Fast in-Place File Carving for Digital Forensics*

Xinyan Zha and Sartaj Sahni

Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611
{xzha,sahni}@cise.ufl.edu

Abstract. Scalpel, a popular open source file recovery tool, performs file carving using the Boyer-Moore string search algorithm to locate headers and footers in a disk image. We show that the time required for file carving may be reduced significantly by employing multi-pattern search algorithms such as the multipattern Boyer-Moore and Aho-Corasick algorithms as well as asynchronous disk reads and multithreading as typically supported on multicore commodity PCs. Using these methods, we are able to do in-place file carving in essentially the time it takes to read the disk whose files are being carved. Since, using our methods, the limiting factor for performance is the disk read time, there is no advantage to using accelerators such as GPUs as has been proposed by others. To further speed in-place file carving, we would need a mechanism to read disk faster.

Keywords: Digital forensics, Scalpel, Aho-Corasick, multipattern Boyer-Moore, multicore computing, asynchronous disk read.

1 Introduction

The normal way to retrieve a file from a disk is to search the disk directory, obtain the file's metadata (e.g., location on disk) from the directory, and then use this information to fetch the file from the disk. Often, even when a file has been deleted, it is possible to retrieve a file using this method as typically when a file is deleted, a delete flag is set in the disk directory and the remainder of the directory metadata associated with the deleted file unaltered. Of course, the creation of new files or changes to remaining files following a delete may make it impossible to retrieve the deleted file using the disk directory as the new files' metadata may overwrite the deleted file's metadata in the directory and changes to the remaining files may use the disk blocks previously used by the deleted file.

In file carving, we attempt to recover files from a target disk whose directory entries have been corrupted. In the extreme case the entire directory is corrupted and all files on the disk are to be recovered using no metadata. The recovery of disk files in the absence of directory metadata is done using header and footer

* This research was supported, in part, by the National Science Foundation under grants 0829916 and CNS-0963812.

X. Lai et al. (Eds.): E-Forensics 2010, LNICST 56, pp. 141–158, 2011.

information for the file types we wish to recover. Figure 1 gives the header and footer for a few popular file types. This information was obtained from the Scalpel configuration file [9]. \x[0-f][0-f] denotes a hexadecimal value while \[0-3][0-7][0-7] is an octal value. So, for example, "\x4F\123\I\sCCI" decodes to "OSI CCI". In file carving, we view a disk as being serial storage (the serialization being done by sequentializing disk blocks) and extract all disk segments that lie between a header and its corresponding footer as being candidates for the files to be recovered. For example, a disk segment that begins with the string "<html" and ends with the string "</html>" is carved into an *htm* file.

Since a file may not actually reside in a consecutive sequence of disk blocks, the recovery process employed in file carving is clearly prone to error. Nonetheless, file carving recovers disk segments delimited by a header and its corresponding footer that potentially represent a file. These recovered segments may be analyzed later using some other process to eliminate false positives. Notice that some file types may have no associated footer (e.g., *txt* files have a header specified in Figure 1 but no footer). Additionally, even when a file type has a specified header and a footer one of these may be absent in the disk because of disk corruption (for example). So, additional information (such as maximum length of file to be carved for each file type) is used in the file carving process. See [7] for a review of file carving methods.

Scalpel [9] is an improved version of the file carver Foremost [13]. At present, Scalpel is the most popular open source file carver available. Scalpel carves files in two phases. In the first phase, Scalpel searches the disk image to determine the location of headers and footers. This phase results in a database with entries such as those shown in Figure 2. This database contains the metadata (i.e., start location of file, file length, file type, etc.) for the files to be carved. Since the names of the files cannot be recovered (as these are typically stored only in the disk directory, which is presumed to be unavailable), synthetic names are assigned to the carved files in the generated metadata database.

The second phase of Scalpel uses the metadata database created in the first phase to carve files from the corrupted disk and write these carved files to a new disk. Even with maximum file length limits placed on the size of files to be recovered, a very large amount of disk space may be needed to store the carved files. For example, Richard et al. [11] reports a recovery case in which "*carving a wide range of file types for a modest 8GB target yielded over 1.1 million files, with a total size exceeding the capacity of one of our 250GB drives.*"

file type	header	footer
gif	\x47\x49\x46\x38\x37\x61	\x00\x3b
gif	\x47\x49\x46\x38\x39\x61	\x00\x3b
jpg	\xff\xd8\xff\xe0\x00\x10	\xff\xd9
htm	<html	</html>
txt	—BEGIN\040PGP	
zip	PK\x03\x04	\x3c\xac

Fig. 1. Example headers and footers in Scalpel's configuration file

As observed by Richard et al. [11], because of the very large number of false positives generated by the file carving process, file carving can be very expensive both in terms of the time taken and the amount of disk space required to store the carved files. To overcome these deficiencies of file carving, Richard et al. [11] propose in-place file carving, which essentially generates only the metadata database of Figure 2. The metadata database can be examined by an expert and many of the false positives eliminated. The remaining entries in the metadata database may be examined further to recover only desired files. Since the runtime of a file carver is typically dominated by the time for phase 2, on-line file carvers take much less time than do file carvers. Additionally, the size of even a 1 million entry metadata database is less than 60MB [11]. So, in-place carving requires less disk space as well.

Although in-place file carving is considerably faster than file carving, it still takes a large amount of time. For example, in-place file carving of an 16GB flash drive with a set of 48 rules (header and footer combinations) using the first phase of Scalpel 1.6 takes more than 30 minutes on an AMD Athlon PC equipped with a 2.6GHZ Core2Duo processor and 2GB RAM. Marziale et al. [10] have proposed the use of massive threads as supported by a GPU to improve the performance of an in-place file carver. In this paper, we demonstrate that hardware accelerators such as GPUs are of little benefit when doing an in-place file carving. Specifically, by replacing the search algorithm used in Scalpel 1.6 with a multipattern search algorithm such as the multipattern Boyer Moore [15,8,14] and Aho-Corasick [1] algorithms and doing disk reads asynchronously, the overall time for in-place file carving using Scalpel 1.6 becomes very comparable to the time taken to just read the target disk that is being carved. So, the limiting factor is disk I/O and not CPU processing. Further reduction in the time spent searching the target disk for footers and headers, as possibly attainable using a GPU, cannot possibly reduce overall time to below the time needed to just read the target disk. To get further improvement in performance, we need improvement in disk I/O.

The remainder of the paper is organized as follows. Section 2 describes the search process employed by Scalpel 1.6 to identify headers and footers in the target disk. In Sections 3 and 4, respectively, we describe the Boyer-Moore and Aho-Corasick multipattern matching algorithms. Our dual-core search strategy is described in Section 5 and our asynchronous read strategy is described in Section 6. In Section 7 we describe strategies for a multicore in-place file carver. Experimental results demonstrating the effectiveness of our methods are presented in Section 8.

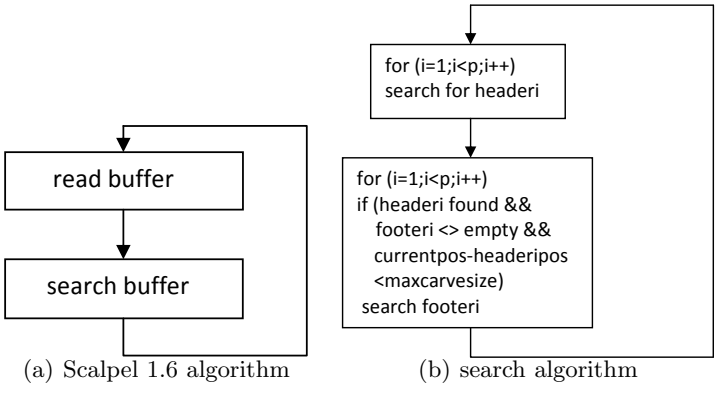
filename	start	truncated	length	image
gif/0000001.gif	27465839	NO	2746	/tmp/linux-image
gif/0000006.gif	45496392	NO	4234	/tmp/linux-image
jpg/0000047.jpg	55645747	NO	675	/tmp/linux-image
htm/0000013.htm	23123244	NO	823	/tmp/linux-image
txt/0000021.txt	34235233	NO	56	/tmp/linux-image
zip/0000008.zip	76452352	NO	1423646	/tmp/linux-image

Fig. 2. Examples of in-place file carving output

2 In-Place Carving Using Scalpel 1.6

There are essentially two tasks associated with in-place carving–(a) identify the location of specified headers and footers in the target disk and (b) pair headers and corresponding footers while respecting the additional constraints (e.g., maximum file length) specified by the user. The time required for (b) is insignificant compared to that required for (a). So, we focus on (a).

Scalpel 1.6 locates headers and footers by searching the target disk using a buffer of size 10MB. Figure 3(a) gives the high-level control flow of Scalpel 1.6. A 10MB buffer is filled from disk and then searched for headers and footers. This process is repeated until the entire disk has been searched. When the search moves from one buffer to the next, care is exercised to ensure that headers/footers that span a buffer boundary are detected. Searching within a buffer is done using the algorithm of Figure 3(b). In each buffer, we first search for headers. The search for headers is followed by a search for footers. Only non-null footers that are within the maximum carving length of an already found header are searched for.



(a) Scalpel 1.6 algorithm (b) search algorithm

Fig. 3. Control flow Scalpel 1.6

To search a buffer for an individual header of footer, Scalpel 1.6 uses the Boyer-Moore pattern matching algorithm [4], which was developed to find all occurrences of a pattern P in a string S.. This algorithm begins by positioning the first character of P at the first character of S. This results in a pairing of the first $|P|$ characters of S with characters of P. The characters in each pair are compared beginning with those in the rightmost pair. If all pairs of characters match, we have found an occurrence of P in S and P is shifted right by 1 character (or by $|P|$ if only non-overlapping matches are to be found). Otherwise, we stop at the rightmost pair (or first pair since we compare right to left) where there is a mismatch and use the *bad character function* for P to determine how

many characters to shift P right before re-examining pairs of characters from P and S for a match. More specifically, the bad character function for P gives the distance from the end of P of the last occurrence of each possible character that may appear in S. So, for example, if the characters of S are drawn from the alphabet $\{a, b, c, d\}$, the bad character function, B, for P = "abcabcd" has $B(a) = 4$, $B(b) = 3$, $B(c)= 2$, and $B(d) = 1$. In practice, many of the shifts in the bad character function of a pattern are close to the length, $|P|$, of the pattern P making the Boyer-Moore algorithm a very fast search algorithm. In fact, when the alphabet size is large, the average run time of the Boyer-Moore algorithm is $O(|S|/|P|)$. Galil [5] has proposed a variation for which the worst-case run time is $O(|S|)$. Horspool [6] proposes a simplification to the Boyer-Moore algorithm whose performance is about the same as that of the Boyer-Moore algorithm.

Even though the Boyer-Moore algorithm is a very fast way to find all occurrences of a pattern in a string, using it in our in-place carving application isn't optimal because we must use the algorithm once for each pattern (header/footer) to be searched. So, the time to search for all patterns grows linearly in the number of patterns. Locating headers and footers using the Boyer-Moore algorithm, as is done in Scalpel 1.6, takes $O(mn)$ time where m is the number of file types being searched and n is the size of the target disk. Consequently, the run time for in-place carving grows linearly with both the number of file types and the size of the target disk. Doubling either the number of file types or the disk size will double the expected run time; doubling both will quadruple the run time. However, when a multipattern search algorithm is used, the run time is $O(n)$ (both expected and worst case). That is, the time is independent of the number of file types. Whether we are searching for 20 file types or 40, the time to find the locations of all headers and footers is the same!

3 Multipattern Boyer-Moore Algorithm

Several multipattern extensions to the Boyer-Moore search algorithm have been proposed [2,15,14,8]. All of these multipattern search algorithms extend the basic bad character function employed by the Boyer-Moore algorithm to a bad character function for a set of patterns. This is done by combining the bad character functions for the individual patterns to be searched into a single bad character function for the entire set of patterns. The combined bad character function B for a set of p patterns has

$B(c) = min\{B_i(c), 1 \leq i \leq p\}$

for each character c in the alphabet. Here B_i is the bad character function for the ith pattern. The Set-wise Boyer-Moore algorithm of [14] performs multipattern matching using this combined bad function. The multipattern search algorithms of [2,15,8] employ additional techniques to speed the search further. The average run time of the algorithms of [2,15,8] is $O(|S|/minL)$, where $minL$ is the length of the shortest pattern. Baeza and Gonnet [3] extend multipattern matching to allow for don't cares and complements in patterns. This extension isn't required for our in-place file carving application.

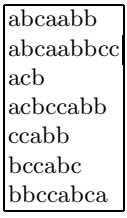


Fig. 4. An example pattern set

4 Aho-Corasick Algorithm

The Aho-Corasick algorithm [1] for multipattern matching uses a finite automaton to process the target string S. When a character of the target string is examined, one or more finite automaton moves are made. Aho and Corasick [1] propose two versions of their automaton–unoptimized and optimized–for multipattern matching. In the unoptimized version, there is a failure pointer for each state while in the optimized version, which we propose using for in-place file carving, no state has a failure pointer. In both versions, each state has success pointers and each success pointer has an associated label, which is a character from the string alphabet. Also, each state has a list of patterns/rules (from the pattern database) that are matched when that state is reached by following a success pointer. This is the list of matched rules.

In the unoptimized version, the search starts with the automaton start state designated as the current state and the first character in the text string, S, that is being searched designated as the current character. At each step, a state transition is made by examining the current character of S. If the current state has a success pointer labeled by the current character, a transition to the state pointed at by this success pointer is made and the next character of S becomes the current character. When there is no corresponding success pointer, a transition to the state pointed at by the failure pointer is made and the current character is not changed. Whenever a state is reached by following a success pointer, the rules in the list of matched rules for the reached state are output along with the position in S of the current character. This output is sufficient to identify all occurrences, in S, of all database strings. Aho and Corasick [1] have shown that when their unoptimized automaton is used, the total number of state transitions is $2n$, where n is the length of S.

In the optimized version, each state has a success pointer for every character in the alphabet and so, there is no failure pointer. Aho and Corasick [1] show how to compute the success pointer for pairs of states and characters for which there is no success pointer in the unoptimized automaton thereby transforming a unoptimized automaton into an optimized one. The number of state transitions made by an optimized automaton when searching for matches in a string of length n is n.

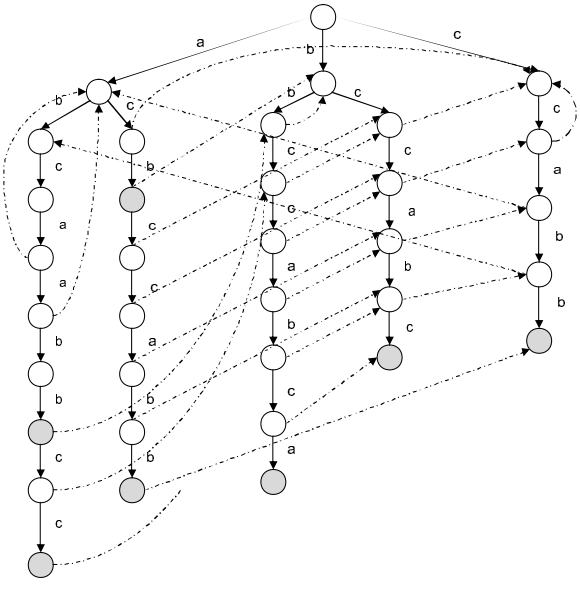


Fig. 5. Unoptimized Aho-Corasick automata for strings of Figure 4

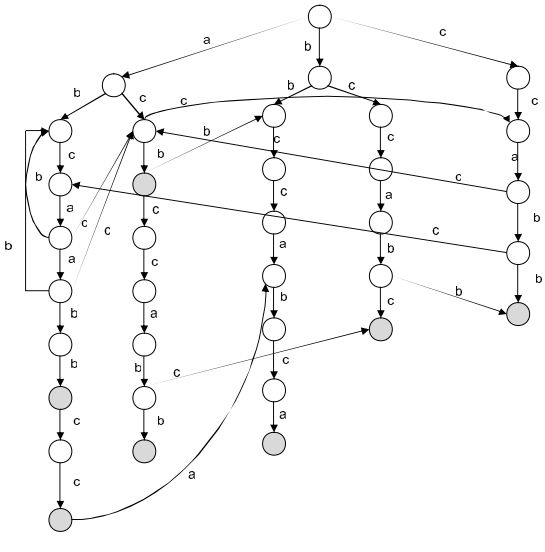


Fig. 6. Optimized Aho-Corasick automata for strings of Figure 4

Figure 4 shows an example set of patterns drawn from the 3-letter alphabet {a,b,c}. Figures 5 and 6, respectively, show the unoptimized and optimized Aho-Corasick automata for this set of patterns.

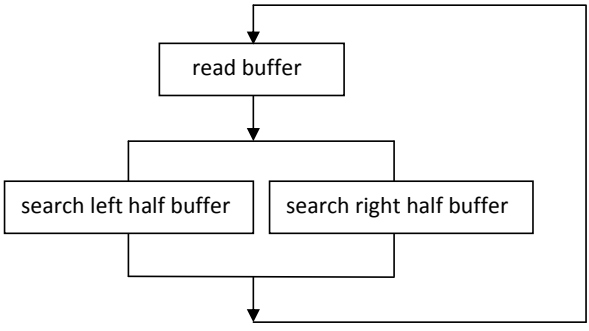


Fig. 7. Control flow for 2-threaded search

5 Multicore Searching

Contemporary commodity PCs have either a dualcore or quadcore processor. We may exploit the availability of more than one core to speed the search for headers and footers. This is done by creating as many threads as the number of cores (experiments indicate that there is no performance gain when we use more threads than the number of cores). Each thread searches a portion of the string S. So, if the number of threads is t, each thread searches a substring of size $|S|/t$ plus the length of the longest pattern minus 1. Figure 7 shows the control flow when two threads are used to do the search.

6 Asynchronous Read

Scalpel 1.6 fills its search buffer using *synchronous* (or blocking) reads of the target disk. In a synchronous read, the CPU is unable to do any computing while the read is in progress. Contemporary PCs, however, permit *asynchronous* (or non-blocking) reads of disk. When an asynchronous read is done, the CPU is able to perform computations that do not involve the data being read from disk while the disk read is in progress. When asynchronous reads are used, we need two buffers–active and inactive. In the steady state, our computer is doing an asynchronous read into the inactive buffer while simultaneously searching the active buffer. When the search of the active buffer completes, we wait for the ongoing asynchronous read to complete, swap the roles of the active and inactive buffers, initiate a new asynchronous read into the current inactive buffer, and proceed to search the current active buffer. This is stated more formally in Figure 8.

Let T_{read} be the time needed to read the target disk and let T_{search} be the time needed to search for headers and footers (exclusive of the time to read from disk). When synchronous reads are used as in Figure 3, the total time for in-place carving is approximately $T_{read} + T_{search}$ (note that the time required

```
Algorithm Asynchronous
begin
   read activebuffer
   repeat
    if there is more input
       asynchronous read inactivebuffer
    search activebuffer
    wait for asynchronous read (if any) to complete
    swap the roles of the 2 buffers
   until done
end
```

Fig. 8. In-place carving using asynchronous reads

for task (b) of in-place carving is relatively small). When asynchronous reads are used, all but the first buffer is read concurrently with the search of another buffer. So, the time for each iteration of the `repeat-until` loop is the larger of the time to read a buffer and that to search the buffer. When the buffer read time is consistently larger than the buffer search time or when the buffer search time is consistently larger than the buffer read time, the total in-place carving time using asynchronous reads is approximately $\max\{T_{read}, T_{search}\}$. Therefore, using asynchronous reads rather than synchronous reads has the potential to reduce run time by as much as 50%. The search algorithms of Sections 2 and 3, other than the Aho-Corasick algorithm, employ heuristics whose effectiveness depends on both the rule set and the actual contents of the buffer being searched. As a result, it is entirely possible that when we search one buffer, the read time exceeds the search time while when another buffer is searched, the read time exceeds the search time. So, when these search methods are used, it is possible that the in-place carving time is somewhat more than $\max\{T_{read}, T_{search}\}$.

7 Multicore in-Place Carving

In Section 5 we saw how to use multiple cores to speed the search for headers and footers. Task (a) of in-place carving, however, needs to both read data from disk and search the data that is read. There are several ways in which we can utilize the available cores to perform both these tasks. The first is to use synchronous reads followed by multicore searching as described in Section 5. We refer to this strategy as SRMS (synchronous read multicore search). Extension to a larger number of cores is straightforward.

The second possibility is to use one thread to read a buffer using a synchronous read and the second to do the search (Figure 9). We refer to this strategy as SRSS (single core read and single core search).

A third possibility is to use 4 buffers and have each thread run the asynchronous read algorithm of Figure 8 as shown in Figures 10 and 11. In Figure 10 the threads are synchronized for every pair of buffers searched while in Figure 11,

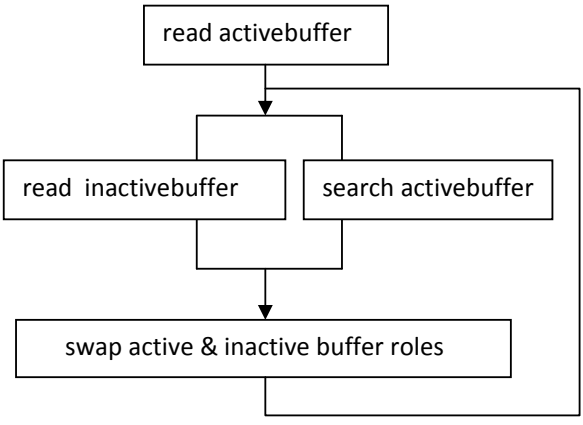


Fig. 9. Control flow for single core read and single core search (SRSS)

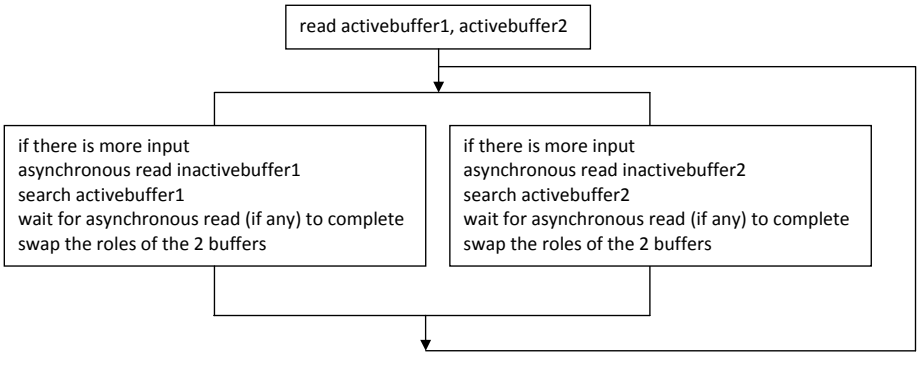


Fig. 10. Control flow for multicore asynchronous read and search (MARS1)

the synchronization is done only when the entire disk has been searched. So, using the strategy of Figure 10, each thread processes the same number of buffers (except when the number of buffers of data is odd). When the time to fill a buffer from disk consistently exceeds the time to search that buffer, the strategy of Figure 11 also processes the same number of buffers per thread. However, when the buffer fill time is less than the search time and there is sufficient variability in the time to search a buffer, it is possible, using the strategy of Figure 11, for one thread to process many more buffers than processed by the other thread. In this case, the strategy of Figure 11 will outperform that of Figure 10. For our application, the time to fill a buffer exceeds the time to search it excepts when the number of rules is large (more than 30) and the search is done using an algorithm such as Boyer Moore (as is the case in Scalpel 1.6), which is not

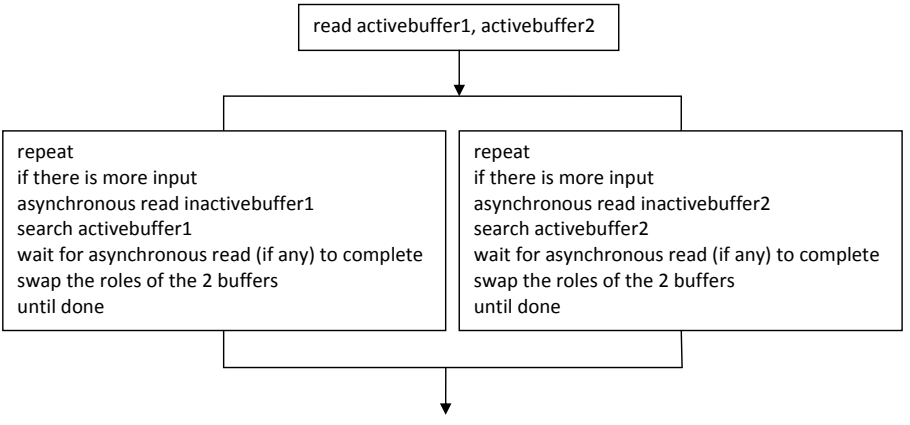


Fig. 11. Another control flow for multicore asynchronous read and search (MARS2)

designed for multipattern search. Hence, we expect both strategies to have similar performance. We refer to these strategies as MARS1 (multicore asynchronous read and search) and MARS2, respectively.

8 Experimental Results

We evaluated the strategies for in-place carving proposed in this paper using a dual processor,dual core AMD Athlon (2.6GHZ Core2Duo processor, 2GB RAM). We started with Scalpel 1.6 and shut off its second phase so that it stopped as soon as the metadata database of carved files was created. All our experiments used pattern/rule sets derived from the 48-rules in the configuration file in [12]. From this rule set we generated rule sets of smaller size by selecting the desired number of rules randomly from this set of 48 rules. We used the following search strategies: Boyer Moore as used in Scalpel 1.6 (BM); SBM-S (set-wise Boyer Moore-simple), which uses the combined bad character function given in Section 3 and the search algorithm employed in [14]; SBM-C (set-wise Boyer-Moore-complex) [15]; WuM [8]; and Aho Corasick (AC). Our experiments were designed to first measure the impact of each strategy proposed in the paper. These experiments were done using as our target disk a 16GB flash drive. All times reported in this paper are the average from repeating the experiment five times. A final experiment was conducted by coupling several strategies to obtain a new "best performance" Scalpel in-place carving program. This program is called FastScalpel. For this final experiment, we used flash drives and hard disks of varying capacity.

8.1 Run Time of Scalpel 1.6

Our first experiment analyzed the run time of in-place carving. Figure 12 shows the overall time to do an in-place carve of our 16GB flash drive as well as time

number of carving rules	6	12	24	36	48
total time	967s	1069s	1532s	1788s	1905s
disk read	833s	833s	833s	833s	833s
search	133s	232s	693s	947s	1063s
other	1s	4s	6s	8s	9s

Fig. 12. In-place carving time by Scalpel 1.6 for a 16GB falshdisk

buffer size	100KB	1MB	10MB	20MB
time	2030s	1895s	1905s	1916s

Fig. 13. In-place carving time by Scalpel 1.6 with different buffer size with 48 carving rules

spent to read the disk and that spent to search the disk for headers and footers. The time spent on other tasks (this is the difference between the total time and the sum of the read and search times) also is shown. As can be seen, the search time increases with the number of rules. However, the increase in search time isn't quite linear in the number of rules because the effectiveness of the bad character function varies from one rule to the next. For small rule sets (approximately 30 or less), the input time (time to read from disk) exceeds the search time while for larger rule sets, the search time exceeds the input time. The time spent on activities other than input and search is very small compared to that spent on search and input for all rule sets. So, to reduce overall time, we need to focus on reducing the time spent reading data from the disk and the time spent searching for headers and footers.

8.2 Buffer Size

Scalpel 1.6 spends almost all of its time reading the disk and searching for headers and footers (Figure 12). The time to read the disk is independent of the size of the processing buffer as this time depends on the disk block size used rather than the number of blocks per buffer. The search time too is relatively insensitive to the buffer size as changing the buffer size affects only the number of times the overhead of processing buffer boundaries is incurred. For large buffer sizes (say 100K and more), this overhead is negligible. Although the time spent on "other" tasks is relatively small when the buffer size is 10MB (as used in Scalpel 1.6), this time increases as the buffer size is reduced. For example, Scalpel 1.6 refreshes the progress bar following the processing of each buffer load. When the buffer size is reduced from 10MB to 100KB, this refresh is done 100 times as often. The variation in time spent on "other" activities results in a variation in the run time of Scalpel 1.6 with changing buffer size. Figure 13 shows the in-place carving time by Scalpel 1.6 with different buffer size with 48 carving rules. This variation may be virtually eliminated by altering the code for the

number of carving rules	6	12	24	36	48
BM	133s	232s	693s	947s	1063s
SBM-S	99s	108s	124s	132s	158s
SBM-C	107s	117s	142s	155s	178s
WuM	206s	205s	201s	219s	212s
AC	63s	62s	64s	65s	64s

Fig. 14. Search time for a 16GB flash drive

number of carving rules	6	12	24	36	48
SBM-S	1.34	2.15	5.59	7.17	6.73
SBM-C	1.24	1.98	4.88	6.09	5.97
WuM	0.64	1.13	3.45	4.32	5.01
AC	2.11	3.74	10.83	14.57	16.61

Fig. 15. Speedup in search time relative to Boyer-Moore

"other" components to (say) refresh the progress bar after every (say) 10 MB of data has been processed, thereby eliminating the dependency on buffer size. *So, we can get the same performance using a much smaller buffer size.*

8.3 Multipattern Matching

Figure 14 shows the time required to search our 16GB flash drive for headers and footers using different search methods. This time does not include the time needed to read from disk to buffer or the time to do other activities (see Figure 12). Figures 15 and 16 give the speedup achieved by the various multipattern search algorithms relative to the Boyer-Moore search algorithm that is used in Scalpel 1.6. As can be seen, the run time is fairly independent of the number of rules when the Aho-Corasick (AC) multipattern search algorithm is used. Although the theoretical expected run time of the remaining multipattern search algorithms (SBM-S, SBM-C, and WuM) is independent of the number of search patterns, the observed run time shows some increase with the increase in number of patterns. This is because of the variability in the effectiveness of the heuristics employed by these methods and the fact that our experiment is limited to a single rule set for each rule set size. Employing a large number of rule sets for each rule set size and searching over many different disks should result in an average time that does not increase with rule set size. The Aho-Corasick multipattern search algorithm is the clear winner for all rule set sizes. The speedup in search time when this method is used ranges from a low of 2.1 when we have 6 rules to a high of 17 when we have 48 rules.

8.4 Multicore Searching

Figure 17 gives the time to search our 16GB flash drive (exclusive of the time to read from the drive to the buffer and exclusive of the time spent on "other"

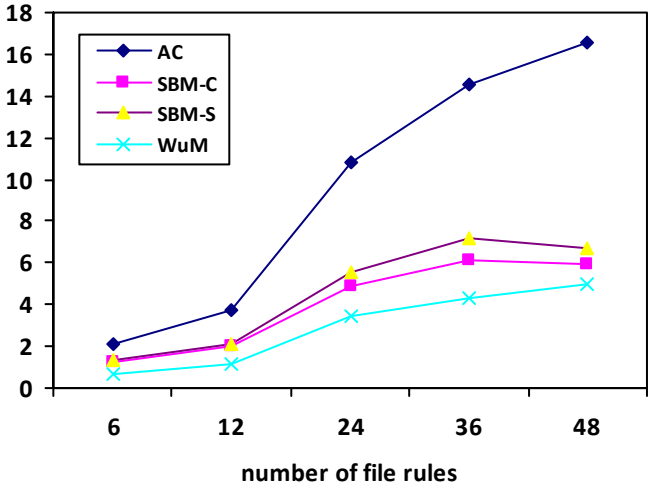


Fig. 16. Multi-Pattern Search Algorithms Speedup

Algorithms	unthreaded	2 threads	speedup
BM	693s	380s	1.82
SBM-S	124s	88s	1.41
SBM-C	142s	99s	1.43
WuM	201s	149s	1.35
AC	64s	58s	1.10

Fig. 17. Time to search using dualcore strategy with 24 rules

number of carving rules	6	12	24	36	48
BM	843s	855s	968s	966s	1100s
SBM-S	838s	837s	839s	888s	847s
SBM-C	832s	843s	837s	829s	847s
WuM	840s	841s	840s	843s	842s
AC	832s	834s	828s	833s	828s

Fig. 18. In-place carving time using Algorithm Asynchronous

activities) using 24 rules and the dualcore search strategy of Section 5. The column labeled "unthreaded" is the same as that labeled "24" in Figure 14. Although the search task is easily partitioned into 2 or more threads with little extra work required to ensure that matches that cross partition boundaries are not missed, the observed speedup from using 2 threads on a dualcore processor is quite a bit less than 2. This is due to the overhead associated with spawning and synchronizing threads. The impact of this overhead is very noticeable when the search time for each thread launch is relatively small as in the case of AC

number of carving rules	6	12	24	36	48
BM	961s	987s	1217s	1338s	1393s
SBM-S	942s	944s	953s	958s	944s
SBM-C	948s	937s	928s	935s	979s
WuM	978s	977s	975s	987s	1042s
AC	924s	925s	929s	927s	973s

Fig. 19. In-place carving time using SRMS

number of carving rules	6	12	24	36	48
BM	846	826	937s	932s	1006s
SBM-S	849s	850s	849s	844s	881s
SBM-C	852s	847s	844s	854s	845s
WuM	843s	837s	870s	843s	833s
AC	850s	852s	852s	852s	849s

Fig. 20. In-pace carving time using SRSS

number of carving rules	6	12	24	36	48
BM	909s	912s	943s	938s	1011s
SBM-S	907s	907s	908s	908s	909s
SBM-C	904s	906s	905s	907s	917s
WuM	906s	906s	907s	908s	908s
AC	904s	903s	902s	904s	904s

Fig. 21. In-place carving time using MARS2

and less noticeable when this search time is large as in the case of BM. In the case of AC, we get virtually no speedup in total search time using a dualcore search while for BM, the speedup is 1.8.

8.5 Asynchronous Read

Figure 18 gives the time taken to do an in-place carving of our 16GB disk using Algorithm Asynchronous (Figure 8). The measured time is generally quite close to the expected time of $\max\{T_{read}, T_{search}\}$. A notable exception is the time for BM with 24 rules where the in-place carving time is substantially more than $\max\{833, 693\} = 833$ (see Figure 12). This discrepancy has to do with variation in the effectiveness of the bad character heuristic used in BM from one buffer to the next as explained at the end of Section 6. Although using asynchronous reads, we are able to speedup Scalpel 1.6 by a factor of almost 2 when the number of rules is 48, this isn't sufficient to overcome the inherent inefficiency of using the Boyer-Moore search algorithm in this application over using one of the stated multipattern search algorithms.

number of carving rules	6	12	24	36	48
Scalpel 1.6(16GB)	967s	1069s	1532s	1788s	1905s
FastScalpel(16GB)	832s	834s	828s	833s	828s
Speedup(16GB)	1.16	1.28	1.85	2.15	2.31
Scalpel 1.6(32GB)	1581s	1737s	2573s	3263s	3386s
FastScalpel(32GB)	1443s	1460s	1448s	1447s	1438s
Speedup(32GB)	1.10	1.19	1.78	2.26	2.35
Scalpel 1.6(75GB)	3766s	4150s	6348s	7801s	8307s
FastScalpel(75GB)	3376s	3393s	3386s	3375s	3396s
Speedup(75GB)	1.12	1.22	1.87	2.31	2.45

Fig. 22. In-place carving time and speedup using FastScalpel and Scalpel 1.6

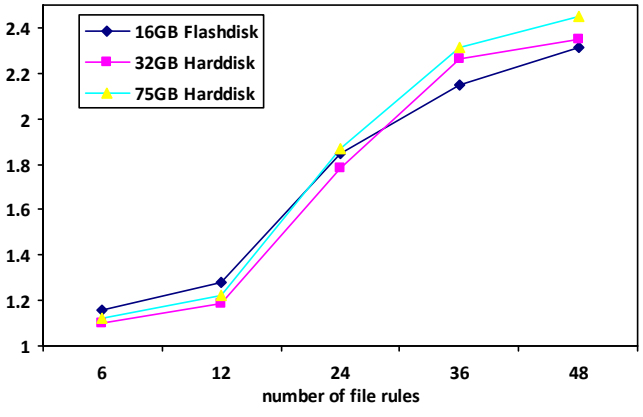


Fig. 23. Speedup of FastScalpel relative to Scalpel 1.6

8.6 Multicore in-Place Carving

Figures 19 through 21, respectively, give the time taken by the multicore carving strategies SRMS, SRSS, and MARS2 of Section 7. When the Boyer-Moore search algorithm is used, a multicore strategy results in some improvement over Algorithm Asynchronous only when we have a large number of rules (in our experiments, 24 or more rules) as when the number of rules is small, the search time is dominated by the read time and the overhead of spawning and synchronizing threads. When a multipattern search algorithm is used, no performance improvement results from the use of multiple cores. Although we experimented only with a dualcore, this conclusion applies to a large number of cores, GPUs, and other accelerators as the bottleneck is the read time from disk and not the time spent searching for headers and footers.

8.7 Scalpel 1.6 vs. FastScalpel

Based on our preliminary experiments, we modified the first phase of Scalpel 1.6 in the following way:

1. Replace the synchronous buffer reads of Scalpel 1.6 by asynchronous reads.
2. Replace the Boyer-Moore search algorithm used in Scalpel 1.6 by the Aho-Corasick multipattern search algorithm

We refer to this modified version as FastScalpel. Although FastScalpel uses the same buffer size (10MB) as used by Scalpel 1.6, we can reduce the buffer size to tens of KBs without impacting performance provided we modify the code for the "other" components of Scalpel 1.6 as described in Section 8.2. The performance of FastScalpel relative to Scalpel 1.6 was measured using a variety of target disks. Figure 22 gives the measured in-pace carving time as well as the speedup achieved by FastScalpel relative to Scalpel 1.6. Figure 23 plots the measured speedup. The 16GB disk used in these experiments is a flash disk while the 32GB and 75GB disks are hard drives. While speedup increases as we increase the size of the rule set, the speedup is relatively independent of the disk size and type. The speedup ranged from about 1.1 when the rule set size is 6 to about 2.4 when the rule set size is 48. For larger rule sets, we expect even greater speedup. Since the total time taken by FastScalpel is approximately equal to the time to read the disk being carved, further speedup is possible only by reducing the time to read the disk. This would require a higher bandwidth between the disk and buffer.

9 Conclusions

We have analyzed the performance of the popular file-carving software Scalpel 1.6 and determined that this software spend almost all of its time reading from disk and searching for headers and footers. The time spent on the latter activity may be drastically reduced (by a factor of 17 when we have 48 rules) by replacing Scalpel's current search algorithm (Boyer Moore) by the Aho-Corasick algorithm. Further, by using asynchronous disk reads, we can fully mask the search time by the read time and do in-place carving in essentially the time it takes to read the target disk. FastScalpel is an enhanced version of Scalpel 1.6 that uses asynchronous reads and the Aho-Corasick multipattern search algorithm. FastScalpel achieves a speedup of about 2.4 over Scalpel 1.6 with rule sets of size 48. Larger rule sets will result in a larger speedup. Further, our analysis and experiments show that the time to do in-place carving cannot be reduced through the use of multicores and GPUs as suggested in [11]. This is because the bottleneck is disk read and not header and footer search. The use of multicores, GPUs, and other accelerators can reduce only the search time. To improve the performance of in-place carving beyond that achieved by FastScalpel requires a reduction in the disk read time.

References

1. Aho, A., Corasick, M.: Efficient string matching: An aid to bibliographic search. CACM 18(6), 333–340 (1975)
2. Baeza-Yates, R.: Improved string searching. Software-Practice and Experience 19, 257–271 (1989)

3. Baeza-Yates, R., Gonnet, G.: A new approach to text searching. CACM 35(10), 74–82 (1992)
4. Boyer, R., Moore, J.: A fast string searching algorithm. CACM 20(10), 262–272 (1977)
5. Galil, Z.: On improving the worst case running time of Boyer-Moore string matching algorithm. In: 5th Colloquia on Automata, Languages and Programming. EATCS (1978)
6. Horspool, N.: Practical fast searching in strings. Software-Practice and Experience 10 (1980)
7. Pal, A., Memon, N.: The evolution of file carving. IEEE Signal Processing Magazine, 59–72 (2009)
8. Wu, S., Manber, U.: Agrep–a fast algorithm for multi-pattern searching, Technical Report, Department of Computer Science, University of Arizona (1994)
9. Richard III, G., Roussev, V.: Scalpel: A Frugal, High Performance FIle Carver. In: Digital Forensics Research Workshop (2005)
10. Marziale, L., Richard III, G., Roussev, V.: Massive Threading: Using GPUs to increase the performance of digit forensics tools. Science Direct (2007)
11. Richard III, G., Roussev, V., Marziale, L.: In-Place File Carving. Science Direct (2007)
12. `http://www.digitalforensicssolutions.com/Scalpel/`
13. `http://foremost.sourceforge.net/`
14. Fisk, M., Varghese, G.: Applying Fast String Matching to Intrusion Detection. Los Alamos National Lab NM (2002)
15. Commentz-Walter, B.: A String Matching Algorithm Fast on the Average. In: Maurer, H.A. (ed.) ICALP 1979. LNCS, vol. 71, pp. 118–132. Springer, Heidelberg (1979)