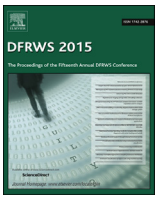
Digital Investigation 14 (2015) S95–S105

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin



DFRWS 2015 USA

Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb

Simson L. Garfinkel [a, *], Michael McCarrin [b]

[a] National Institute of Standards and Technology, USA
[b] Naval Postgraduate School, USA

Keywords:
Hash-based carving
Hashdb
bulk_extractor
Sector hashing
Similarity

A B S T R A C T

Hash-based carving is a technique for detecting the presence of specific "target files" on digital media by evaluating the hashes of individual data blocks, rather than the hashes of entire files. Unlike whole-file hashing, hash-based carving can identify files that are fragmented, files that are incomplete, or files that have been partially modified. Previous efforts at hash-based carving have looked for evidence of a single file or a few files. We attempt hash-based carving with a target file database of roughly a million files and discover an unexpectedly high false identification rate resulting from common data structures in Microsoft Office documents and multimedia files. We call such blocks "non-probative blocks." We present the HASH-SETS algorithm that can determine the presence of files, and the HASH-RUNS algorithm that can reassemble files using a database of file block hashes. Both algorithms address the problem of non-probative blocks and provide results that can be used by analysts looking for target data on searched media. We demonstrate our technique using the *bulk_extractor* forensic tool, the *hashdb* hash database, and an algorithm implementation written in Python.

Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Introduction

It is common for forensic practitioners to use databases of cryptographic hashes to search for known files. For example, some law enforcement organizations maintain databases of hash values of illegal images and videos. When media is obtained in a case, every file is cryptographically hashed and those hashes are compared to the hash database. Matches indicate the presence of a target file.

"Hash-based carving" is an alternative approach that relies on comparing hashes of physical sectors of the media to a database of hashes created by hashing every block of the target files (Collange et al., 2009b). One use-case is searching for child pornography: a block-hash database developed from a corpus of objectionable content should allow investigators to readily detect fragments of movies or still images on a storage device, even if the files have been deleted and partially overwritten. Sector hashing should also identify files that have been slightly modified—for example, files that have had a few bytes of random data appended for the explicit purpose of defeating file hashing (as may be done by an anti-forensics tool). Sector hashing can also be combined with random sampling to statistically sample the searched media, producing a high probability of finding target data within a relatively short amount of time. Finally, hash-based carving should also be able to find sectors of files in virtual memory swap files.

Although there has been some interest in hash-based carving in recent years, the technique is not widely used, and we are aware of no published algorithm describing how to assemble files from a database of sectors and sector hashes.

* Corresponding author.
E-mail addresses: simsong@acm.org (S.L. Garfinkel), mrmccarr@nps.edu (M. McCarrin).

http://dx.doi.org/10.1016/j.diin.2015.05.001
1742-2876/Published by Elsevier Ltd on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Contributions

In this paper we present our real-world experience attempting to use hash-based carving on a non-trivial problem. We build two block-hash databases, one for a small corpus of 78 target files, and one for a larger corpus of approximately 1 million document files (Garfinkel et al., 2009). We use the small database to demonstrate that individual target files can be easily identified on searched media; we use the large database to explore issues that arise when searching for files with a large target database.

Until now, the primary difficulty of implementing hash-based carving was thought to be the substantial demands that the approach places on the block hash database: since a 1 TB hard drive contains 2 billion sectors, processing such a drive in 5 h requires a database that can perform roughly 100,000 hash lookups per second. Conventional databases cannot provide such performance: a custom database written specifically for hash-based carving is required. We developed *hashdb* for this purpose (Allen, 2014) and used it to implement HASH-SETS, a simple algorithm that can determine the presence of target files on searched media.

With database in hand, we discovered that a significant hurdle to hash-based carving was resolving chance matches that occur between individuals blocks on the searched media and individual blocks in the database—matches that happened not because the target files were once present on the searched media, but because identical 4KiB blocks happen to be present in many different files.

Foster (2012) identified the existence of such "common blocks" and proposed identifying and ignoring them based on the fact that they appeared multiple times in a single database. We found that, to the contrary, even a database with a million files having many blocks that appeared to be "singleton," or unique, nevertheless contained many chance matches to other files on the searched media. *If such blocks are not explicitly anticipated, an examiner may incorrectly believe that unique blocks from a target file are present on a piece of searched media, falsely implying that the target file was once present, when in fact the blocks are not unique but merely uncommon.*

We therefore present a new approach for classifying single blocks that combines frequency and content to determine which blocks are useful for identifying the presence of target files and which are not. We use this approach in a matching algorithm for hash-based carving called HASH-RUNS that reports the location of specific consecutive disk sectors on searched media that can be unambiguously mapped to a specific target file.

Prior work

As part of his solution to the DFRWS 2006 Carving Challenge, Garfinkel introduced the technique of hash-based carving, calling it "the MD5 trick." Garfinkel (2006) extracted text from the carving challenge and used it to identify the original target documents on the Internet. He then block-hashed the target files, sector-hashed the Carving Challenge, and manually matched the two sets to identify the location of the target files in the Challenge. Three years later, Garfinkel (2009) released *frag_find*, a tool that automated the process. A more complete description of the tool appears in Garfinkel et al. (2010).

Dandass et al. (2008) performed a survey of disk sector hashes in 2008 and found no wild collisions for the MD5 or SHA1 algorithms in a sample of 528 million sectors extracted from 433,000 files. In 2009 Collange et al. introduced the term "hash-based carving" in a pair of publications (Collange et al., 2009a, b) that explored the use of GPUs to speed the hashing load, but did not address the question of what kind of database would be required to look up hashes produced at such a rate, and did not discuss how to match files given the fact that the same block hashes appear in many different files.

Key (2013) developed the File Block Hash Map Analysis (FBHMA) EnScript, a tool that both creates a hash-map of file blocks from a master file list and searches selected areas of a target drive for the blocks. Similar to *frag_find*, FBHMA can only search for a few files at one time.

Garfinkel et al. (2009) created the GOVDOCS corpus for the purpose of enabling research on file-based digital forensics. The corpus consists of approximately 1 million files downloaded from US Government web servers. Garfinkel et al. (2009) also created the M57-Patents scenario, a collection of disk images, memory dumps, and network packet captures from a fictional company called M57. The scenario consists of three interwoven crimes. We make extensive use of both corpora in this paper.

Garfinkel et al. (2010) discussed using distinct blocks to detect the presence of known files on target media, but presented no algorithms for doing so.

Foster (2012) examined 50 randomly chosen blocks that were shared between different files in the GOVDOCS corpus and attempted to determine the reason for their repetition.

Young et al. (2012) expanded the idea of using a database of distinct block hashes of target files to detect the presence of those files on media being searched. The article provided an overview of the basic idea and timing of a variety of database options, as well as counts regarding the prevalence of distinct hashes in the GOVDOCS corpus, and discussed the initial implementation of the *hashdb* hash database.

Taguchi (2013) examined coverage/time trade-offs for different sample sizes when using random sampling and sector hashing for drive triage. He concluded that a 64KiB read size was optimal in a wide variety of circumstances.

Background: hash-based carving

Terminology

We use the phrase "hash-based carving" to describe the process of recognizing a *target file* on a piece of *searched media* by hashing same-sized blocks of data from both the file and the media and looking for hash matches.

The phrases "hash" and "hash value" refer to the output of a hash algorithm. In our implementation we use the MD5 hash algorithm because of its speed; the fact that MD5 is not collision resistant (Wang and Yu, 2005) is not relevant here, as we are using MD5 to search for known

content, rather than to screen out content that is to be ignored.

A "file block" is a block of data from the target file that is hashed. Our implementation uses 4KiB blocks. The resulting hashes are stored in a database of file block hashes.

A "sector" is the minimum allocation unit of the searched media. Many modern hard drives use 4KiB sectors internally but present 512-byte sectors to the operating system for backwards compatibility. Likewise, modern file systems generally do not allocate single 512-byte sectors, but instead allocate data in blocks of 4KiB or larger. (Microsoft allocates data in clusters that are 4KiB or larger for all NTFS file systems since Windows NT 4.0 and for all FAT file systems on volumes larger than 256 MB (Microsoft Corporation, 2014).) We recommend using 4KiB sectors if they are directly supported by the drive. Otherwise, we combine 8 adjacent 512B sectors into a single 4096-byte super-sector for hashing; this block can be thought of as corresponding to a Windows NTFS disk cluster.

Finally, we use the phrase "probative" to describe data that conveys a high probability that an entire file (or a file based on the entire target file) was once present. Garfinkel et al. (2010) called such blocks "distinct." We changed terminology after discovering that many blocks that appear to be distinct in one dataset are not distinct when a larger file corpus is considered.

Scenarios for matches

The basic idea of hash-based carving is thus to compare the hashes of 4KiB sector clusters from a storage device ("search hashes") with the hashes of every 4KiB block from a file block hash database and identify files based on hashes that the two sets have in common.

Hash matches result from a variety of scenarios, including:

- A copy of an intact target file is present on the searched media. Because hash-based carving is file system agnostic, it makes no difference if the file is allocated, deleted, or in free space.
- A copy of the target file might have been placed on the searched media at some time in the past and later deleted and partially overwritten. In this case, there may be small fragments of a target file on searched media that are probative.
- A file that has many sectors in common with the target file may be on the searched media. In this case, hash-based carving will identify the blocks shared between the two similar files.
- A target file may be embedded in a larger carrying file, provided that the file is embedded on an even sector boundary. (Microsoft's ".doc" format embeds objects such as JPEG files on 512B boundaries, but the ".docx" format does not.)

Ideally, the hash-based carving algorithm should address all of these cases simultaneously.

A hash-based carving process

We approach hash-based carving as a four-step process:

1. DATABASE BUILDING: Create a database of file block hashes from the *target files*.
2. MEDIA SCANNING: Scan the *searched media* by hashing 4KiB of sectors and searching for those hashes in the database. This produces a set of hash values that can be matched to target files.
3. CANDIDATE SELECTION: Some sector hashes map to a single file, while others map to many possible files in the database. This step determines the set of target files that are likely to be present on the searched media based on the hashes observed.
4. TARGET ASSEMBLY: For each identified candidate, attempt to identify runs of matching blocks on the searched media and map these back to the corresponding target files.

Common blocks

A significant complication arises from the fact that the same 4KiB block may be present in many different target files. Foster (2012) called such blocks "common blocks." The most common block is the block of all NULLs, which is used to initialize blank media and is also found in many document and database files. The NULL block thus poses a special challenge for hash-based carving and must be specially handled, since building a list of every NULL sector on a drive would result in significant inefficiencies and possibly memory exhaustion.

A second common block pattern identified by Foster is a block of monotonically increasing 32-bit numbers. For example, Fig. 1 shows an excerpt from a Microsoft Excel file that is part of the file's Sector Allocation Table (SAT) data structure, defined by the Microsoft Office Compound File Binary Format. Any Microsoft Office file that contains an embedded 1 MB object (for example, a JPEG), will have 8KiB of data devoted to such a pattern, with the initial value depending on the location of the embedded file. The result is a low probability of a match between the SAT structures of any two specific Microsoft Office files, but a high chance that there will be a few matches between two large collections of Office files.

The existence of such common blocks complicates hash-based carving in two ways. First, because these blocks match multiple files, they cannot be used for Candidate Selection: finding a block that appears in a hundred files should not be taken as evidence that any of those hundred files are present. A second problem is that the larger we make the database, the more common blocks we discover.

```
8102 0000 8202 0000 8302 0000 8402 0000
8502 0000 8602 0000 8702 0000 8802 0000
8902 0000 8a02 0000 8b02 0000 8c02 0000
8d02 0000 8e02 0000 8f02 0000 9002 0000
```

Fig. 1. 64 bytes from the file 007533.xls shows the "ramp" structure of the Microsoft Office Sector Allocation Table.

We need an approach for recognizing common blocks before we even encounter a collision because it is simply not possible to collect and enumerate all such blocks in advance.

Sector size and alignment issues

One of the clear advantages of using 4KiB blocks over 512B blocks is that hashes of 4KiB blocks represent eight times as much data. This is especially important for hash-based carving, as it is critical to hold the entire database in RAM to support the high-speed access required.

The problem with using a 4KiB block size is file system alignment. The hashed sectors must be aligned with the file system allocation blocks, so the sector hashes will align with the file block hashes. This alignment is achieved by aligning the sector hashes with the start of the file system.

In some cases it is not possible to determine the start of the file system. This happens if the partition table is corrupted, or if there is a previous file system that was created with a different starting point.

If the partition offset is not known, or if examiner wishes to account for the possibility that there may have been a previous partitioning scheme, our solution is to hash overlapping blocks with a 4KiB sliding window over the entire drive, moving the window one sector (512B) at a time (Fig. 2). This results in eight distinct sets of 4KiB sector hashes, one where every group of 8 hashed sectors has a starting sector number of (mod 8) = 0, one where every group has a start of (mod 8) = 1, and so on. Because all of the 4KiB blocks from the same file system will necessarily have the same sector alignment, each alignment set can be processed independently.

Generating and searching overlapping hashes may seem to create needless work, since the result of calculating overlapping hashes is that the 4KiB block size requires the same number of hash operations and database lookups as the 512B block size. However, hashing every 512B need only be performed on drives with a 512B sector size, and only if the examiner is unsure of the partition start, or if there is a chance that the drive was previously partitioned with a different partitioning scheme. Moreover, even with overlapping sector hashes, the hash database is still an eighth the size when using 4KiB blocks vice 512B blocks, because only the disk sectors need to be hashed with an overlapping window.

Experimental setup

In this section we discuss our setup used to develop and test hash-based carving algorithms.

One of the fictional crimes in the M57-Patents datasets involves an employee named Jo who is collecting photographs of cats. The photographs come from the "Monterey Kitty" dataset, a set of 82 JPEG files, 2 QuickTime files, and 4 MPEG4 files (201 MB in total) recorded in Monterey CA. This dataset was used as a surrogate for child pornography. In the scenario, the employee's computer was decommissioned on 2009-11-20 by the IT coordinator and replaced with another computer. The remainder of this discussion focuses on the disk image *jo-2009-11-20-oldComputer*, which we will call *oldComputer* for brevity.

We performed sector hashing of the *oldComputer* image using *bulk_extractor* (Garfinkel, 2013), an open source digital forensics tool, and *hashdb*, a special-purpose database for storing cryptographic hashes (Allen, 2014). Included with *bulk_extractor* is a scanner called scan_hashdb that can *import* block hashes into a new database or can *scan* sector hashes against a pre-existing hash database. We use these features for the first two steps of hash-based carving, described in the next two subsections.

DATABASE BUILDING: creating the target hashdb

Using *bulk_extractor*, we created a *hashdb* database containing 4KiB block hashes corresponding to each 4KiB block of the Monterey Kitty files[1] and renamed the output database as *kitty.hdb*. The *hashdb* "size" command reports that the database has 50,206 hashes from 88 different files. The *hashdb* "histogram" command reports that all of these hashes are "distinct"—that is, there are no 4KiB blocks in the input files that have the same content.

We also used *bulk_extractor* to create a larger *hashdb* containing block hashes from the GOVDOCS corpus. The resulting *hashdb* contained 119,687,300 hashes from 909,815 files. (We did not hash files smaller than 4KiB). Of these, 117,213,026 hashes appeared only once in the dataset, 514,238 appeared twice, 60,317 appeared three times, and so on. At the other end of the distribution there was one hash that was present in 11,434 different locations. (Our software skips over the block containing all NULLs.) We call this database *govdocs.hdb*.

We added the two databases together to create a third 4KiB block hash database called *kitty + govdocs*.

MEDIA SCANNING: finding instances of known content on the searched media

Next, we used the *kitty + govdocs* database to search the M57 disk image using the *bulk_extractor* hashdb scanner.[2] This command took an average of 116 s on our 64-core reference system to scan the 13 GB disk image. *bulk_extractor* breaks the disk image into 16MiB "pages" and only processes pages that are not blank. Each page is broken into 32,768 overlapping 4KiB blocks, each block is hashed with the MD5 algorithm and the resulting hash is used to query the block hash database. (Bytes at the end of a page are joined with the bytes from the beginning of the next by *bulk_extractor*'s "margin" system.) In all, 394 pages were scanned, for a total of 6.3 GB, which translates to roughly 12.9 million sector hashes. The computation of the hashes is multi-threaded, but the version of *hashdb* that we used was single-threaded. The database was therefore performing over 111K lookups/sec.

The *bulk_extractor* program reported being I/O bound—a predictable result of running on a system with 64

[1] `bulk_extractor -S hashdb_mode=import -E hashdb -o out-kitty -R 2009-m57-patents/KittyMaterial/.`

[2] `bulk_extractor -S hashdb_mode=scan -S hashdb_scan_path_or_socket=kitty.hdb -E hashdb -o out-kitty2 jo-2009-11-20-oldComputer.E01.`

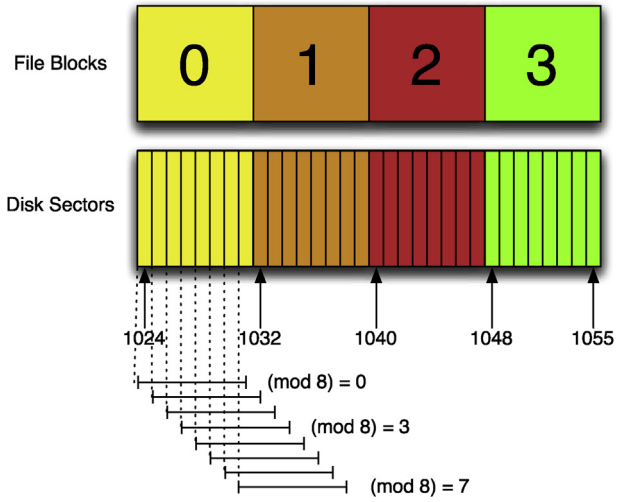


Fig. 2. 4096-byte file blocks align with groups of eight 512-byte disk sectors. By reading many sectors at once and chunking them into 4KiB runs with an 8-sector sliding window, it is possible to account for different file system offsets. A file system that starts on sector 63 will have valid data in runs that have a starting sector number of (mod 8) = 7, while a file system that starts on sector 2048 will have valid data in those blocks that have an offset of (mod 8) = 0.

cores. If the program had been CPU bound, performance could have been improved by telling the program to hash non-overlapping 4KiB blocks,[3] although the (mod 8) offset of the file system's partition would have needed to be known in advance.[4] We did this as an experiment and saw the running time decrease to an average of 103 s.

Matching sector hashes are reported in *bulk_extractor*'s *identified_blocks.txt* file. In this run 33,847 matches were found. Each hash may match one or more target files, as indicated by the "`count`" value. An excerpt of the file appears in Fig. 3.

Lines 1–3 indicate that the same 4KiB block is found at three overlapping locations that are 512B apart. Examining the media we found that the sectors consist of a repeating pattern of "ffff ff00" for 5120 bytes. The "count":39 notation states that this 4KiB block appears 39 times in the hash database; of these 39 times, one is from the file `172023.doc`, one is from the file `175907.doc`, and the remaining 37 are from the file `395714.doc`. The contents of the "flags" field indicates whether the 4KiB block matches one or more of the three rules we used to suppress non-probative blocks (in this case, the "histogram" rule). Section 4.1 describes these rules in detail.

Lines 4–6 are three consecutive 4KiB blocks on the drive that correspond to Monterey Kitty file `DSC00051.JPG`. Line 4 satisfies the histogram rule, explained below.

Line 7 is a block that matches two different files and is flagged by the histogram and whitespace rules.

Matched hashes are used in both the "candidate selection" and "target assembly" phases, first to identify target files that may be on the searched media, then to attempt to reassemble them. These steps are discussed in the following sections.

[3] To avoid hashing overlapping blocks, use the `-S hashdb_scan_sector_size=4096` option.

[4] In this example, the (mod 8) offset is 7, so the flag `-Y 3584` would need to be added.

Identifying non-probative blocks to improve candidate selection

Our initial attempt at hash-based carving did not include an explicit candidate selection step. Instead, we attempted to reassemble any file containing a block with a hash that matched the searched media. The problem with this approach is that some identified sectors are present in thousands of different files. Attempting to reassemble each of these files caused significant performance degradation and needlessly notified the human analyst with reports of files that were only referenced by these extremely common blocks.

Our second attempt at an algorithm included a candidate selection step, but we selected candidates based on the presence of *locally distinct* blocks—that is, a block on the media that was present just once in our database. Restricting candidate selection to locally distinct blocks reduced the number of candidates, but still left hundreds of false positives. The problem was that even though our database contained just one copy of the block, associated with a single file, in the wild the block actually appeared in many different files. Inspection revealed these blocks typically contained binary data structures produced by Microsoft Office and Adobe Acrobat. Even though we saw no collisions within GOVDOCS, each time we searched another drive the chances for a collision increased.

A second problem with basing candidate selection on locally distinct blocks is that there may be legitimate reasons for the hashes of low-probability blocks to appear multiple times in the block hash database. For example, our database might include two versions of the same video: a full-length 60 min version, and a truncated 50 min version. If candidate selection is based solely on the property of being locally distinct, then the first 50 min of the video would be ignored.

```
86435328      736d99610d0097be78651ecdae4714bb     {"count":39,"flags":"H"}
86435840      736d99610d0097be78651ecdae4714bb     {"count":39,"flags":"H"}
86436352      736d99610d0097be78651ecdae4714bb     {"count":39,"flags":"H"}
1231920640    90ccbdf24a74c8c05b94032b4ce1825d     {"count":1,"flags":"H"}
1231924736    9403e1cac89e860b93570ac452d232a5     {"count":1}
1231928832    b59246507f2bedb21957fae92bcf37d0     {"count":1}
1351669248    1e79c17035c597269b6fedf614663a1e     {"count":2,"flags":"HW"}
```

Fig. 3. Seven lines selected from the *identified_blocks.txt* file that resulted from scanning the jo-2009-11-20-oldComputer disk image with the *kitty+govdocs* block hash database. The first column is the byte offset from the beginning of the disk image at which the 4KiB block appears; the second column is the MD5 hash value of the block; the third column contains metadata including the count of the number of target files in which the block appears and any content-specific flags.

Instead, we re-designed candidate selection to take into account both the frequency with which the block occurs in our dataset and the characteristics of the pattern of bytes of which the block is composed. These rules are applied during media scanning and non-probative blocks are tagged as such so that they will not be used in candidate selection.

Finding rules for non-probative blocks

To develop rules for identifying non-probative blocks, we looked for common block hashes between the GOVDOCS hash database and the *oldComputer* disk image. We assumed all matches between the two datasets would be non-probative blocks, since we did not think that there would be any of the GOVDOCS files on the *oldComputer* image. We conducted the experiment by hashing each block on the *oldComputer* image and filtering against the GOVDOCS block hash database using a modified implementation of the HASH-SETS algorithm (described below) in which candidate selection had been disabled.

Much to our surprise, we found that there were four complete files from the GOVDOCS corpus present on the *oldComputer* image: File 466749.csv is a reference on Section 508 compliance; file 809089.eps is the "tcl Powered" logo. Files 574989.csv and 466982.csv are database dumps containing names, addresses, and dollar values. The csv files were likely downloaded by the students constructing the scenario, while the logo was present in a Python distribution on the searched drive (Fig. 1).

Eliminating these files, we found a total of 677 distinct blocks that were shared with 235 other files in the GOVDOCS set. We categorized the content of these spurious matches into three primary types and developed *ad-hoc* rules to identify and eliminate them. The next sections describe the rules and discuss our unsuccessful attempt to use Shannon entropy on 16-bit values as an alternative. (Fig. 4)

```
0000 6400 0000 01ff ffff 9c00 0000 0100
0000 6400 0000 01ff ffff 9c00 0000 0200
0000 0000 0000 0100 0000 6400 0000 01ff
ffff 9c00 0000 0100 0000 6400 0000 01ff
ffff 9c00 0000 0100 0000 6400 0000 01ff
ffff 9c00 0000 0100 0000 6400 0000 01ff
ffff 9c00 0000 0100 0000 6400 0000 01ff
ffff 9c00 0000 0100 0000 6400 0000 01ff
```

Fig. 4. An example of the low-entropy pattern found in both the QuickTime file KittyMontage.mov and in the PowerPoint file 182853.ppt.

The ramp test

By far the most common non-probative blocks identified by this test are those that appear to contain Microsoft Office Sector Allocation Tables (SAT). These data structures are defined by their length (typically 4KiB-12KiB) and their starting value, which corresponds to the ordinal number of the 512B block where the content stream appears in the disk file. There are thus tens of thousands of different 4KiB "ramp" blocks that can be observed for a given Microsoft Office file, with the result that the chance of a collision between any two Office files is small, but the chance of collision within a corpus of a few thousand Office files is quite high.

We developed a simple test that returns True if half of the bytes in a buffer match the ramp pattern, which was sufficient to weed out many cases in which the 4KiB block contained the SAT and other binary structures, or where the SAT referenced a few objects.

The White space test

Another kind of block that we have encountered are blocks consisting of blank lines of 100 spaces, each terminated by a newline character. Such blocks are commonly seen in JPEG files that were produced with Adobe PhotoShop, and are the result of whitespace padding located within the Extensible Metadata Platform XMP section (Adobe, 2012, p.10). Because the sections can appear on any 1-byte boundary, there are 101 different such common blocks filled with whitespace as a result of alignment issues. The end of the whitespace section typically has patterned data as well, resulting in tens of thousands of possible blocks that are mostly spaces but contain additional common material. Our whitespace test classifies blocks as non-probative if three-quarters or more of the block contains whitespace.

The 4-byte histogram test

Another common structure that we discovered with manual analysis is a block of patterned 4-byte values, either repeating or alternating 4-byte values. Analysis revealed these data structures in both Apple QuickTime and Microsoft Office file formats.

We devised a rule for eliminating sectors which contain a few repeating 4-grams. The rule treats the 4KiB buffer as a sequence of 1024 4-byte integers and computes a histogram of those numbers. It suppresses the sector if any single 4-gram is present more than 256 times (more than a

quarter of the block). This rule is unlikely to trigger on either text or image data, since neither typically have long runs of the same 4-byte values.

The entropy test

Foster (2012) observed that common blocks tended to contain low entropy data. This observation holds for the many of the non-probative blocks we examined while creating our ramp, whitespace and histogram tests. We hoped that we could replace our *ad-hoc* non-probative tests with a single test for low entropy data.

Our entropy test treated each 4KiB buffer as a collection of 16-bit unsigned integers and calculated the Shannon entropy for each. This method yields an entropy score of 6.0 for buffers flagged by the "ramp" test, and a score between 0 and 1 for blocks matching the whitespace test. The vast majority of blocks flagged by the histogram test scored less than 5, though these blocks had the widest range overall, with some blocks scoring as low as 0 and as high as 8.373.

We experimented with a range of threshold values and found that flagging blocks with an entropy value of less than 7.0 produced results that closely corresponded to the union of the sets identified by the whitespace, ramp and histogram tests.

Determining the effectiveness of the rules

We applied the rules described in this section to the 677 distinct blocks in the GOVDOCS dataset that matched the *oldComputer* drive and were not in the four identified files. The ramp rule matched 200 blocks and the histogram rule matched 400. An entropy threshold of 7 also identified a set of 600 blocks as non-probative. This set was identical to that given by the combination of histogram and ramp rules, except that it omitted one block flagged by the histogram rule, and included one additional block that was not flagged by any other rule.

We examined each of the remaining, unflagged blocks and could discern no obvious patterns in the data. We suspect that they may contain shared binary structures, fonts, or other kinds of information.

Although both the rules and the Shannon entropy threshold appear to be effective at eliminating spurious matches in our experiment, we need to evaluate them also with respect to the number of probative blocks that they suppress. Among the 21,469 blocks belonging to either the kitty materials or the four identified GOVDOCS files, the *ad hoc* rules match 126 blocks, whereas the entropy threshold test flags 149. A total of 78 blocks are matched by both methods, and 197 are identified as non-probative either because they match one of the rules or because their entropy is below 7.0.

We inspected each of these 197 blocks and found that they were divided more or less equally into two types: non-probative blocks consisting of metadata, unpopulated arrays, control structures, etc., and hybrid blocks containing a mix of data and long strings of nulls.

Because our carving technique tolerates the elimination of some probative blocks, we conclude that either classification method is adequate as long as it does not eliminate the majority of the blocks in a given target file. We performed a full analysis of all the blocks in our database and found that, unfortunately, there are some cases where this does occur. Specifically, we found that 1.6% of the files in our corpus had over 90% of their blocks flagged by the *ad hoc* tests, and over 90% of the blocks in a target file fell below the 7.0 entropy threshold for 12.5% of all target files. The probability of carving these files is obviously greatly reduced if fewer than 10% of their blocks can be chosen as candidates. These numbers led us to prefer the rule-based approach over the entropy threshold.

The candidate selection implementation

We implement candidate selection with the combination of two programs: *hashdb* and a post-processing script called *report_identified_runs.py*.

We first use *hashdb's explain_identified_blocks* command to determine the files to which these block hashes correspond.[5] Because some hashes match thousands of target files, and some hashes appear in tens of thousands of locations on searched media, *hashdb* implements a data reduction algorithm: the program reads the *identified_blocks.txt* file and builds an in-memory set of de-duplicated sector hashes. For each block, if the block maps to fewer than N files (the default is 20), those files are added to the set of candidate files. Finally, the program writes out a new file called *explain_identified_blocks.txt* with a list of the de-duplicated sector hashes, the number of times that the hash appeared in the original file, and all of the source files in which the hash appears. The database also creates a list mapping the source file IDs to the repository, filenames, original file sizes, and file hashes. An excerpt is shown in Fig. 5.

Next, we run the program *report_identified_runs.py* which reads the *identified_blocks_explained.txt* file. For each identified block, the block's `source_id` is added to the set *candidate sources* if the block is not flagged by any of the *ad hoc* tests described in the previous section. This is easily implemented using the *explain_identified_blocks.txt* output file: if the hash has no flags, the hash's sources are added to the set of candidate files.

Target matching

After the candidate files have been selected, the block hashes corresponding to candidates are grouped into source files and eventually tallied or reassembled into runs of matching blocks with the HASH-SETS and HASH-RUNS algorithms.

HASH-SETS: reporting the fraction of target files

HASH-SETS is a simple, memory efficient algorithm that uses block hashes to report the fraction of blocks associated with each target file that is present on the searched media.

[5] `hashdb explain_identified_blocks kitty.hdb out-kitty2/identified_blocks.txt > out-kitty2/identified_blocks_expanded.txt`.

```
["0a99....9187",{"count":1,"flags":"H"}, [{"source_id":137363,"file_offset":98304}]]
["1e79....3a1e",{"count":2,"flags":"HW"},[{"source_id":461464,"file_offset":536576},
                                          {"source_id":942213,"file_offset":675840}]]
...
["90cc....825d",{"count":1,"flags:"H"},  [{"source_id":974751,"file_offset":4096}]]
["9403....32a5",{"count":1},             [{"source_id":974751,"file_offset": 8192}]]
["b592....37d0",{"count":1},             [{"source_id":974751,"file_offset":12288}]]
...
{"source_id":137363,"repository_name":"default_repository","filename":"/govdocs/736/736684.ppt",
     "filesize":819200,"file_hashdigest":"43b9964b1585015d6f888e46f7851557"}
...
{"source_id":974751,"repository_name":"default_repository",
     "filename":"KittyMaterial/HighQuality/DSC00051.JPG",
     "filesize":1050508,"file_hashdigest":"b1ee6681fa420932319b75bd1e36eb21"}
```

Fig. 5. Applying *hashdb's explain_identified_blocks* command to the excerpt in Fig. 3, the first set of hashes located in 39 separate sources are suppressed, while the remaining are annotated in the JSON output indicating the source files and the offsets within the files. Information on each source is then presented. The hashes are shortened and spaces are added for brevity and clarity. The remaining hashes map to the files 736684.ppt and DSC00051.JPG.

The algorithm gets its efficiency by not tracking the exact location of each block hash on the media. The implementation is straightforward:

1. A list of candidate targets is determined using the candidate selection algorithm in the previous section.
2. For each block hash *H* in the file *identified_blocks_explained.txt*:
   (a) For each target *T* that matched against *H*:
      i. If *T* is a Candidate target, add 1 to that target's score.
3. For each target *T*, compute the fraction of the file present by dividing the score by the number of blocks in the target file.
4. The targets are sorted in inverse order using the fraction present as the key.
5. If the fraction recovered exceeds a predetermined threshold, the target file name, number of recovered blocks, and fraction recovered are reported.

An earlier version of this algorithm scored each block using inverse document frequency (1/N), but we changed the algorithm to simply count the number of blocks so that the score would be solely a function of the target file and the searched drive, and not of the construction of the hash database.

Searching oldComputer for Monterey Kitty

We ran HASH-SETS on the identified files associated with the *oldComputer* disk image. The algorithm completely recovered 86 files, including 82 of the Monterey Kitty files and 4 of the GOVDOCS files (which were also on the drive). However, the algorithm could only recover 67% of the file Cat.mov, as the QuickTime file contains 463 blocks that are entirely filled with NULLs. We address this problem in the next section.

The search of *oldComputer* also identified the presence of the GOVDOCS file 153348.png with a fraction detected of 0.18, and 34 other files with a fraction detected below 0.03 (Table 1). We think that these files were not present, but that they share a few non-probative blocks with other files on the drive. These files were not screened out by the candidate selection process because in each case they contained at least a single block that was identical to a block found in the disk image and that was not eliminated by our non-probative block rules. Presumably, these blocks contained binary data structures created by Acrobat or Office.

HASH-RUNS: locating target files

HASH-SETS detects the presence of target files but does not report their location, as location is discarded during the data reduction step. The HASH-RUNS algorithm reports location and has additional improvements:

- It gracefully handles the case when the target file is on the searched media in multiple locations.
- It takes advantage of the fact that adjacent logical sectors in a file tend to occupy adjacent physical sectors of the searched media.
- It takes into account the fact that different blocks in a file will have the same (mod 8) value.
- It can detect when runs of recognized blocks are separated by NULL blocks and combine them.

The algorithm starts with the data structures created by the HASH-SETS implementation. It then identifies all of the block runs on the physical disk that correspond to logical runs of the target files. These blocks are sorted by logical block number in the target file and reported to the analyst.

Table 1
Some matched files from GOVDOCS on the *oldComputer* disk, sorted by fraction detected. The first four files in **bold** could be completely recovered from the *oldComputer* drive. Only a few blocks from the other files are recovered. These are non-probative blocks which occur by chance in both GOVDOCS and on the searched drive.

File	4KiB blocks in file	Blocks detected	Fraction detected
466982.csv	848	848	1
809089.eps	6	6	1
574989.csv	6	6	1
466749.csv	3	3	1
…			
153348.png	11	2	0.18
569152.pdf	395	11	0.028
284845.ps	113	3	0.027
393395.eps	60	1	0.017
30 more files …			

1. The algorithm first reads the *identified_blocks_explained.txt* file (previously produced by the *hashdb* program) and builds an in-memory database that maintains the set of sector hashes associated with each target file.
2. For each (target file, sector hash) pair, the algorithm builds a second database that records the set of block numbers in the target file where the block appears. So if target file A has 6 blocks, and both blocks 1 and 4 match sector hash H1, then the element (A,H1) of the database contains the set {1,4}.
3. Next, the algorithm reads the *identified_blocks.txt* file and, for each sector hash that was found in a target file, records the disk block at which the hash was found (where the disk block number is the sector number, or the offset in bytes from the beginning of the disk, divided by 512.)
4. Now comes the target matching step. For each (target file, mod8) combination:
   (a) The algorithm builds an array consisting of elements in the form:
   `[disk block, {file blocks}, count]`
   Where `disk block` is the physical disk block, `{file blocks}` is a set of blocks within the target file where the hash was found, and count is the number of times in the sector hash database that the sector hash was found. (The `count` is not used by the algorithm but is included here for illustrative purposes.) (Table 2)
   (b) The array is sorted by `disk block`. For the hypothetical carving example in Fig. 6, the array would look like this:
   ```
   [1024, {0}, 1]
   [1032, {1, 4}, 2]
   [1040, {2}, 1]
   [1056, {3}, 1]
   [1064, {1, 4}, 2]
   ```
   (c) The algorithm runs a sliding window over the rows of the array to identify rows that represent sequential disk blocks and file blocks. Such sequences satisfy the constraint that, for each consecutive pair in the sequence, `diskSectorB == diskSectorA+8` and there exists an element in the set `fileBlocksB` that is 1 larger than an element in the set `fileBlocksA`. A new array of block runs is created, where each element in the array has the values:
   - *Identified File* (from the hash database)
   - *Score* (The number of identified blocks)
   - *Physical sector start*
   - *Logical block start*
   - *Logical block end*

   For our demo, the first three elements in the array can be combined, as can the last two, producing these two runs:
   ```
   [Demo, 3, 1024, 0, 2]
   [Demo, 2, 1056, 4, 5]
   ```
   (d) Two block run elements are combined if the number of bytes in the sector gap between the two runs matches the number of bytes in the logical blocks, and if all of the sectors on the drive corresponding to the gap contain only NULLs. (This step is required because the block of all NULLs is not indexed by our sector hash database engine.) *This is the only step that requires access to the original disk image.*

   For our demo, the two elements in the array of rows would be combined to create a single element:
   ```
   [Demo, 6, 1024, 0, 5]
   ```
   (e) Block runs that are smaller than a predetermined threshold are dropped. (By default, we drop those smaller than 3 logical blocks.)
   (f) Finally, for every reported run, we use SleuthKit to determine the allocated or deleted file that corresponds to the first block in the run.[6] This reporting is solely for the benefit of the analyst and is not used in the algorithm.

The algorithm is implemented in the Python program *report_identified_runs.py* included with the *bulk_extractor* release. The program's output is a CSV file that can be readily imported into Microsoft Excel.

Algorithm results

We probed the *oldComputer* drive using the *kitty + govdocs* database and the HASH-RUNS algorithm. The algorithm was able to provide significant information beyond the fraction of blocks present (reported by the HASH-SETS algorithm):

- HASH-RUNS determined that the entire *Cat.mov* file was present on the drive, since the recovered runs were separated by blank blocks and could be combined using the algorithm presented in the previous section.
- Unlike the HASH-SETS algorithm, HASH-RUNS detected several instances in which multiple copies of target files were present in multiple locations on the searched media.
- The number of false positive possible matches was reduced considerably. Whereas HASH-SETS identified 46 possible matches to the GOVDOCS corpus, HASH-RUNS identified the 4 correct matches and just 4 miss-matches due to common blocks. The higher precision is a result of the added requirement that recovered sectors form runs in which the sector number and the logical block number increase in step and at the appropriate rate.
- All of the miss-matches had scores of 3 or less and sector (mod 8) values (4, 5 and 6) that were different from the recovered files. Recall that all of the files within a single file system should have the same (mod 8) value (in this case 7). This is another indication that the files are in fact miss-matches.

[6] The actual determination is made using the SQLite3 database produced by SleuthKit's *tsk_loaddb* program and the rather complicated SQL statement `SELECT B.parent_path||B.name,size from tsk_file_layout as A JOIN tsk_files as B on A.obj_id=B.obj_id JOIN tsk_fs_info as C on B.fs_obj_id=C.obj_id where byte_start+img_offset ≤ BOFF and byte_start+img_offset+byte_len> BOFF`, where BOFF is the byte offset of the run from the beginning of the disk image.

Table 2
An excerpt of the output of the report_identified_runs.py program, in tabular form, and sorted by (mod 8) values. The files in **bold** are actually present on the media, whereas the others represent miss-matches from non-probative blocks. Notice that the files 466982.csv and 574989.csv are both present in their entirety, but in multiple runs that are not combined together in this report.

Target File name	Score	Start Sector	Start Block	End Block	Sector (mod 8)	Percent Recovered	Allocated File on Target Media
569152.pdf	3	18433052	373	375	4		n/a
569152.pdf	3	19652860	373	375	4	4%	*/WINDOWS/Fonts/courbd.ttf*
…							
970013.pdf	2	18433380	279	280	4	0.2%	n/a
970013.pdf	2	19653188	279	280	4	0.2%	*/WINDOWS/Fonts/courbi.ttf*
215955.ps	3	18192573	571	573	5	0.2%	*/Program Files/ … Data1.cab*
235835.ps	2	18192598	11503	11504	6	0.02%	*/Program Files/Adobe/Reader 9.0/ … Data1.cab*
MontereyKittyHQ.m4v	6132	18639703	0	6131	7	100%	*/Documents and Settings/ … /MontereyKittyHQ.m4v*
TiggerTheCat.m4v	3059	3532519	0	3058	7	100%	*/Documents and Settings/ … /TiggerTheCat.m4v*
KittyMaterial/Cat.mov	1374	18696759	0	1393	7	100%	*/Documents and Settings/ … /Cat.mov*
466982.csv	4	2932551	0	3	7	0%	*… /Cache/F5433139d01*
466982.csv	8	2833023	4	11	7	1%	*… /Cache/F5433139d01*
466982.csv	16	2800423	12	27	7	2%	*… /Cache/F5433139d01*
466982.csv	36	10062599	28	63	7	4%	*… /Cache/F5433139d01*
…							
DSC00072.JPG	234	14306831	0	233	7	100%	*/Documents and Settings/ … /DSC00072.JPG*
…							
809089.eps	6	11907023	0	5	7	100%	*/Python26/ … /pwrdLogo.eps*
574989.csv	4	3713503	0	3	7	67%	*… /Cache/4787E2CCd01*
574989.csv	2	3713359	4	5	7	33%	*… /Cache/4787E2CCd01*
…							
466749.csv	2	5032175	0	1	7	100%	*… Cache/_CACHE_003_*

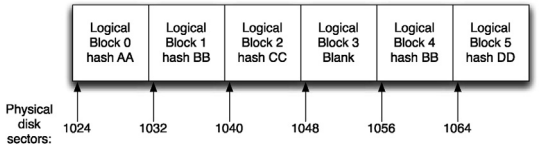


Fig. 6. A hypothetical hash-based carving exercise in which a file with 6 logical blocks is recovered with the HASH-RUNS algorithm. Logical blocks 1 and 4 are identical, whereas Logical block 3 is blank.

Conclusion

We have presented a hash-based carving system that uses the *bulk_extractor* program to create a block hash database of target files and a sector hash record of searched media, supported by the *hashdb* hash database to provide rapid lookups and perform initial correlation steps. Assembly of individually recognized block hashes into carved runs is performed by a post-processing Python script.

Although there has been considerable interest in hash-based carving for nearly a decade, to the best of our knowledge this article presents the first workable algorithm and reference implementation that can work with a large database of target file block hashes.

Possible improvements

Additional efficiency gains could be realized by flagging non-probative blocks during the database construction phase and storing the flag status in the target database. This alteration would permit each test to be run only once per distinct block, rather than repeating each time the block is encountered on the searched media. Furthermore, by moving the classification process to the target ingestion stage, we open the possibility for more computationally intensive tests, since there is no longer the need to keep up with speed of the disk scanning process. Finally, we can avoid storing files that are composed entirely of non-probative blocks and are therefore not recoverable by this method.

Another improvement is to consider file allocation status in addition to considering (mod 8). In general, we would expect all of the blocks associated with a target file to be in the same file allocation status—either allocated or unallocated. (When the target file is written to the media it is allocated; if it is later deleted, it is unallocated.) File allocation status could be considered by simply adding another filtering step. However, computing file allocation status on a per-block level is computationally expensive and would not improve the accuracy of the file recovery in our examples here.

Limitations

If the media contains an encrypted file system, it is necessary to first decrypt the file system so that the unencrypted blocks can be accessed. This is typically done by mounting the media using an appropriate decrypting driver.

Acknowledgments

Bruce Allen is the primary developer of the hash database; Joel Young contributed to *hashdb*'s initial design. Rob Beverly, Kevin Fairbanks and the anonymous reviewers provided useful comments on this article.

References

Adobe. XMP specification part 1: data model, serializaiton, and core properties. 2012. http://www.adobe.com/devnet/xmp.html.

Allen B. hashdb. 2014. https://github.com/simsong/hashdb.git.

Collange S, Dandass YS, Daumas M, Defour D. Using graphics processors for parallelizing hash-based data carving. CoRR abs/0901.1307. 2009., http://arxiv.org/abs/0901.1307.

Collange S, Daumas M, Dandass YS, Defour D. Using graphics processors for parallelizing hash-based data carving. In: Proceedings of the 42nd Hawaii International Conference on System Sciences; 2009. Last accessed 03.12.11, http://hal.archives-ouvertes.fr/docs/00/35/09/62/PDF/ColDanDauDef09.pdf.

Dandass YS, Necaise NJ, Thomas SR. An empirical analysis of disk sector hashes for data carving. J Digit Forensic Pract Apr. 2008;2(2):95–104. http://dx.doi.org/10.1080/15567280802050436.

Foster K. Using distinct sectors in media sampling and full media analysis to detect presence of documents from a corpus. Master's thesis. Naval Postgraduate School; Sep. 2012.

Garfinkel S. Digital media triage with bulk data analysis and bulk_extractor. Comput Secur Feb. 2013;32:57–72.

Garfinkel S, Nelson A, White D, Roussev V. Using purpose-built functions and block hashes to enable small block and sub-file forensics. In: Proc. of the Tenth Annual DFRWS Conference. Portland, OR: Elsevier; 2010. S13–23. http://simson.net/clips/academic/2010.DFRWS.SmallBlockForensics.pdf.

Garfinkel SL. Dfrws 2006 challenge report. 2006. http://sandbox.dfrws.org/2006/garfinkel/part1.txt.

Garfinkel SL. Announcing frag_find: finding file fragments in disk images using sector hashing. Mar. 2009. http://tech.groups.yahoo.com/group/linux_forensics/message/3063.

Garfinkel SL, Farrell P, Roussev V, Dinolt G. Bringing science to digital forensics with standardized forensic corpora. In: Proceedings of the 9th Annual Digital Forensic Research Workshop (DFRWS). Quebec, CA: Elsevier; Aug. 2009.

Key S. File block hash map analysis. 2013. https://www.guidancesoftware.com/appcentral/.

Microsoft Corporation. Default cluster sizes for ntfs, fat and exfat. 2014. http://support.microsoft.com/KB/140365.

Taguchi JK. Optimal sector sampling for drive triage. Master's thesis. Naval Postgraduate School; 2013.

Wang X, Yu H. How to break md5 and other hash functions. Adv Cryptol Lect Notes Comput Sci 2005;3494:19–35.

Young J, Foster K, Garfinkel S, Fairbanks K. Distinct sector hashing for target detection. IEEE Comput Dec. 2012:28–35. http://simson.net/clips/academic/2012.IEEE.SectorHashing.pdf.