allocating the next free space being large enough to hold the entire file, but the search is done from the current position in the file system. If there is no single free space to hold the entire file the file is fragmented and the available free spaces are used to hold the file.

Best fit uses the free space best fitting the new file, i. e. giving the smallest remaining free space. This requires all the available free spaces to be scanned before the best fit can be chosen.

Worst fit uses the free space having the worst fit to the new file. This is the opposite to best fit, i. e. giving the largest remaining free space. As for the best fit algorithm the available free spaces of the entire file system have to be scanned before the worst fit can be chosen.

Quick fit uses several lists of different sizes of free block areas. In this way a fitting area can be found very quickly, but the algorithm suffers from a complex process during deallocation when freed up areas might have to be merged. If this is not done the storage will soon fragment into a large amount of free areas too small to be usable.

These free space allocation algorithms are not specific to storage of data on disk, they are also used in for example memory allocation in Random Access Memory (RAM) [20].

Based on the information given by [3] and [15] Microsoft Windows' NTFS is using an index allocation strategy. The problem of space being wasted when using index allocation is solved in NTFS by storing the data of smaller files (up to approximately 700 bytes)[1] in the Master File Table (MFT) meta data records themselves [15].

According to Carrier [1] the best fit free space algorithm is used by Windows XP on NTFS formatted hard disks. Since the book was written in 2005 it does not cover the allocation algorithms used by Windows 7 and newer. Our previous experiments [5, 11] indicate that the actual behavior of the allocation algorithm in NTFS is not strictly best fit. The results from experiments indicate that groups of free clusters are allocated in descending order of size, which is best fit, but not always. The deviating behavior is mentioned in a Superuser Q&A [22].

When formatting an NTFS partition 12.5% of the space is by default reserved for the MFT [15]. The MFT records are 1 KiB in size and usually the size of the smallest allocatable unit (called cluster) in NTFS is 4 KiB [15]. The allocation status of every cluster in the file system is stored in the $Bitmap file, which is record number 6 in the MFT. Each bit in the $Bitmap file represents one file system cluster in ascending LBA order. If a cluster is allocated the corresponding bit in the $Bitmap file is set to 1, hence 0 represents an unallocated cluster.

Files can be written to disk in two ways; either as a stream, or the entire file as a block. In the first case the OS does not know the final size of the file and therefore cannot optimize the allocation accordingly. This often leads to file fragmentation, but the behavior is partly mitigated by the internal buffering of the OS. When the entire file is written in one piece the OS knows its size in advance and can utilize the standard allocation algorithm, which optimizes the storage. This behavior is probably more common when dealing with smaller files that easily can be held in RAM, than for large files. The specific write behavior is software dependent and might incorporate temporary writing of files to protect the data in case of a power loss or hardware failure.

1.2 Related work

We have not found any related work directly addressing the detailed behavior of the allocation algorithm in Microsoft Windows 7 and 10 running in NTFS formatted partitions. There are however some work done on the connection between timestamps, cluster allocation and file creation order.

Willassen studies the formal foundation of timestamps in his PhD thesis [26]. There he also formulates the criteria needed to use the allocation strategy of a file system for checking timestamps. He also briefly discusses the allocation strategy of NTFS and states that it is best fit in Windows XP and that a first fit allocation strategy is used within the MFT. The PhD thesis is partially based on an earlier article by Willassen [25] describing a method to use a first fit allocation strategy to detect antedating of files.

Tse [24] intends to use the allocation pattern of files to estimate the causal ordering of events, but concludes that using the actual allocation patter is complex. Tse instead defines two metrics approximating the allocation pattern causality and then tests the metrics against standard timestamps.

Minnaard [16] studies the inner workings of the Linux implementation of the File Allocation Table (FAT32) file system in an article from 2014 by looking at the source code and then uses the result to find the causal order of files from a TomTom Go. The article mentions that the Windows 7 implementation of FAT32 is more complex than the Linux variant.

2 Experimental setup

The experiment is designed to test whether Windows in combination with NTFS is using the best fit allocation strategy as indicated in the literature [1, 26]. To enable this the behavior of the allocation algorithm has to be studied in different situations regarding disk utilization, file system fragmentation, file size and partition size. We therefore manipulated the $Bitmap file of the file systems of the virtual machines used to emulate different states of file system fragmentation.

2.1 Virtual hardware

The experiments were performed on two virtual machines using VirtualBox running Windows 7 and Windows 10. The Windows 7 machine had a fixed size 64 GiB hard disk and the Windows 10 machine a 1 TiB ditto to cover both older and newer Windows OSs, as well as small and large hard disks. Each virtual machine was given a folder shared with the host. To enable us to use standard digital forensic tools (the Sleuth Kit [2]) on the virtual hard disks and their partitions they were loop mounted with read/write access rights.

The disk of the Windows 7 machine was already used since it was taken from one of our earlier experiments [5],

[1]The maximum size of an internal $Data attribute varies depending on the size of other attributes stored in the MFT record. Most sources give a maximum internal $Data attribute size of 600 to 700 bytes. Microsoft reports a 900 byte limit [15].

where 10000 file operations were performed in a pseudo-random pattern with bias towards file writing. This had given a heavily fragmented file system corresponding to an old and well used (home) computer. The fragmentation status of the file system was checked before the start of the experiment and one remaining large area of free clusters was found at the end of the partition. To even out the fragmentation pattern we decided to split that area into five smaller areas of approximately 262000 clusters each, corresponding to approximately 1 GiB per area.

The Windows 10 machine was freshly installed and the file system therefore only contained files from the installation, without any significant fragmentation. It represented an office computer using mainly network based storage, without synchronization between local and network storage.

2.2 Process description

During the experiment each virtual machine was repeatedly power cycled to ensure that all file operations were flushed to hard disk (both the virtual and the real on the host). The experiment started by copying the $Bitmap file of the powered down virtual machine to the host. Then the following steps were iterated over for each file write operation:

1. The virtual machine was powered on
2. A file signaling the power on sequence had finished was written to the shared folder
3. The file write operation was executed
4. A file signaling the file operation had finished was written to the shared folder
5. The virtual machine was powered down
6. The power down status was checked using `vboxmanage`
7. The allocation information of the newly written file was extracted using the `istat` [2] tool
8. The $Bitmap file of the virtual machine was copied to the host using `icat` [2] tool

To allow the virtual machine, as well as the host, to properly write all files to disk up to 10 second long delays were introduced between the steps 1 to 3 of each iteration. We also used files written to a shared folder to signal when a step was finished.

To cover for both writing a file as one block and writing it as a stream a Python 2.7 script was used to either write a file to an array in RAM and then to disk (block writing) or using Python's own file write operation directly writing one 512 B sector at a time (stream writing). Both types of write operations were buffered by the OS, but in the block writing case the OS got information on the file size before the file was written to disk, which it did not when the file was written as a stream. Each 512 byte block of the files were uniquely marked with a consecutive number together with a file identifier. The marking was used as a backup procedure in case we had to read the raw data from the virtual hard disks due to errors in the process. The marking was also used to verify that the `istat` tool reported the allocation pattern in the same order as the clusters were written to, which it did.

We used files of different sizes to determine if the allocation algorithm behaved differently depending on the size of the written file. In the first round of the experiment we used 4, 128, 511, 512, 513 and 1024 MiB files. The 511 and 513 MiB files originated from the first version of the Python script, where the type of writing was depending on the size of the file. If the file size was $\leq$ 512 MiB the file was written as one block and if it was larger it was written as a stream. The range of file sizes were later expanded to also include 12, 96, 384, 768 and 1536 MiB files to get a more even coverage of small and large files and the Python script was updated to enable different writing strategies regardless of file size.

2.3 Bitmap manipulation

To cover for possibly different allocation behavior depending on the available amount of storage of the file system we manipulated the $Bitmap file of the Windows NTFS virtual hard disks. The $Bitmap file of the Windows 7 machine was manipulated once and the Windows 10 $Bitmap file three times.

The Windows 7 virtual machine's main partition was heavily fragmented from the beginning, but still contained a contiguous area of approximately 5.3 GiB. This area was divided into five smaller areas, which can be seen in Table 1, all other free areas where kept in their original state. The modified layout is called *BM 7:1* throughout the text. The unmodified layout was never used due to its already heavily fragmented file system.

Directly after installation of the Windows 10 virtual machine its 1 TiB main partition contained two large unallocated areas of approximately 497 and 511 GiB respectively at the

Table 1 The unallocated areas in the original 5.3 GiB space after the *BM 7:1* manipulation

Start position	Size [cluster]
15372418	262000
15634546	262143
15896818	262144
16159090	262145
16421363	262160

Table 2 The modified areas of *BM 10:2*

Spaces	Function	Tot. [cluster]
511	int(120000/x + 27; x = [512 : −1 : 2])	711541
23	120000	2760000
512	7x + 13; x = [0 : 511]	922368
16	7999	127984
1	17	17
29	int(1800000/x + 17; x = [1 : 29])	7131462

end of the partition. This original, unmodified, layout is called *BM 10:0* throughout the text.

After the *BM 10:0* file writing operations were executed we manipulated the $Bitmap file to fragment the allocation layout. The smaller of the two large free areas was divided into 501 equally sized unallocated areas of 120000 clusters (468.75 MiB) each and the larger unallocated area was divided into 1026 unallocated areas of increasing size, from 120 clusters (480 KiB) to 123120 clusters (approximately 481 MiB) in steps of 120. The two areas together contained 123342120 (approximately 471 GiB) free clusters after the modification. All other unallocated areas on the partition were unmodified. We refer to this manipulation setting as *BM 10:1*

After testing the *BM 10:1* allocation manipulation we created a new bitmap (referred to as *BM 10:2*) file for the Windows 10 virtual hard disk where we decreased the available space even more by first restoring the virtual machine to its original state and then creating the free areas shown in Table 2 from the two large unallocated spaces.

The *BM 10:2* manipulation was meant to test the block writing allocation behavior by forcing the algorithm to chose between a few very large areas and many small. The seemingly odd values used in Table 2 were chosen to avoid creating free areas of exact multiples of 2. The total amount of free space in the manipulated area was 11715760 clusters (44.7 GiB) after the modification.

However, the *BM 10:2* $Bitmap manipulation was not strict enough to generate any significant fragmentation during the block writing. We therefore manually decreased any remaining free areas larger than 120000 consecutive clusters with a factor 10. This left a total of 3361315 clusters (12.8 GiB) of free space in the manipulated area. This last manipulation is referred to as *BM 10:3*.

3 Result

The overall result of the experiment shows that Windows 7 and 10 using NTFS formatted partitions are both using a best fit allocation algorithm. This is however not always true, as will be shown in the following sections. In the text we will refer to *lower rest of free clusters* as a term for allocation patterns where the pattern gives a lower number of remaining free clusters compared to a strict best fit pattern. Please observe that the term is only defining a local minimum, i. e. we have only looked at one alternative pattern.

3.1 Block writing

The result of the block writing operations are found to be following the best fit allocation strategy in most of the cases in Windows 10, but not for Windows 7. We also present an interesting pattern common to both versions of Windows regarding the sizes of fragments.

3.1.1 Windows 7

The result of the block writing file operations on the 64 GiB hard disk of the Windows 7 virtual machine using *BM 7:1* can be seen in Fig. 1. The file system was already partially fragmented due to 10000 random file operations executed in an earlier experiment and we fragmented the remaining large free area into five smaller ones to put an even higher burden on the allocation algorithm.

All but three block writing file operations result in between 2 and 9 fragments, which are all allocated in descending order of size within each file operation (apart for file operations 5 to 7 (fragments 14 to 16 in Fig. 1), which are small and unfragmented). None of the fragmented file operations are best fit allocations and neither are they optimized from the point of lower rest of free clusters. The last fragment in every file operation is smaller than the previous fragments in the operation, which is represented by the dips in the curve in Fig. 1. What also can be noticed

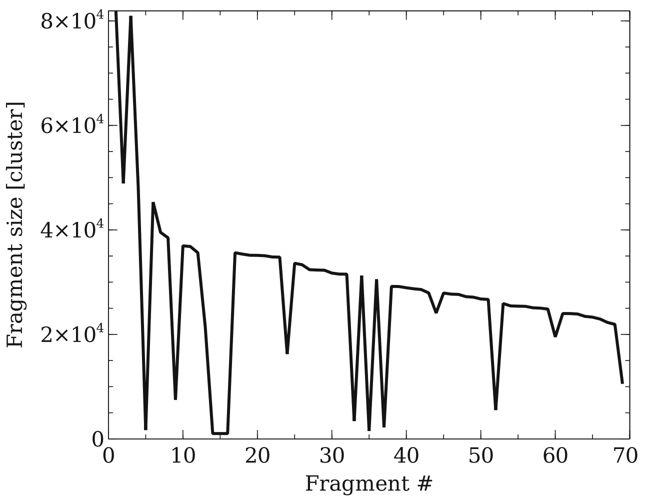

Fig. 1 The decrease in fragment size for the Windows 7 block writing experiment using the *BM 7:1* layout

is the decreasing size of the fragments even between file operations, where the size of the second last fragment in the previous file operation is larger than the starting fragment size in the following file operation. Also the free areas chosen by the algorithm are often slightly larger ($<$ 10 clusters) than the fragment and therefore free space might be wasted.

The continuously decreasing fragment sizes seen in Fig. 1 might seem to indicate a typical best fit behavior, but there were larger free areas that would have been better to use from a best fit point of view. Instead the algorithm seems to balance the sizes of the fragments to be as similar as possible, apart from the last fragment. We did not see any pattern regarding the position of the chosen fragments.

3.1.2 Windows 10

The original installation (*BM 10:0*) of Windows 10 on a 1 TiB virtual hard disk contains two large areas of free clusters at the end separated by a 512 MiB large area holding a number of files related to the System Volume Information directory. The two areas are the largest free areas on disk and over 500 times larger than the third largest area. The 11 largest free areas on the partition can be seen in Table 3.

All file operations using *BM 10:0* allocated files of one block each, which can be seen in Table 4. The first and second write operations are best fit allocations, as can be seen when comparing their sizes and positions to the available free areas in Table 3. The rest of the file write operations all allocate consecutive areas in the second largest free area, which is not a strict best fit behavior. Even the three 4 MiB file write operations (operation 3 to 5 in Table 4) are allocated in the same area and not in any of the better fitting available areas. Although there are system related and stream writing file operations executed between operations 3 to 5 there are several free areas that would fit the file sizes of these operations better than the actual allocation do.

After the file write operations in *BM 10:0* 19 block write operations are performed using the *BM 10:1* allocation layout. The larger write operations are split into $n \geq 2$ fragments, where the $n-1$ first fragments all have approximately the same size within each operation. The largest available free area is 123120 clusters in size and as can be seen in Table 5 that area is the first to be allocated in this part of the experiment.

All positions in Table 5 ending in “4544” are free areas created by us. The larger areas belonging to the “4544” series are also allocated in descending order, hence they are allocated in best fit order. Also the smaller, remaining parts, of the file write operations belonging to the “4544” series are allocated in best fit order, although some of them do not fully occupy the original free area. The unfragmented file operation 6 is also a best fit allocation due to a number of previous system file deallocations that left a free area of suitable size.

The part of the experiment using the *BM 10:2* allocation layout results in single block allocations and all of them are made in order of best fit. Thus even eight 768 MiB files all result in single block allocations. Worth noticing is the fact that all but one allocation are made within the modified area, i. e. not in any of the original free areas. The best fit allocation approach is proven by the fact that an area outside of the modified area is used in a sequence of equally sized files. That area has been available since the installation of the OS and has a size that fits in between the sizes of the modified free areas.

The execution of the file operations using the *BM 10:3* allocation layout gives files fragmented into 3, 4 and 5 fragments, as can be seen in Table 7. The allocation strategy is no longer strictly best fit. Fragment 2 in file operation 1 is not best fit, because the free area at position 264466712

Table 3 The 11 largest free areas in the original installation of Windows 10 on a 1 TiB hard disk (*BM 10:0*)

Position [cluster]	Free size [cluster]
150728	869
135305	1259
2009009	1679
2005999	3009
2010944	3742
2570502	4774
56466	14272
809984	27680
2775255	239688
3021071	131132401
134284544	134022399

Table 4 The 8 block write operations performed using the original allocation layout *BM 10:0*. The operations are presented in order of execution

File op.	Position [cluster]	Size [cluster]
1	2775255	131072
2	3148190	131072
3	3280413	1024
4	3283997	1024
5	3282700	1024
6	3291406	262144
7	3815408	32768
8	3882725	32768

Table 5 The 19 block write operations performed using the *BM 10:1* allocation layout. The operations are presented in order of execution

Op.	Frag.	Pos. [cluster]	Size [cluster]
1	0	267534544	123120
1	1	142864544	7952
2	0	267404544	123000
2	1	142994544	8072
3	0	267274544	122880
3	1	267144544	122760
3	2	152094544	16504
4	0	267014544	122640
4	1	266884544	122520
4	2	152614544	16984
5	0	137534544	3072
6	0	3553550	3072
7	0	160804544	24576
8	0	137664544	3072
9	0	160934544	24576
10	0	161064544	24576
11	0	161194544	24576
12	0	240754544	98304
13	0	240884544	98304
14	0	241014544	98304
15	0	241144544	98304
16	0	266754544	122400
16	1	266624544	122280
16	2	266494544	122160
16	3	162754544	26376
17	0	266364544	122040
17	1	266234544	121920
17	2	266104544	121800
17	3	163924544	27456
18	0	265974544	121680
18	1	265844544	121560
18	2	265714544	121440
18	3	165094544	28536
19	0	265584544	121320
19	1	265454544	121200
19	2	265324544	121080
19	3	166264544	29616

would have been better to use, as can be seen in Table 6 showing the 21 largest free areas available to file allocation 1. The second file allocation in Table 7 is strictly best fit, fragments 0 to 2 belong to the largest available free areas in that round and fragment 3 to the free area best fitting the remainder. The last file allocation, number 3 in Table 7 is not best fit at a first glance. There is a fragment of 56801 clusters that would have been better to use. However that free area would then have been combined with an area leaving 1589 free clusters. The current allocation only leaves 205 clusters unallocated and therefore has a lower rest of free clusters than a potential strict best fit allocation. If this is the actual behavior of the allocation algorithm is however not possible to deduce from only one file operation.

Table 6 The 21 largest free areas in the (*BM 10:3*) allocation layout. These are also all free areas equal to or bigger than 18017 clusters in the partition

Position [cluster]	Free size [cluster]
265866802	18017
124324183	24027
265638022	28409
124564183	30027
124804183	40027
263669912	40407
264466712	56801
265380686	60550
267499096	62144
267646484	69247
267574290	72017
267338630	81835
267252722	85731
262618164	85961
267162528	90017
267067598	94753
266967404	100017
265080492	103409
266861328	105899
125536472	107711
266748634	112517

Table 7 The 3 block write operations performed using the *BM 10:3* allocation layout. The operations are presented in order of execution

Op.	Frag.	Pos. [cluster]	Size [cluster]
1	0	266748636	112512
1	1	125536472	107708
1	2	265380688	41924
2	0	267252724	85728
2	1	267338632	81832
2	2	267574292	72012
2	3	124324184	22572
3	0	267646484	69244
3	1	267499096	62140
3	2	267438388	60531
3	3	263669912	40407
3	4	124564184	29822

3.1.3 General observations

We have not been able to see any patterns in the allocation algorithm's behavior regarding how the allocated positions are chosen. Likewise we have not been able to find an algorithm for how the fragment sizes are chosen. In most cases the fragments are somewhat smaller (a few clusters) than the free area, leaving small free areas before and/or after the fragment. Hence the the positioning and size of the fragments within the free areas are probably governed by some rules, but we have to few data to fully deduce them.

All fragmented block writing operations have one feature in common, which is the globally decreasing fragment size. The second last fragment in a file operation is always larger than the first fragment in the following file operation. The decreasing size feature is also valid within each file operation. The feature can be seen in Fig. 1 for the Windows 7 virtual machine and in Tables 4, 5 and 7 for the Windows 10 virtual machine. This pattern is valid even if there are system file or stream writing operations interleaved with the block writing operations.

3.2 Stream writing

The result of the stream writing file operations consists of 15 operations done in Windows 7 and 23 operations in Windows 10. Since the files are written as a stream the OS cannot implement a proper best fit allocation strategy and we therefore do not check for it.

3.2.1 Windows 7

There is a weak correlation between file size, the file writing order and number of fragments in the results, which can be seen in Table 8. The correlation is however very weak and seems to be increasing for small files and decreasing for larger files.

The allocation patterns of the different file operations always begin with very small fragment sizes, often there is a single cluster allocated first. The size of the fragments increases as more data is written to disk. In some cases the fragment size is doubled in each of the first 5 consecutive allocations. This pattern diminishes as even more data is written, but the size of the fragments is constantly increasing, occasionally with a large deviation either up or down. The deviations are sometimes as frequent as every second allocation. In some cases the fragment size is suddenly increased with one or two orders of magnitude, a few times even more.

3.2.2 Windows 10

As for the Windows 7 case we show the number of fragments per file and their sizes in clusters (see Table 9). Windows 10 does not show any correlation between file size, writing order and number of fragments.

Table 8 The number of fragments per file and their sizes in clusters for the Windows 7 stream writing experiment using *BM 7:1*. There is a weak correlation between the number of fragments, the writing order and the file size, which is most clearly seen for the largest files. The result is presented in order of size and time of writing

No. of frag.	File size [cluster]
26	1024
35	1024
40	1024
70	32768
44	32768
1131	131328
468	131328
195	131328
242	131328
339	262144
226	262144
1657	393216
340	393216
328	393216
206	393216

The stream writing file allocation for Windows 10 is smoother than for Windows 7, with fewer and lower deviations. The general increase of the fragment's size seen in Windows 7, as well as the small fragments at the beginning of the file operations, are present also in the Windows 10 results. The first few file fragments often reach sizes of approximately 40 clusters in two or three steps, which is faster than in Windows 7. The first stream writing file operations using the *BM 10:0* allocation layout ends with one very large allocation, especially when the files are bigger. In Table 9 this is manifested in the low number of fragments connected to some of the files containing 131072 and 262144 clusters.

3.2.3 General observations

Both Windows 7 and 10 show a general increase, although not without deviations, regarding the size of the allocated fragments. The allocation always starts with a small fragment, the maximum size of the first fragment for both versions of Windows is 5. The distribution of starting fragment sizes can be seen in Table 10.

All (100%) of the Windows 7 starting fragments are of size 1 and for Windows 10 the amount of size 1 starting fragments is 52%.

Table 9 The number of fragments per file and their sizes in clusters for the Windows 10 stream writing experiment using *BM 10:0*. No real correlation between the number of fragments, the writing order and the file size can be seen. The result is presented in order of size and time of writing

No. of frag.	File size [cluster]
7	1024
17	1024
27	1024
13	1024
48	1536
70	12288
18	32768
20	32768
174	49152
21	131072
39	131072
64	131072
43	131072
326	196608
165	196608
417	262144
1024	262144
103	262144
47	262144
23	262144
53	262144
48	262144
152	393216

4 Discussion

The main contribution of the experiment on the behavior of the NTFS allocation algorithm in Windows 7 and 10 is the result showing a globally decreasing fragment size of block writing file operations. However there are a few constraints that first must be fulfilled. First of all the file system must be fragmented, without any large areas of free, unallocated clusters and the files to be written have to be larger than the largest available free area to force the allocation algorithm to fragment them. Consequently there should not be any deallocations of areas larger than the next file to be written. If a new larger free area becomes available the decreasing trend will probably be restarted from there. Apparently stream writing interleaved with block writing is not affecting the decreasing size allocation pattern, but such files are not included in the decreasing pattern. Unfragmented files will also be excluded from the pattern.

Table 10 The accumulated amount of first fragment sizes in both Windows 7 and 10 using stream writing. All Windows 7 starting fragments are single clusters

Frag. size [cluster]	Amount [%]
1	71
2	16
3	8
4	0
5	5

The decreasing fragment size behavior can for example be used in digital forensic investigations to get a relative timestamp or the sequence of writing of a collection of block written files. By comparing the sizes of the first and second last allocated fragments of files their timestamps can be verified, or their internal age relative each other be decided.

It is also possible to separate a block written file from a stream ditto. If that knowledge is combined with knowledge on what type of writing strategy different software packages in a Windows computer use the probable source of a file can be decided. This is for example useful in triage situations where it will be enough to scan the NTFS meta data (the MFT records) to determine the main source of a file.

Also the stream writing behavior of the allocation algorithm can be of use for a digital forensic examiner. Our result shows that the size of the first fragment of a stream written file is 1 cluster in 71% of the cases (100% in the 64 GiB virtual hard disk running Windows 7) and none of the stream written files started with a fragment larger than 5 clusters. That corresponds to files between 4 and 20 KiB in size. We have earlier found the average amount of files in a standard office computer to be approximately 350000 by counting the files in 25 (office) computers running mostly Windows 7 [5, 11]. If we combine that with the official disk space requirement of 20 GiB for a standard Windows 7 to 10 installation [12–14] we get an average file size of 60 KiB. This is a theoretical lower bound, because we did not take the size of the user files included in the computer average of 350000 files into account. Hence any allocated area smaller than 60 KiB belongs to the first part of a stream written file with a high probability. This knowledge can for example be used in file carving processes to quickly find the data type of files directly from the data, without the need to trust the file name, by searching for small areas with similar data content and then extracting any magical bytes from them.

Since we have reverse engineered the allocation algorithm without access to any documentation we have no written proofs of the reasons behind the behavior we have found. We therefore can only hypothesize, but a valid reason for the block writing behavior of creating similarly sized fragments is to create an even distribution of the data over several cylinders or flash memory capsules for faster access and wear levelling.

The reason behind the stream writing behavior of starting small and increasing the fragment sizes might be two-fold. By starting from very small fragment sizes any free clusters left over when doing best fit allocation can be taken care of. The other reason is the fact that without knowledge on the actual size of the file to be written the algorithm has to estimate the final size of the file. By increasing the size of the fragments as more data is written we get a acceptable compromise between speed (less fragmentation) and a good utilization factor (small areas are not wasted). Hence the average fragment size will increase as the file size increases.

The file system allocation algorithm of Windows 7 and 10 when using NTFS should be best fit [1], but our experiment shows that it is not completely true. The block writing allocation is deviating from a strict best fit behavior in a few cases. Possible reasons for any differences in behavior are typically OS version, partition size, degree of file system utilization, file size and type of file writing behavior.

Regarding the influence of the OS we have not noticed any differences in allocation behavior. The result might have been influence by the fact that we only used two versions of Windows, but by using both the first and the latest versions of the latest generation of Windows we cover for any differences introduced in Windows 8 and 8.1.

Any differences in allocation strategy depending on the partition size is covered by the use of both a 64 GiB and a 1 TiB virtual hard disk. During the experiments we noticed small differences in behavior between the small and the large hard disks, especially in the stream writing case where the deviations from the increasing fragment size were smaller for the 1 TiB hard disk than for the 64 GiB hard disk. The most probable reason for this behavior is the larger amount of varied sizes of free areas to choose from in the larger disk.

We found that the allocation algorithm does not use a best fit strategy when allocating block written files in Windows 10 when using the unmodified *BM 10:0* allocation layout. Instead it allocates chunks of the large area of free clusters at the end of the file system, which seems reasonable from a wear levelling point of view. We did not observe the same behavior in the Windows 7 case, but there we had gotten rid of all large unallocated areas before the experiment started.

The allocation behavior during the 4 MiB file operations in Windows 10 (using the *BM 10:0* layout) where the algorithm allocates parts of one of the large free areas at the end of the partition cannot be explained by the fact that the block writing operations and the stream operations where interleaved during the experiment. Nor can it be explained by any system files being written to the free areas that would have been appropriate to use. After file operation 5 in Table 4 there still are several free areas of a few thousand clusters left that could have been used instead. The same applies to the file write operation 5 shown in Table 5. This behavior more resembles a worst fit allocation strategy than a best fit ditto.

The knowledge on the allocation behavior of Windows and NTFS gained through the experiments presented in this paper will benefit our previous work within the file carving area [4, 6–10]. There we experimented with different algorithms to detect the file type of data fragments using only the information held in the fragments themselves. Using for example the fact that the allocation algorithm has different behavior for block writing and stream writing can help identify and separate data types that are written in different ways. Also the fact that the free areas often are not fully utilized can help improve data type separation in heavily fragmented cases by explaining very small areas with different properties intertwined between larger areas of data with equal properties. Especially since it is well known that many file types contain areas of different types of data [4, 19].

5 Conclusion and future work

By writing files ranging in size between 4 MiB to 1,5 GiB to a Windows 7 virtual machine having a 64 GiB hard drive, as well as a Windows 10 virtual machine having a 1 TiB hard drive, we have found that the Windows NTFS allocation algorithm is more complex than the best fit strategy described in the literature [1]. In most of the file operations executed during our experiment the algorithm behaved as a strict best fit type, but when having access to a very large area of free clusters it started to allocate parts of that area instead of using options corresponding to a best fit allocation strategy. Looking at data from previous experiments [5, 11] we have found that having a few very large areas of free clusters at the end of a NTFS partition is the standard situation, thus the allocation strategy used by Windows together with NTFS is only best fit in special circumstances. Likewise the allocation strategy is not strictly best fit when dealing with stream written files, where the allocation algorithm is creating increasingly larger fragments as more data is written (the fragment size and the currently written size correlates). However, the fragments are allocated to the best fitting free areas, so there is a foundation of best fit behavior in the algorithm.

We have not found any literature empirically studying the inner workings of the allocation strategy used in Windows 7 and 10 partitions formatted as NTFS. Therefore already the existence of this work contributes to the digital forensics field. The result can furthermore be used to verify timestamps, rebuild files in file carving and determine the type of file on a high level by looking at how the sizes of the allocated areas increases or decreases. Block written files are allocated in decreasing order of fragment size and stream written files are allocated in increasing order of fragment size. We also found that very small allocated areas (1 to 5 clusters) belong to the start of stream written files with high probability.

As future work we will expand the experiment to be able to isolate the different parameters affecting the behavior of the allocation algorithm. We also need more data to further strengthen our results and conclusions. It would also be of great interest to include other OSs and file systems than Windows and NTFS in the experiment.

Funding Information Open access funding provided by Swedish Defence Research Agency.

References

1. Carrier B (2005) File system forensic analysis. Addison-Wesley Professional, Boston
2. Carrier B (2014) Tsk tool overview. http://wiki.sleuthkit.org/index.php?title=TSK_Tool_Overview
3. Hughes J (2009) The four stages of NTFS file growth. https://blogs.technet.microsoft.com/askcore/2009/10/16/the-four-stages-of-ntfs-file-growth/. Accessed 24-10-2018
4. Karresand M (2008) Completing the picture — fragments and back again. Licentiate thesis, Linköping Institute of Technology, Linköping University, Sweden
5. Karresand M, Axelsson S, Dyrkolbotn GO (2019) Using NTFS cluster allocation behavior to find the location of user data. Digital Investigation. 29(Supplement):S51–S60
6. Karresand M, Shahmehri N (2006) File type identification of data fragments by their binary structure. In: Proceedings from the seventh annual IEEE systems, man and cybernetics (SMC) information assurance workshop, 2006. IEEE, Piscataway, pp 140–147
7. Karresand M, Shahmehri N (2006) Oscar – file type and camera identification using the structure of binary data fragments. In: Haggerty J., Merabti M. (eds) Proceedings of the 1st conference on advances in computer security and forensics, ACSF. The School of Computing and Mathematical Sciences, John Moores University, Liverpool, pp 11–20
8. Karresand M, Shahmehri N (2006) Oscar – file type identification of binary data in disk clusters and RAM pages. In: Proceedings of IFIP international information security conference: security and privacy in dynamic environments (SEC2006), Lecture notes in computer science, pp 413–424
9. Karresand M, Shahmehri N (2007) Oscar – using byte pairs to find file type and camera make of data fragments. In: Blyth A., Sutherland I. (eds) Proceedings of the 2nd European conference on computer network defence, in conjunction with the first workshop on digital forensics and incident analysis (EC2ND 2006). Springer, Berlin, pp 85–94
10. Karresand M, Shahmehri N (2008) Reassembly of fragmented jpeg images containing restart markers. In: Proceedings - 4th annual European conference on computer network defense, EC2ND 2008, pp 25–32
11. Karresand M, Warnqvist Å, Lindahl D, Axelsson S, Dyrkolbotn GO (2019). In: Advances in Digital Forensics XIV, chap. 8 Creating a map of user data in NTFS to improve file carving, pp. 133–158. Springer International Publishing AG, Cham
12. Microsoft: System requirements (2017). https://support.microsoft.com/en-gb/help/12660/windows-8-system-requirements. Accessed 30-04-2018
13. Microsoft: Windows 10 system requirements (2017). https://support.microsoft.com/en-us/help/4028142/windows-windows-10-system-requirements. Accessed 30-04-2018
14. Microsoft: Windows 7 system requirements (2017). https://support.microsoft.com/en-us/help/10737/windows-7-system-requirements. Accessed 30-04-2018
15. Microsoft: How ntfs works (2018). https://technet.microsoft.com/pt-pt/library/cc781134(v=ws.10).aspx. Accessed 30-09-2018
16. Minnaard W (2014) The Linux FAT32 allocator and file creation order reconstruction. Digital Investigation 11(3):224–233. https://doi.org/10.1016/j.diin.2014.06.008. Special Issue: Embedded Forensics
17. Pal A, Memon N (2009) The evolution of file carving. IEEE Signal Proc Mag 26(2):59–71. https://doi.org/10.1109/MSP.2008.931081
18. Poisel R, Tjoa S (2013) A comprehensive literature review of file carving. In: 2013 International conference on availability, reliability and security, pp 475–484. https://doi.org/10.1109/ARES.2013.62
19. Roussev V, Garfinkel S (2009) File fragment classification-the case for specialized approaches. In: 2009 Fourth international IEEE workshop on systematic approaches to digital forensic engineering, pp 3–14. https://doi.org/10.1109/SADFE.2009.21
20. Silberschatz A, Galvin P, Gagne G (2012) Operating system concepts, 9th edn. Wiley, Hoboken
21. Stallings W (2012) Operating systems – internals and design principles, 7th edn. Pearson Education Inc., Upper Saddle River
22. (2017) Superuser: What block allocation algorithm does ntfs use?. https://superuser.com/questions/274855/what-block-allocation-algorithm-does-ntfs-use. Accessed 24-01-2019
23. Tanenbaum A, Bos H (2015) Modern operating systems, 4th edn. Pearson Education Inc., Upper Saddle River
24. Tse W (2011) Forensic analysis using fat32 file cluster allocation patterns. Master's thesis, University of Hong Kong
25. Willassen S (2008) Finding evidence of antedating in digital investigations. In: 2008 Third international conference on availability, reliability and security, pp 26–32. https://doi.org/10.1109/ARES.2008.149
26. Willassen S (2008) Methods for enhancement of timestamp evidence in digital investigations. Ph.D. thesis, Norwegian University of Science and Technology, Faculty of Information Technology, Mathematics and Electrical Engineering, Department of Telematics