Summer 8-11-2016

Practical Application of Fast Disk Analysis for Selective Data Acquisition

sergey gorbov
University of New Orleans, New Orleans, sgorbov@uno.edu

Practical Application of Fast Disk Analysis for Selective Data Acquisition

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science
Information Assurance

by

Sergey Gorbov

University of New Orleans

August, 2016

Abstract

Using a forensic imager to produce a copy of the storage is a common practice. Due to the large volumes of the modern disks, the imaging may impose severe time overhead which ultimately delays the investigation process. We proposed automated disk analysis techniques that precisely identify regions on the disk that contain data. We also developed a high performance imager that produces AFFv3 images at rates exceeding 300MB/s. Using multiple disk analysis strategies we can analyze a disk within a few minutes and yet reduce the imaging time of by many hours. Partial AFFv3 images produced by our imager can be analyzed by existing digital forensics tools, which makes our approach to be easily incorporated into the workflow of practicing forensics investigators. The proposed approach renders feasible in the forensic environments where the time is critical constraint, as it provides significant performance boost, which facilitates faster investigation turnaround times and reduces case backlogs.

Keywords

Digital forensics, selective imaging, partial image, fast disk analysis, statistical disk analysis, fast imaging, concurrent imager, bitmap walking

Table of Contents

List of Figures

List of Tables

Nomenclature and Abbreviations

chunk - an atomic data unit for disk representation
chunk map - an ordered sequence of chunks that represent a source disk
generic bitmap - the generic interface to the allocation management facility of a filesystem
bitmap mask - the interface that converts a chunk map in to a bitmap
complete bitmap - overlapping of the generic bitmap and the bitmap mask
bitmap walking - the procedure that implies traversing the complete bitmap
CBW - Coarse Bitmap Walk
FBW - Fine Bitmap Walk

Chapter 1

Introduction

Almost all digital forensics investigations require evidence preservation. Depending on the nature of the evidence being examined, different techniques apply in order to effectively extract data. For example, in order to extract data from non-volatile storage devices such as hard disk drives, solid state drives, and flash drives, forensic investigators typically use a tool called an *imager*. There are plenty of imagers available, both open source as well as commercial ones. Imagers vary from the simplest tools that blindly replicate a source device into a raw image (e.g., the Unix *dd tool*[1]), to complex software kits, e.g. *AccessData FTK*[2], *X-way*[3], *ADF Triage*[4] that can perform disk analysis and preserve data in special space-efficient forensic formats. The ultimate objective for an imager is to produce an accurate forensic image as quickly as possible, as time is often the critical limiting factor in forensic investigations.

The trend of continuously increasing capacity for non-volatile storage devices has created a problem known in the digital forensics community as the *volume challenge*. Traditional approaches to imaging require a complete copy of a source storage device, which means that the source disk must be read in its entirety. The two main constraints that impose time overhead are the speed at which the source disk can be read and the speed at which a forensic image can being written onto a destination disk that will hold the copy of the data.

The write constraint can be relaxed by using data formatting, which reduces the amount data to be written onto the copy of the evidence. Modern forensic formats typically use data compression for reducing space of a resulting image, while also incorporating hashing to preserve the authenticity of evidence (e.g., the *Advanced Forensic Format*[5][6]). However, both formatting features come at the expense additional CPU overhead. Even if a source device is plugged into a fast interface, imaging speed can still be degraded due to inefficient algorithms used by the imaging tool.

The read constraint has widely been discussed in the digital forensics community, and current trends are directed towards the selective imaging. For example imaging a 10TB storage device at the speed of the SATA3 interface (6GBit/s or 600MB/s) would take approximately 5 hours, which is actually not bad. However in practice an average SATA3 disk's read speed will be limited at approximately 150MB/s, which means that, in the ideal case scenario when the disk does not contain errors, the imaging time extends from 5 to 20 hours. The situation with solid state drives is a bit better, as they provide read speeds of 300 - 500 MB/s, depending on vendor. Nevertheless, as solid state drives gain popularity, they too will increase substantially in size, and exacerbate the volume challenge. Thus selective imaging is meant to reduce the quantity of data that is required, while attempting to maximize the quality of the resulting image.

In the present work, we propose techniques that are meant to reduce imaging time by carefully omitting empty and other unimportant regions on the disk, so that only potentially important data needs to be imaged. Using multiple analysis strategies in conjunction, we are able to analyze a disk within a few minutes and yet reduce the imaging time of large devices by many hours. After the analysis of a disk is completed, our high performance selective imager can quickly produce an AFFv3 image of the disk, which is compatible with any forensics tool that can read AFFv3 images. This allows our tools to be easily incorporated into the workflow of practicing forensics investigators.

Related Work

Turner described the concepts of selective and intelligent imaging and discussed a number of options available when capturing data in selective manner[7]. He also proposed the data container called a Digital Evidence Bag (DEB), which is used to store objects selectively gleaned from multiple sources. The logical and structured approach of DEB maintains provenance and integrity of the evidence.

Whereas Turner applied selectiveness only at the file level, J. Stuttgen introduced a more generic concept, where any data that can be found on disk is considered as an object of potential interest and included in the imaging process[8]. However Stuttgen's scheme involves a human in the loop, which imposes additional time requirement for an investigator to pre-process the evidence. Furthermore, as storage sizes increase even further, a fully-automated solution is preferable.

Richard and Grier proposed a rapid imaging technique called sifting collectors[9] and they developed various sifting techniques to identify relevant data on disk. Using The Sleuth Kit's (TSK)[10] disk analysis capabilities, they automatically identified sectors associated with files of potential forensic interest. The file filtering is handled by specifying profiles that are essentially collections of regular expressions that express which files are relevant and should be collected. Once the analysis phase is done, the intermediary data, called grainset, is passed to a collector that extracts designated data from source disk and stores it as a sifted Advanced Forensic Format v3(AFFv3) image.

Richard and Grier achieved up to x13 speed increases by sifting out irrelevant disk regions. The weak link in sifting collectors is that it performs the selectiveness only on the filesystem level. Because of this limitation, sifting collectors cannot identify the presence of data on a sub-filesystem level, e.g., volume slack space, unallocated space. Such analysis requires different approaches, e.g., sampling[11][12][13][14], which we use in the present work. Another major limitation of sifting collectors was poor performance of the imager. Richard and Grier used the AFFv3 library[6] in order to produce *.AFF* format images and this library performs reading, hashing, compression and writing routines sequentially. Such inefficient resource utilization ultimately degraded the imager's performance in all aspects: the source I/O read time, the CPU overhead imposed by hashing and compression, and the destination I/O write time. While our approach also generates AFFv3 images, we have completely parallelized both the AFFv3 implementation as well as our own imaging components, which rely upon it, all without compromising the portability that AFFv3 offers.

Shatz proposed extensions for Advanced Forensic Format 4 (AFF4) that allowed the acquisition routine to proceed at higher bitrates[15], by using lightweight compression schemes, which efficiently handle low entropy regions by compressing them symbolically. The symbolic compression is facilitated by the AFF4 format, which uses supports predefined streams of homogeneous data. Using block-based hashing, Shatz entirely omits hashing of symbolically compressed blocks, thus decreasing CPU overhead. Shatz also proposed horizontal scaling, e.g., storing evidence across multiple drives using data striping during acquisition. Nevertheless the proposed approach still requires the source disk to be read from the first to the last byte during the imaging. Also, the AFF4 format has not seen widespread adoption due to its complexity, and cannot be uniformly used with existing digital forensics tools. Aside from these concerns, the data striping acquisition is not a portable solution, since it would require additional hardware with custom drivers in order to store and view the striped data.

Chapter 2

Background

This chapter discusses essential background concepts that underlie our efforts to develop a time efficient and accurate forensic imager.

Target Filesystems

Carrier described in detail the most widely used filesystems and volume systems in his book *File System Forensic Analysis*[16]. So as not to reinvent the wheel, we used the open source *TSK API*[17] in order to evaluate disk layout and run analysis on filesystems. TSK does not rely on the native OS filesystem drivers, which can easily be subverted by malware, but rather it analyzes the disk in a direct and non-intrusive fashion, which ensures data integrity. TSK has support for a wide variety of volume and filesystem types.

As of today, Microsoft Windows has the largest segment of the commodity operating systems market, and Windows systems must typically be installed on a New Technology Filesystem (NTFS). Thus we mainly consider Windows filesystems in our effort. The FATxx filesystem has been around for long time and is still in widespread use. We do not support FAT16 at all, as it is very limited in maximum partition size and it is easy to simply capture the entire filesystem. FAT32 remains useful as a cross-platform filesystem, so we support FAT32, even though maximum partition sizes are relatively modest. The Linux *ext3/4* family of filesystems and Mac HFS+ are also widely used, but in this initial effort we do not currently fully support them, deferring such support for future work. Nevertheless, any filesystem that is recognized and supported by the TSK has limited support and still can be analyzed.

Disk Representation and Partial Image

An operating system sees a storage device as a flat array of sectors, and a sector is an atomic unit for a storage device. A typical sector size is 512 bytes and while there are disks that support 4096 byte sectors, these typically have backward compatibility features that also support 512 byte sectors. Modern storage devices may consist of billions of sectors. For example, a disk volume of 6TB translates into 11,718,750,000 sectors. In our research effort, we will have to maintain state per storage unit to track which units are imaged, therefore, for the 6TB case, we would have to have an array of almost 12 billion elements to track sectors individually. This would be extremely inefficient, so for performance concerns we use a larger atomic unit called a *chunk.* A chunk is a fixed size data unit and can be: 2^x, where $x = [20, 29]$. A disk representation using this strategy is shown in the Figure 2.1. The disk is represented as a set of chunks called *chunk map*.

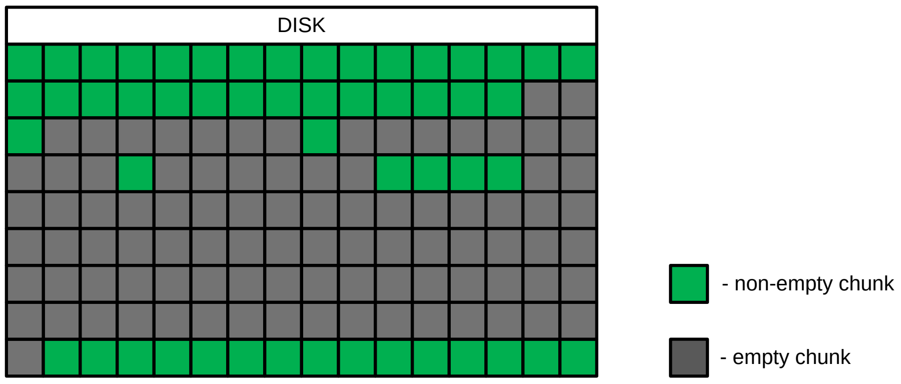

Figure 2.1. A disk represented as a set of fixed size chunks.

Any disk is considered to be a totally ordered set comprised of a finite numbers of fixed size chunks:

$$D = C_0 \cup C_1 \cup \ldots \cup C_n \qquad 2.1$$

A chunk is considered empty if and only if all the bytes within the chunk are equal to zero, denoted z. Otherwise the chunk is considered to be non-empty, and is denoted $\bar{z}$. Empty chunks never intersect non-empty chunks, i.e., $\forall z,\ z \in Z, z \notin \bar{Z}$; $\forall \bar{z},\ \bar{z} \in \bar{Z},\ \bar{z} \notin Z$. In order to produce a full image I_{full}, all chunks must be included into the final subset:

$$I_{full} = Z \cup \bar{Z} = D \qquad 2.2$$

In order to produce a partial image $I_{partial}$, only selected chunks must be included: $I_{partial} = f(D)$, where f is an fast disk analysis function.

The ground truth partial image is the subset of all non-empty chunks: $I_{partial}^{GroundTruth} = \bar{Z}$. A filesystem may contain legitimate files that are filled with zeroes and if, for example, a filesystem does not support spasrse files or a file is not set the attribute that turns on sparse storage feature, then such files might be also included into the ground truth partial image. In practice there is no way to produce an ideal ground truth image unless the entire disk is read. Our goal is to use fast disk analysis methods in order to compose the $I_{partial}$, that is the closest approximation to the partial ground truth image $I_{partial}^{GroundTruth}$. The difference Δ between the partial image and partial ground truth image is expressed as a symmetric difference between $I_{partial}$ and $I_{partial}^{GroundTruth}$:

$$I_{partial} \dot{-} I_{partial}^{GroundTruth} = \Delta \qquad 2.3$$

The Δ subset sub-divides onto false negative and false positive subsets:

$$\Delta = FN \cup FP \qquad 2.4$$

The false positives only impose additional overhead due to inclusion of empty chunks into the resulting image. The false negatives are undesired, as they break the completeness of the image. The overhead is evaluated as the following ratio:

$$Overhead = |FP|/|D| \qquad 2.5$$

Logical Representation

A disk can be considered as a complex infrastructure that accommodates multiple partitions and within each partition, a filesystem. In the scope of this research we only focus on the filesystem analysis. All problems related to integrity of volume systems, correct identification of partitions, and filesystem recognition are delegated to TSK, which has mature facilities for handling these issues. The Figure 2.2 shows the logical structure of a typical disk. The disk is logically split onto multiple partitions, which either contain a filesystem or a unallocated region.

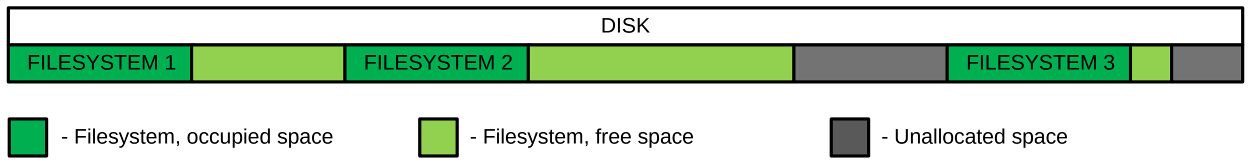

Figure 2.2. A typical disk layout.

A filesystem is a self-contained entity with numerous control structures and is aware of its own size. Each filesystem has a bookkeeping mechanism that allows for managing allocations and deallocations within the filesystem's storage space. A filesystem distinguishes only two regions, allocated (shown as green), and not allocated (shown as light-green). A filesystem uses a mechanism that keeps track of free space. The NTFS uses a bitmap that is a flat array of bits, each of which represents one cluster on the filesystem. If a bit in the bitmap is set then the associated with it cluster is allocated, otherwise the cluster is not allocated. FATxx systems use a similar approach but instead of a literal bitmap they use a file allocation table (FAT), which serves two purposes. First, each entry contains a reference to the next entry, such that walking the chain of entries in the FAT tracks file fragments. The second purpose is that it keeps track of allocations, i.e., if an entry has a non-zero value then the cluster associated with that entry is either used by some file, or is a bad cluster. The important issue is that regardless of the specific mechanism, filesystems know which regions are free and which are occupied.

The front end of a filesystem is the abstraction of file, directory, and an interface for interacting with these objects. There are a wide variety of filesystems in current use and while different filesystems may have completely different internal mechanisms, they carry out many of the same functions. TSK abstracts away these differences between filesystems and allows for conducting filesystem analysis in a generic way.

The contents of a filesystem can be treated as the filesystem's critical metadata structures plus a set of files. Each file is defined by metadata and typically has associated with it a group of clusters. There are also special types of files, which may not have distinct clusters associated with them (e.g., very small files on the NTFS are stored within the *$MFT* entries). File attributes and parameters are maintained by a file metadata facility, e.g. $MFT on NTFS. Any file that is currently allocated and has valid metadata can be located and accessed. If a file has been deleted, then its metadata may be fully or partially destroyed. In this case the file is unrecoverable via the filesystem's facilities. An example of a filesystem that does not facilitate data recovery is the ext3/4, which explicitly destroys the inode blocks chains associated with a file upon deletion, making it impossible to retrieve the deleted file via the filesystem. However, there are filesystems that do facilitate data recovery, whether as an intentional design choice or otherwise. These do not completely destroy the metadata upon file deletion, which allows queries against the file metadata facility to reveal the data clusters associated with the deleted file. Nevertheless, even if the filesystem facilitates file recovery, there are cases when recovery fails. There are two required conditions for the recovery to succeed. First, the file metadata must be intact in order to correctly locate the clusters and recover the file content. Second, the clusters must not be overwritten by other data, otherwise the content will be not recoverable. The clusters no longer associated with active files typically linger in the filesystem until they are reused. Thus, a typical filesystem's contents can be expressed as a union of the three data subsets:

$$FS = FS_{allocated} \cup FS_{deleted} \cup FS_{unallocated} \qquad 2.6$$

An unallocated region(shaded regions) is considered any region that was not identified by TSK as a valid filesystem. Our approach allows for fast allocation pattern analysis within the disk, therefore each unallocated region can be pre-analyzed, and an appropriate decision can then be made accordingly.

Chunk Subsets

In the left portion of **Error! Reference source not found.**, data is represented from the filesystem perspective. As in e xpression (2.6), there are three subsets that comprise the filesystem.

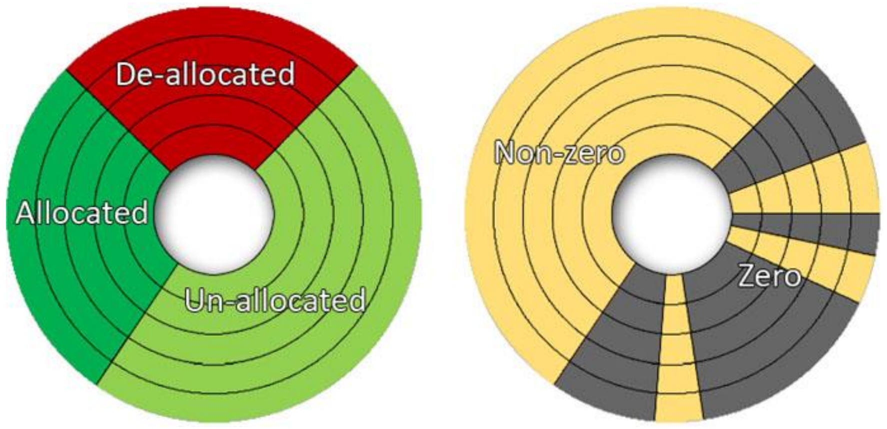

Figure 2.3. Left portion: data representation from the filesystem's perspective. Right portion: raw data representation.

The allocated subset describes critical filesystem structures that are required for a filesystem to properly function. The allocated subset also includes all currently allocated data that is available to user. Imaging only the allocated region is enough for a filesystem to operate normally. Therefore, the allocated subset has the highest priority in terms of forensic imaging, and must be included in the resulting image.

The deallocated subset describes data regions that were occupied by files in the past until they were deleted. Deleted files are often of forensic interest, as an adversary might delete some portions of available evidence either during a normal course of action or in a deliberate attempt to render data unrecoverable. Like the allocated subsets, deallocated subsets should be fully included in a forensic image.

Richard and Grier used the TSK API[17] for automated filesystem analysis and the approach they developed is able to efficiently identify allocated and deleted files and locate clusters associated with them. However, the scope of TSK does not extend to unallocated regions, which are also a part of the filesystem. Also, while a filesystem is aware of the limits of the unallocated regions, it is typically unaware of the content lingering in those regions. Unallocated regions may contain data that either has accumulated over time due to deletion of files and relocation of corresponsive metadata entries, or due to disk reformatting. Although the unallocated data within a filesystem is essentially a raw binary blob, it may contain important evidence or fragments of evidence. Therefore portions of an unallocated subset that contains data should be included into the resulting forensic image. In the right portion of **Error! Reference source not found.**, there is an alternate view of the same, but expressed in zero and non-zero r egions. Denote the data part of unallocated region as FS_u' and an ideal resulting image would be:

$$I_{partial}^{GroundTruth} = \bar{Z} = FS_a \cup FS_d \cup FS_u' \qquad 2.7$$

Chapter 3

Methodology

From the discussion in the previous section, it is clear that in order create a partial forensic image that closely approximates the ground truth partial image, we must devise a disk analysis function $f(D)$ which will effectively identify chunks subsets FS_a, FS_d, and FS'_u. The complete disk analysis can be split into disk pre-processing and subsequent analysis of all of its partitions.

The processing of the contents of a partition may include more than one analysis strategy and specifically, we distinguish three fundamental strategies: *file-based*, *bitmap-based,* and *allocation-based* strategies. The three approaches differ in the principles they use to identify data regions. The file-based strategy utilizes the metadata facility on a filesystem in order to identify chunks that should be included into the final chunk subset. The bitmap-based strategy uses the allocation management facility on a filesystem in order to capture the allocated chunk subsets. The allocation-based strategy views the disk as raw data and applies statistical approaches in order to build up the unallocated subset FS'_u that contains data. The analysis strategies have four basic attributes: the processing time T, the scope S, analysis capabilities A, and the prerequisites R, which define what is required in order to run the analysis routine. Table 3.1 briefly reviews the analysis strategies and their attributes. The analysis strategies are discussed in detail in the following sections.

Table 3.1. An overview of the filesystem analysis strategies and their attributes.

Analysis strategy	Attributes			
	T	S	A	R
File	Content dependent. The analysis time depends on the amount of allocated and deallocated contents, and a file system type.	FS_a, FS_d	Basic consistency checks on content and metadata. The analysis is stopped if a metadata inconsistency is detected. Profile-based file triage.	Requires TSK support only. Consistency checks can only be done on NTFS and FAT32 filesystems.
Bitmap	Filesystem size dependent. For a 1TB with cluster size 4Kb the bitmap will be ~30MB. The time to walk through 30MB is approximately 5 seconds.	FS_a	Captures any data hidden via setting cluster allocation status on the bitmap. Checks if the bitmap size corresponds to the size of the filesystem.	Analysis can be done only for NTFS and FAT32. Analysis cannot be done if the bitmap cannot be located on the filesystem.
Allocation	Filesystem size dependent. Data allocation pattern dependant. Depends on multiple input parameters.	FS_d, FS'_u	N/A	Requires either bitmap-based or file-based analysis to be done in order to proceed.

Disk Pre-Processing

When a source disk is supplied as an input, the first analysis routine must pre-process the disk and pass it on for further processing. The disk pre-processing routine is illustrated in the Figure 3.1. Its primary objectives are to measure the disk, initialize the chunk map, add the first and last chunks of the disk, and query the disk partition table to recognize filesystems and unallocated regions between them. For each partition, the routine includes the first and last chunks in the final chunk subset. If there is a slack space between the filesystem and the volume boundaries, this slack space is also added to the final chunk subset. If a partition does not contain a valid filesystem or the filesystem is not supported by TSK, then it can be fully added to the final chunk subset or passed for further analysis (small regions can be added without analysis). Each discovered filesystem is measured and a filesystem object is created and inserted into the filesystem array, which upon completion of the disk pre-processing is passed down to filesystem processor.

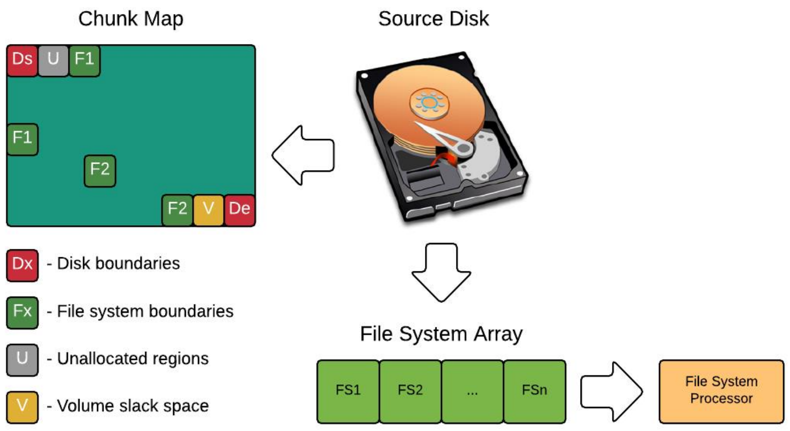

Figure 3.1. Disk Pre-Processing abstract schema.

One might be curious about why we are turning on chunks that designate boundaries for the disk and each volume. A typical disk normally starts with a *Master Boot Record (MBR)*, but if the disk has a *GPT* structure then the first chunk will also include the primary GPT header, and the last chunk will capture the secondary GPT header. Even with the minimum chunk size that is equal only 1MB, these essential structures will be included into the resulting image.

A similar reason justifies capturing a volume's boundaries. Some volumes may contain a *Volume Boot Record (VBR)* structure, which is the first sector on the volume. Notably, however, a filesystem inside a volume might not be the same size as the volume. In this case there is slack space that can exist before the first byte of the filesystem and/or after the last byte of the filesystem. Some filesystems, e.g., NTFS, may use such this slack space to store a backup of the boot record. Therefore to ensure the integrity of the disk we add these corresponding chunks into the final chunk subset.

Any error during disk pre-processing will cause termination of further analysis, which will ultimately mean that the disk should not be analyzed using our methods because the disk metadata is corrupted. The analysis of corrupted disk may produce inaccurate results, which defeats the purpose of imaging. In this case the investigator is urged to produce a complete forensic image using traditional methods, if possible.

Filesystem Processing

After the disk pre-processing has completed, the resulting filesystem array is passed to the filesystem processor, where each filesystem object is analyzed individually. The filesystem processor is an abstract interface. For each filesystem the filesystem processor must be implemented individually. The objective of a filesystem processor is to check whether a filesystem meets sanity checks so it can be analyzed. If a filesystem passes consistency checks then it is approved for further analysis, otherwise it is rejected and added into the final chunk subset in its entirety.

As mentioned earlier, we currently support only NTFS and FAT32 filesystems. For each filesystem we make sure that its critical structures are in a valid state. For example, for FAT32 we examine the boot sector and make sure that it does not have suspicious values set. We also identify and capture the FAT32 reserved region and File Allocation Tables FAT1 and FAT2. For NTFS we simply make sure that we can locate and access the clusters of the essential MFT entries, such as $MFT, $MFTMirr, $Bitmap, $BadClus, et al. that comprise the filesystem metadata. If all entries are located then they are included in the final chunk subset and the analysis proceeds. If one entry could not be found then the consistency check is failed and the entire filesystem is added to the final chunk subset.

Bitmap Interface

For each filesystem that successfully passes the consistency check we create a generic *bitmap interface*. A bitmap interface is an abstract entity that defines access operations for the allocation management facility of a filesystem. For NTFS, the generic bitmap interface simply provides access to the $Bitmap file. For FAT32, which does not have a literal bitmap, the generic bitmap interface provides on-the-fly translation of the FAT into a bitmap, e.g., each FAT entry is converted into one bit. Bitmaps are queried via a *bitmap index*, which represents one *bitmap transaction*, spanning 64 sequential bits. Thus one bitmap transaction describes 64 clusters on a filesystem and represents one *data transaction*. For a typical cluster size of 4096, the bitmap transaction translates into a data transaction of 262144 bytes. Thus larger cluster sizes result in larger data transaction sizes.

The bitmap interface provides optimizes the interaction with actual bitmaps using bitmap buffering, e.g., only a small size of a bitmap resides in the memory at any given time.

The main reason behind introducing the concept of a custom bitmap interface is that the bitmap is the central component for the bitmap-based and the allocation-based analysis strategies. The bitmap interface turns a static bitmap structure B, fetched from a filesystem into a dynamically updated bitmap by overlapping the actual bitmap with the *bitmap mask* M. The bitmap mask is a translation of chunk map region that corresponds to a filesystem being analyzed such that the *complete bitmap* $B_{complete}$ is expressed as: $B_{complete} = B \cup M$.

There is an edge case when a bitmap mask is used as a complete bitmap: $B_{complete} = M$. For unsupported filesystems the bitmap structure cannot be retrieved. If the filesystem is supported by TSK, then having conducted the file-based phase it is possible to produce a chunk map that may be used as a start point for the allocation-based analysis.

Analysis Strategies

File-based strategy

The files are located on the filesystem using their metadata information. Based on the byte offsets a file is expressed in terms of chunks and the file chunk subset is added to the final chunk map. This procedure is performed for all valid file metadata entries that TSK can discover on the filesystem.

During this phase there is a possibility of content or file metadata inconsistency. Content inconsistencies occur when allocated clusters are associated with more than one entry. A file metadata inconsistency occurs when a file's metadata indicates that its data is stored in a region of unallocated clusters or beyond the limits of the filesystem. Any inconsistency will stop further analysis and add the entire filesystem to the final chunk subset, since our tools do not currently differentiate between minor and major filesystem corruptions.

The file-based strategy is mainly used either for *profile-based file triage* or for non-supported filesystems in order to build up a bitmap mask based on the produced chunk map.

Profile-based File Triage

In order to apply selectiveness at the file level, we have implemented a profile-based triage feature, which allows imaging of desired file types and/or names based on a pre-defined *profile*. This system is an expansion of that proposed by Richard and Grier[9]. The profile is organized as a structured set of regular expressions with simple syntax, and must have the extension *.profile* in order to be recognized by the system. If a profile is malformed, the parser will provide an error message and point out the error location and describe the issue.

The profile supports three fundamental directives:

- *:Files* - defines the patterns for files. During the file search if a file matches at least one pattern the file location is expressed in terms of chunks and added to the final chunk map.
- *:Path* - defines the patterns for paths. The mechanism is the same as in the case with files, but instead of filenames, paths are evaluated according to the supplied patterns.
- *:include* - includes an existing profile into the current profile. This allows multiple profiles to be used in conjunction.

The following section represents an example of a profile that targets pictures:

```
:Paths
/Pictures
/My Pictures
/Pics
/Photos

:Files
\$
\.(png|gif|jpg|bmp|ppm|xbm|raw|tiff|tif|crw|cr2|nef)$
```

In order to supply desired paths and files we first supply a desired directive, e.g., *:Paths* or *:Files*. Any data supplied under the directive is considered related to that directive. A directive's scope ends if another directive is encountered.

If we want to combine several profiles into one we use *:include* directive, as shown below:

```
:include "Email.profile"
:include "Financial.profile"
:include "Photography.profile"
:include "WindowsBrowserHistory.profile"
:include "WindowsDataExfiltration.profile"
:include "WindowsDocuments.profile"
:include "WindowsLogs.profile"
:include "WindowsRegistry.profile"

Files:
\$
^SAM(\.log)?$
^SECURITY(\.log)?$
^SOFTWARE(\.log)?$
^Default(\.log)?$
^Userdiff(\.log)?$
^Ntuser\.dat$
^Usrclass\.dat$
^Repair(\.log)?$
^System(\.alt|\.log)$
```

The given profile has the *:File* directive that is given the set of desired file patterns. The profile also includes other profiles and inherits their search criteria. It is possible that a profile may try to include another profile that has the calling profile already included, which causes infinite recursion. The parser recognizes this situation and ignores recursive profile inclusions.

The final chunk subset produced via profile-based file triage describes a filesystem in such a way that it can be mounted and analyzed as if it was cloned. In the resulting partial image the filesystem metadata will be preserved, so it will have an identical layout with the original filesystem. However an investigator will be able to access only those items that matched the search patterns—others will appear to be zero-filled.

This technique is useful if an investigator has a preliminary knowledge about the case, so he can quickly use the profile and extract only relevant data that may be several orders of magnitude smaller than the size of the entire disk. The complete procedure may take just a few minutes, compared to hours if traditional imaging were used.

The profile-based file triage is a file-based analysis, and it inherits all the limitations related to it. Thus the profile-based file triage cannot be used if the filesystem is in an inconsistent state, e.g., has corrupted metadata. Another limitation is that if a filesystem is has a very large number of files, then in some cases it would be faster to image the entire filesystem. In particular, TSK processing time drastically increases as the amount of entries on a filesystem grows.

During the pre-processing stage, our tool summarizes some important information about each analyzed filesystem and provides this information to the investigator:

- The number of entries in the metadata management facility. For NTFS, this is information about how large the $MFT structure is. For filesystems like FAT32, we cannot provide this information, because there is no means to discover the number of files without searching for *Directory Entries,* which may be scattered all over the filesystem.
- The average read speed and approximate imaging time for a current filesystem. Thus prior to starting selective imaging, an investigator knows how long would it take to produce a complete image. In order to enable *the read speed check* feature the configuration must be adjusted, so the program knows the interface bandwidth. This is currently required due to caching issues.

Thus having basic information about the filesystem that is about to be passed for further analyses, an investigator can get a first impression of what to expect if the analysis proceeds. In any event, if the file-based analysis is predicted to take too long and is considered impractical, it can always be terminated, and other means used for further disk examination.

Bitmap-based strategy

The bitmap-based strategy captures all valid entries that are currently allocated on a filesystem. It is much faster than the file-based approach as it does not invoke TSK file searching routines, which may take a long time for filesystems with large number of files.

During the bitmap-based analysis, the complete bitmap is walked from start to ending bitmap index, sequentially processing each bitmap transaction. The entire complete bitmap can be expressed as a set of all the bitmap transactions, $B_{complete} = (t_{start}, t_{start+1}, \dots, t_{end})$.

Each chunk in the chunk map can be expressed as a subset of i sequential bitmap transactions. If there is a chain of sequential bitmap transactions $C = (t_0, t_1, \dots, t_i)$, $\forall j, j \in [0, i]$, $t_j = 0$, then the chunk that corresponds to the chain C is said to be empty and is not added to the final chunk subset. Otherwise if at least one bitmap transaction is non-zero, then the entire chunk to which the transaction belongs is added to the final chunk subset.

For a typical NTFS or FAT32 filesystem, the bitmap-based strategy will ultimately collect the entire filesystem. If we feed the chunk map into our selective imager the resulting imager will be a fully functional clone of the source filesystem, omitting unallocated space. The Figure 3.2 illustrates a disk layout used in our experiments. The Figure 3.3 illustrates a bitmap capture, while the Figure 3.4 represents the ground truth capture of the same disk.

500GB disk			
327680000(167.8GB)	409600000(209.7GB)	81920000(41.9GB)	152571072(80.7GB)
Windows 8 2.5 years old	Empty	Random numbers	Empty

Figure 3.2. The layout of a 500GB disk with 4 NTFS filesystems.

Comparing the Figure 3.3 with the Figure 3.4, we can see that they are slightly different. In particular, the bitmap capture is missing some deallocated data that is located within the first filesystem (the upper part). Analyzing the chunk maps, we can identify the limits of the filesystems, e.g., chunk 10000 is equivalent to sector 327680000, which corresponds to the end of the first filesystem and beginning of the second one. The second filesystem starts at chunk 10000 and ends at chunk 22500. It does not contain any data other than NTFS system files, therefore the region occupied by the second filesystem is almost completely filled with zeroes. The third filesystem contains a 40GB file filled with random numbers, therefore the entire range of chunks from 22500 to 25000 is filled with data. The last filesystem is analogous to the second filesystem and does not contain user data.

If a disk has valid filesystems then it is most likely that a large part will be captured during bitmap-based analysis. However it is also possible that a disk was reformatted. In this case the bitmap-based strategy will not be able to produce a sound chunk map, because the information about the allocations on the previous filesystem(s) has been destroyed. This raw data discovery is in the scope for the allocation-based routines.

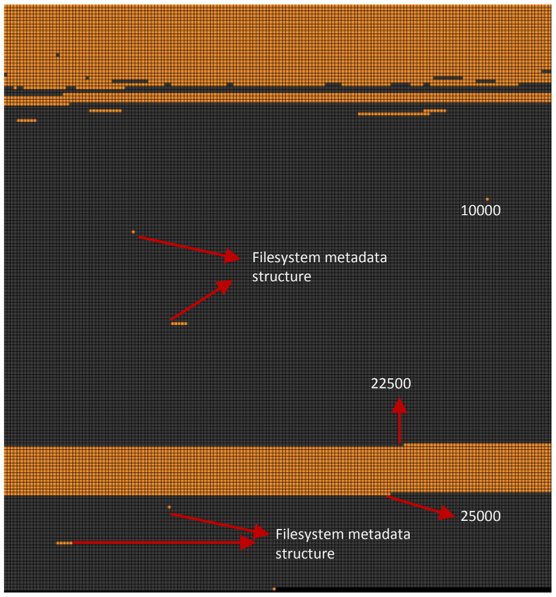

Figure 3.3. The chunk map produced by the bitmap-based strategy for the 500GB disk with four valid filesystems.

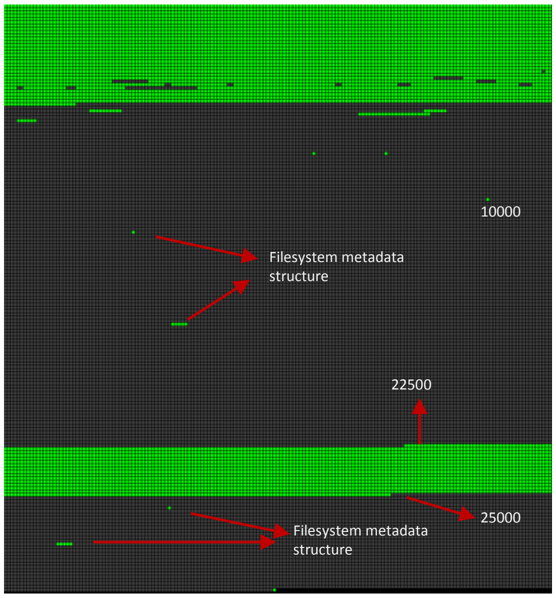

Figure 3.4. The ground truth chunk map for the 500GB disk with four NTFS filesystems.

Allocation-based

Technically, the allocation-based strategy uses the same basis as the bitmap-based strategy. It is walking through the complete bitmap, but evaluates whether regions that are unallocated according to the bitmap may in fact contain non-zero data, corresponding to, e.g., deleted files. Such data composes the subset FS'_u, that has been defined earlier.

In order to identify a data subset FS'_u within an unallocated subset FS_u we use statistical analysis methods, namely *sampling*[11]. The allocation-based analysis is split into two phases: the *coarse bitmap walk (CBW)* and the *fine bitmap walk (FBW)*. The coarse bitmap walk is bitmap size dependent only, whereas the fine bitmap walk also depends on the allocation patterns in a filesystem.

Random Sampling in Bitmap Walking

In general, the idea behind sampling is to evaluate the characteristics of the entire disk without reading it in full, which would defeat the purpose of partial imaging. Therefore we read a small part of a disk, sufficient to identify the data subset FS'_u with an acceptable level of error and overhead. Garfinkel applied random sampling to identify known artifacts and data presence, e.g., whether a disk was properly wiped[14].

The scope of allocation-based analysis is not concerned with file contents, but with the identification of regions that contain data. Simply applying sampling on the unallocated data subset FS_u is not enough for identifying data boundaries. Hence we need a robust way for accurate identification of data regions on a disk. We have employed the random sampling algorithm in bitmap walking procedures. The randomization part is necessary in order to ensure that there will be no easy way to predict where a sample is taken from.

Both phases of the allocation-based analysis have three common parameters that can specified by a user:

1. *Maximum data step size, DS_{max}* -- specifies the size of a disk to be sampled. After the data step is established, it is randomly sampled with one data transaction. If the data transaction contains all zeros, then the entire step is assumed to be zeros. Otherwise the entire step is considered to be non-zero, expressed in terms of chunks, and then added to the chunk map. For CBW, this parameter is static and for the FBW this parameter is dynamic.
2. *Reference data transaction size, DT_{ref}* -- specifies the data transaction size that is read from a data step. This parameter remains fixed for both walking procedures throughout the analysis.
3. *Join distance* - specifies the minimum size in chunks for a zero region. All data gaps not larger than the join distance are added to the resulting partial image.

Figure 3.5 illustrates our bitmap walking procedure and the mapping between bitmap transactions (represented by yellow and light gray squares, where each square is a single bitmap transaction) and data regions (of size *bitmap data transaction size* and represented by blue and dark gray squares). The notations related to the complete bitmap facility are prefixed with "BM". The notations related to the data region facility are prefixed with "D".

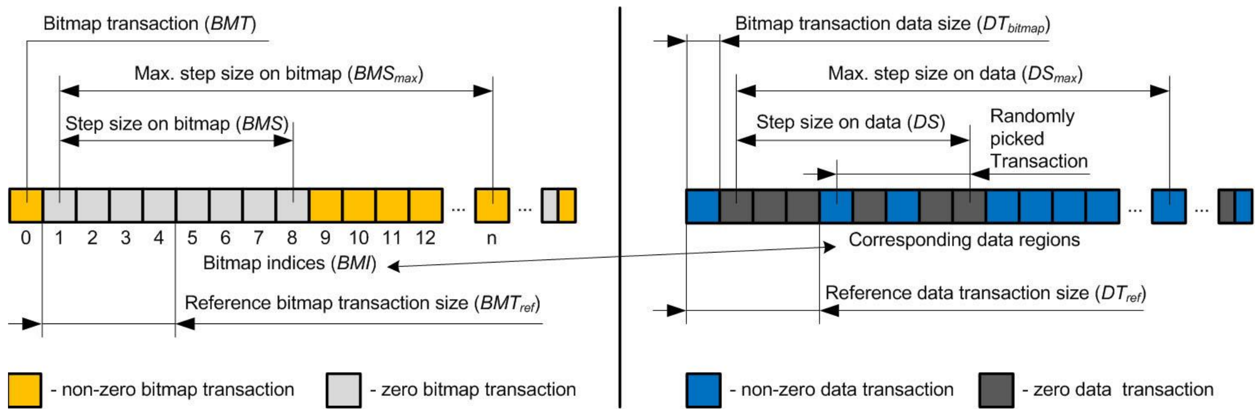
Bitmap transaction (BMT)
Max. step size on bitmap (BMSmax)
Step size on bitmap (BMS)
0 1 2 3 4 5 6 7 8 9 10 11 12 n
Bitmap indices (BMI)
Reference bitmap transaction size (BMTref)
Bitmap transaction data size (DTbitmap)
Max. step size on data (DSmax)
Step size on data (DS)
Randomly picked Transaction
Corresponding data regions
Reference data transaction size (DTref)
- non-zero bitmap transaction
- zero bitmap transaction
- non-zero data transaction
- zero data transaction

Figure 3.5. Description of bitmap walking procedure for allocation-based analysis.

Whereas the complete bitmap is addressed by a bitmap index BMI, the data region is addressed by actual data offsets on the disk. In order to convert a bitmap index into the actual disk offset the bitmap index must multiplied by the bitmap transaction data size, and the offset of the start of the filesystem must be added.

For a *cluster size* as CS, the bitmap transaction data size that corresponds to one bitmap transaction is expressed as: $DT_{bitmap} = 64 \times CS$. The maximum data step size DS_{max} maps onto the complete bitmap where it is called *the maximum bitmap step size BMS_{max}*. For $DS_{max} \geq DT_{bitmap}$:

$$BMS_{max} = \frac{DS_{max}}{DT_{bitmap}} \tag{3.1}$$

In order to sample data on a given step, a chain of zero bitmap transactions must be accumulated. The quantity of sequential bitmap transactions is denoted as the current bitmap step BMS. For the zero chain $C = (t_1, t_2, \dots, t_N)$, where $BMS = N$, there are two situations:

$$N < BMS_{max};\ \forall k, k \in [1, N], t_k = 0;\ t_{N+1} \neq 0 \tag{3.2}$$

$$N = BMS_{max};\ \forall k, k \in [1, N], t_k = 0 \tag{3.3}$$

The scenario described by the expression 3.2 corresponds to the Figure 3.5. The chain of sequential zero bitmap transactions spans the bitmap indices from 1 to 8. Due to encountering a non-zero bitmap transaction $t_9 \neq 0$, the chain of sequential zero transactions breaks, having accumulated only 7 bitmap transactions that are equal zero. Thus, the resulting bitmap step size $BMS = 8 - 1 = 7$, whereas the maximum bitmap step size $BMS_{max} = n - 1$.

The scenario related to the expression 3.3 represents a case when the chain of sequential zero bitmap transactions has reached the maximum bitmap step size BMS_{max} such that $BMS = BMS_{max}$.

Denote the *first bitmap transaction for the current step* as BMI_s. In order to identify the location on the disk from which to take a sample we need to calculate the *data step start offset* DI_S that must correspond to BMI_s:

$$DI_S = DT_{bitmap} \times BMI_s \tag{3.4}$$

Since the current BMS is known, the resulting data step size is calculated as:

$$DS = BMS \times DT_{bitmap} \tag{3.5}$$

Finally, the *data transaction size* DT for the given *data step size* DS is calculated as:

$$DT = MIN(DT_{ref}, DS) \tag{3.6}$$

The data transaction DT may two edge cases:

1. The first case is depicted in the Figure 3.6, where $DT_{ref} < DT_{bitmap}$. The resulting data transaction size in this scenario will be always equal to DT_{ref}.
2. In the Figure 3.7 there is a case when BMS corresponds to DS such that: $DT_{ref} > DS \geq DT_{bitmap}$. In this situation the resulting data transaction size will be equal to DS.

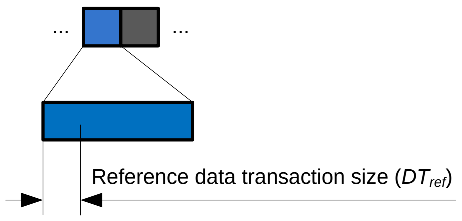

Figure 3.6. Example when $DT_{bitmap} > DT_{ref}$

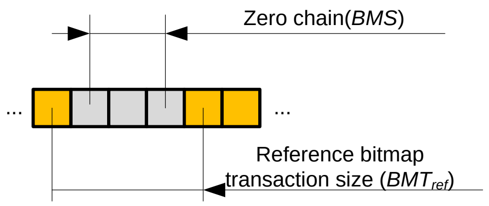

Figure 3.7. Example when $DT_{ref} > DS$

Sampling Parameters

Since the allocation-based analysis uses a statistical approach we have to evaluate the *probability of misclassification a data region as a zero region*, P_{miss}. The sampling accuracy for a given step i depends on *current data step size* DS_i, the *current data transaction size* DT_i, and the *current data density*, Φ_i, where $\Phi_i \in [0, \Phi_{ref}]$, and Φ_{ref} is the *reference data density*.

The reference data density is a data density of a region that contains data. Our imager will encounter a wide variety of data types and hence, allocation patterns. For example a compressed data typically has very high data density, whereas an uncompressed data may have much lower data density. The analysis of data types and their allocation patterns are beyond the scope of this research, and we are striving to choose parameters that work well in common scenarios Therefore we assume that the reference data density on a region that contains data is a fixed value $\Phi_{ref} = 0.1$. Thus any sector within a data region has the probability Φ_{ref} that it is not completely filled with zeroes, e.g. at least one bit within the sector is set to 1.

Throughout the analysis any given step i may span the amount of non-zero data transactions that is smaller than the data step size, thus forming data steps with different density Φ_i. In the Figure 3.8 there is an example of possible data transaction layouts within a data step of a fixed size.

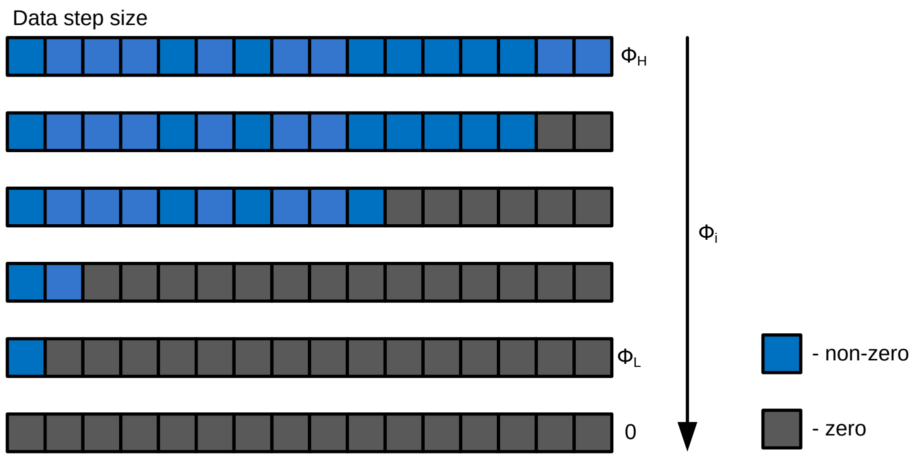

Figure 3.8. Data density for a fixed data step size.

In the Figure 3.8 Φ_L corresponds to a low data density, and Φ_H represents a high data density. Each square represents one data transaction. Each blue square is a non-zero data transaction and is assumed to have the data density Φ_{ref}. Each dark-gray square is a zero data transaction and has data density 0.

The layout of data transaction within a given data step is not known and assumed to be random. Thus if we randomly read one data transaction from within a data step of size DS_i with n non-zero data transactions, the resulting data density is expressed as:

$$\Phi_i = \Phi_{ref} \frac{n}{DS_i} \qquad 3.7$$

There are two edge cases for the data transaction layout. If there is a transition from a data region to a zero region then the layout within the data step forms the *falling edge*. If there is a transition from a zero region to a data region then the layout within the data step forms the *rising edge*. In the Figure 3.9 there are examples of rising and falling edges.

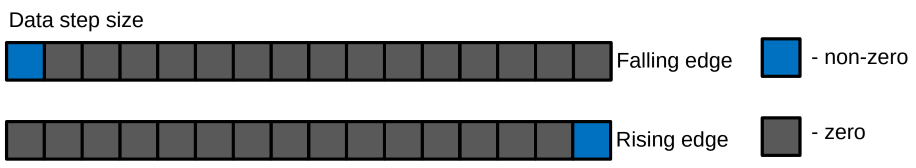

Figure 3.9. Example of a falling and a rising edges.

In reality a data step size can include thousands of data transactions. Generally speaking, if data within a step is clustering at the beginning of the step, then the step represents the falling edge. If data within a step is clustering towards the end of the step, then the step represents the rising edge. The edges tend to have very low data density therefore they accuracy on them is lower than the accuracy on a solid data region.

Due to the fact that the CBW runs with a fixed data step size, it might not precisely capture the falling and rising edges. The edges are not dangerous within solid data regions as all the gaps that are not larger than the join distance will be smoothened during the chunk map post-processing. However the problem arises when a data region ends and the large zero region begins, or vice versa, a large zero region ends and data region begins.

The *data transaction size* DT_i also affect the accuracy of the sampling. After a data transaction has been read from within a given step i, the chances that all the sectors within the transaction are equal to zero depend on the data density Φ_i as well as on the data transaction size itself. From the expression 3.7 we calculate the data density Φ_i and from the expression 3.6 we can find out the size of the data transaction for the step i. A sector is considered empty if and only if all the bits within its limits are equal to 0, otherwise the sector is said to be non-empty. Thus the sectors within the data transaction of size DT_i distributed according with the binomial distribution law[18]:

$$P = \frac{n!}{(n-k)! \times k!} \times p^k \times (1-p)^{n-k} \tag{3.8}$$

The probability of success $p = \Phi_i$, the probability of an fail $q = (1-p)$, the number of trials $n = DT_i$, and the number of expected successes $k = 0$. Plugging the parameters into the expression 3.9, the probability of not finding at least one non-empty sector within the data transaction is calculated as:

$$P_{miss} = (1 - \Phi_i)^{DT_i} \tag{3.9}$$

Optimal Values for Sampling

According to the expression 3.7, the data density is inversely proportional to the data step size. From the expression 3.9, smaller *maximum data step sizes* DS_{max}, decrease the probability of misclassification a data region as a zero region, P_{miss}. In order to determine appropriate range of values for the maximum data transaction sizes, that do not overstress a source disk with high volume I/O load yet produce the results within reasonable amount of time, we have evaluated the impact of read operations on a hard disk performance. We randomly distributed disk offsets, sorted them, and performed read operations from those locations such that the total amount of data to read was 0.48% of the size of the disk. The results are presented in the

Table 3.2. The first column describes the transaction size, the second column shows the number of transaction on the disk, the third column expresses the total amount of read operations, and the fourth column is the time taken to complete all the read operations.

Table 3.2. Random sampling with a fixed sample size and different transaction sizes.

Total on disk: 71677952 sectors (36.7GB). The sample size = 0.48%			
Transaction size, sectors	Total transactions on disk	Number of read operations	Read time, sec
1	71677952	344054	201.97
2	35838976	172027	199.64
4	17919488	86013	178.9
8	8959744	43006	131.78
16	4479872	21503	84.71
32	2239936	10751	49.74
64	1119968	5375	27.83
128	559984	2687	16
256	279992	1343	8.74
512	139996	671	5.41
1024	69998	335	3.07
2048	34999	167	2
4096	17499	83	1.5
8192	8749	41	1.22
16384	4374	20	1.09
32768	2187	10	1.02

The smaller the amount of read operations the less it takes to read the outlined sample. However If instead of the relative sample size we use a fixed amount of data steps that must be traversed in order to complete the sampling the results look a bit different. The results of reading a fixed sample of 1000 transactions is provided in the Table 3.3.

Table 3.3. Random sampling with a fixed amount of transactions for different transaction sizes.

Sample size: 1000 transactions		
Transaction size, sectors	Amount data to read, sectors	Read time, sec
1	1000	5.84
2	2000	5.99
4	4000	5.88
8	8000	6.07
16	16000	6.06
32	32000	6.06
64	64000	6.01
128	128000	6.3
256	256000	6.61
512	512000	7.9
1024	1024000	9.28
2048	2048000	11.92
4096	4096000	17.17
8192	8192000	27.24
16384	16384000	47.42
32768	32768000	88.84

The first column defines the transaction size, the second column shows the amount data to read, and the third column shows the time taken to read 1000 transactions.

Based on the results from the

Table 3.2 and the Table 3.3, we have empirically determined the range of appropriate values for the reference data transaction size (highlighted as light-green): 128, 256, 512, 1024, and 2048 sectors, which corresponds to 64KB, 128KB, 256KB, 512KB, and 1MB respectively. The given values do not over stress the disk with excessive amounts of read operations. Smaller data transaction sizes can be used with small data step sizes, and larger data transaction sizes can be used with large data step sizes.

For the chosen data transaction size values range we have evaluated the *probability of not finding a single non-empty sector within a data transaction, P_{miss}*. The result of the evaluation is presented in the Figure 3.10.

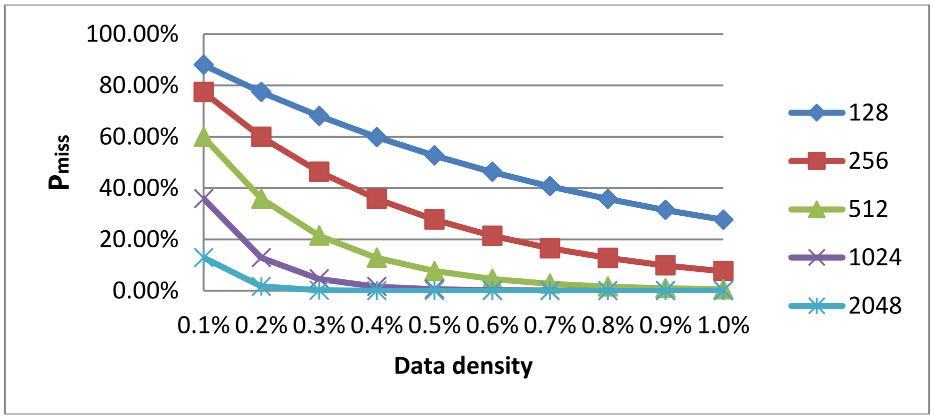

Figure 3.10. Probability of misclassification a data region as a zero region for different data density and data transaction size values.

For an average data density $\Phi = 1\%$, and the data transaction size $DT = 64KB = 128\ sectors$, P_{miss} ≈30%. After the analysis phase has completed, the chunk map undergoes the post-processing which eliminates all small gaps of zero regions based on the specified join distance value. The join distance spans multiple data steps N, where each data step has a certain P_{miss} value. Thus the *probability of not finding a single non-zero data step within the entire region P'_{miss}* equals to the average of the probabilities on each step:

$$P'_{miss} = \frac{\sum_{i=1}^{N} P_{miss}^{i}}{N} \qquad 3.10$$

Using the expression 3.8 we can evaluate the probability of failure, P_{fail} for which our approach fails to locate a region on the disk filled with data. Assume the probability of success $p = (1 - P_{miss})$, the probability of failure $q = P_{miss}$, the number of trials $n = join\ distance/DS_{max}$, and the number of expected successes $k = 0$. The parameters used are: $DS_{max} = 1GB$ (the maximum value that our tool can accept) and the join distance is 500 chunks. The evaluation of probability of failure P_{fail} is presented in the Figure 3.11.

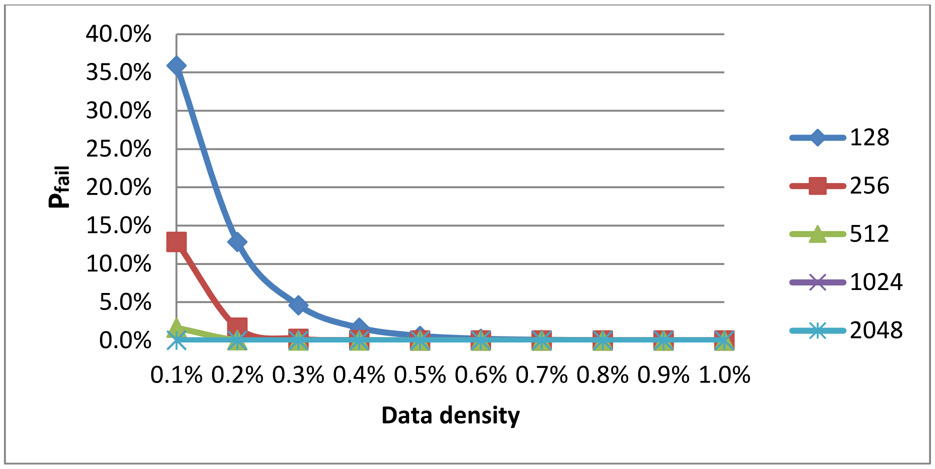

Figure 3.11. The probability of failure of the chunk map post-processing.

For an approximate data density $\Phi > 0.5\%$, $DS = 128MB$, and the join distance of 500 chunks, the chunk map post-processing makes the probability P_{fail} to be negligibly small even for small DT values.

Although the chunk map post-processing increases the accuracy at the cost of reasonable overhead it is not effective for discovering small non-zero areas lying in within large regions of disk filled with zeroes. Specifying a shorter maximum data step size may increase the chances for catching small data regions, but the chunk map post-processing is also inefficient in detecting the borders between data-filled and zero-filled regions. Such regions are in the scope of the FBW.

Fine Bitmap Walk

After the CBW has completed, the FBW starts. Unlike the CBW, FBW has a dynamic maximum data step size DS'_{max}, where $DS'_{max} \in [MAX(DT_{bitmap}, DT_{ref}),\ DS^{FBW}_{max}]$. The main feature of the FBW is that it determines transitions between non-zero and zero bitmap/data transactions, and adjusts DS'_{max} accordingly. The FBW uses an additional parameter called the *shift threshold* ST. The shift threshold defines the minimum amount of zero data transactions to pass in order to shift up DS'_{max}. If the FBW internal counter that counts zero data transactions is denoted as CNT, then the DS'_{max} can be expressed as:

$$DS'_{max} = MAX\left(DT_{bitmap},\qquad DT_{ref} \times 2^{\frac{CNT}{TS}}\right) \qquad 3.11$$

Each time the DS'_{max} is changed the BMS_{max} is recalculated via the expression 3.1. The FBW uses an exponential increment, which allows to quickly pass large zero regions at high DS'_{max}. Using a large shift threshold TS ensures slow step increase which allows to more precisely examine falling and rising edges at low DS'_{max}. There are three major cases when the FBW takes an action to adjust DS'_{max}:

1. Transition from a non-zero bitmap transaction to a zero bitmap transaction (capturing falling edges). In this case the FBW simply drops the speed at its minimum: $DS'_{max} = MAX(DT_{bitmap}, DT_{ref})$.
2. Transition from a zero bitmap transaction to a non-zero bitmap transaction (capturing rising edges). If the $DS'_{max} > MAX(DT_{bitmap}, DT_{ref})$, then the FBW backs off by the amount of bitmap transactions that correspond to the last taken data step. The speed is dropped to its minimum value: $DS'_{max} = MAX(DT_{bitmap}, DT_{ref})$. The region is re-walked. If the $DS'_{max} = MAX(DT_{bitmap}, DT_{ref})$, then simply proceed to the next bitmap transaction.
3. Transition from a zero data transaction to a non-zero data transaction. In this case the data offset translates into the chunk number and the chunk is added to the chunk map. The rest is analogous to the case 2.

It is possible that the FBW backs off at the same bitmap transaction more than one time. For example if the FBW has increased to a certain data step size, the first back off would adjust the bitmap index according to the last taken step. By the time the FBW has reached the bitmap transaction that caused the back off, the data step size may have already increased and is more than the minimum value, thus the back off occurs again.

We evaluated the efficiency of the allocation-based strategy by wiping out the layout of the 500GB disk discussed previously, and trying to identify data region with the allocation-based strategy only. We used the following parameters: $DS_{max} = 32MB, DT_{ref} = 512KB, join\ distance = 500, TS = 100$. The Figure 3.12 shows the chunk map produced during the allocation-based analysis. The Figure 3.13 represents the ground truth chunk map.

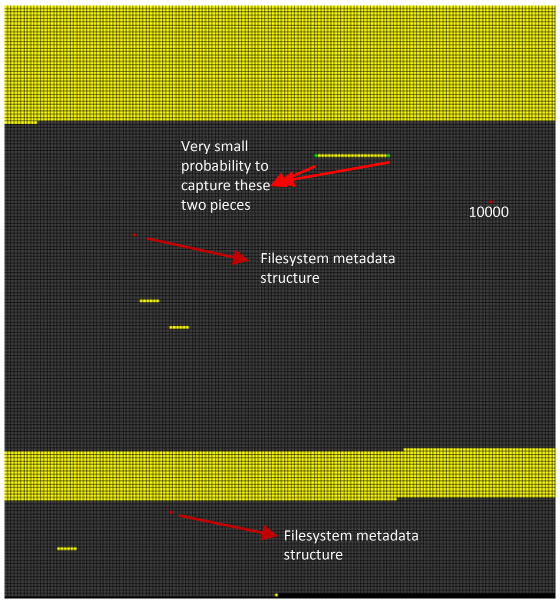

Figure 3.12. The chunk map produced by the allocation-based strategy for a 500GB disk without valid filesystems on it.

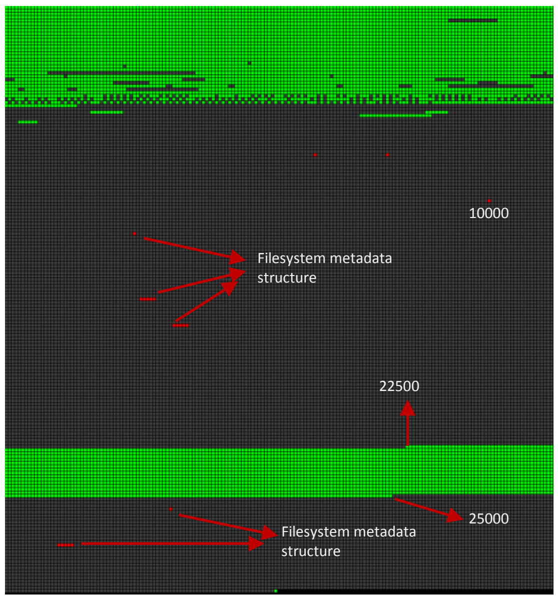

Figure 3.13. The ground truth chunk map for a 500GB disk. Red chunks represent the data that are located within the large zero region.

We performed more thorough examination of the not captured chunks: 7609, 7631, 10000, 11562, and 25601, in order to understand why they are difficult to capture using our approach.

- The chunk 7609 contains only two identical copies of a boot sector.
- The chunk 7631 contains four copies of the same boot sector.
- Running strings on chunk 10000 revealed that it contains two boot sector, back to back and an $MFT. The first boot sector is the last sector of the first volume, the second is the first sector of the second volume. The megabyte of sporadic data is the $MFT for the second NTFS filesystem (see Figure 3.6).
- The chunks 11562 and 25601 contain similar data pattern. They both are parts of the filesystem that were previously defined on the disk. We randomly sampled both chunks 100 times with 1MB sample size. Only 18 hits were discovered. The density on those chunks is considered very low. The chunks 11562 and 25601 are in the scope of the bitmap-based analysis.

For the rest, our allocation-based approach successfully accomplished its goal and composed a very accurate chunk map.

Experiments

For all the experiments the following parameters were used: $join\ distance = 500, TS = 100$. *We have conducted the experiment on the 500GB disk without valid filesystems for different step sizes and transaction sizes in order to evaluate the time taken to complete the analysis. The results are presented in the*
Table 3.4. The first column is the data transaction size, the second column is the maximum data step size, and the third column describes the number of chunks in the final chunk subset. The fourth and fifth columns show false positives and false negatives, respectively. The sixth column expresses the overhead calculated as the ratio of the false positives to the size of the entire disk in chunks. The seventh column displays the time taken by the allocation-based analysis. The eighth and the ninth columns show the number of read operations performed during the CBW and FBW phases respectively. The tenth column shows the total amount of data read during the allocation-based analysis. The last column shows the imaging increase ratio.

Table 3.4. Results of the allocation-based analysis of 500GB disk without filesystems.

Ground truth chunk map size: 7260 Disk size in chunks: 29809

DT_{ref}	DS_{max}	Chunk subset	FP	FN	OH	Time	CBW read, ops	FBW read, ops	Read, Mb	Boost
65536	32	8372	1117	5	4%	183	14904	8756	1479	3.56
65536	64	8372	1122	10	4%	111	7452	7468	933	3.56
65536	128	8369	1139	15	4%	73	3726	6260	624	3.56
65536	256	8391	1141	10	4%	65	1863	7684	597	3.55
65536	1024	8423	1178	15	4%	52	466	7646	507	3.54
131072	32	8373	1118	5	4%	254	14904	7785	2836	3.56
131072	64	8379	1124	5	4%	103	7452	7653	1888	3.56
131072	128	8376	1126	10	4%	110	3726	6553	1285	3.56
131072	256	8363	1123	20	4%	54	1863	3117	623	3.56
131072	1024	8359	1119	20	4%	63	46	6531	822	3.57
262144	32	8376	1121	5	4%	288	14904	6854	5440	3.56
262144	64	8382	1127	5	4%	177	7452	6459	3478	3.56
262144	128	8376	1126	10	4%	117	3726	5564	2323	3.56
262144	256	8378	1133	15	4%	107	1863	7487	2338	3.56
262144	1024	8359	1119	20	4%	56	466	4403	1217	3.57
524288	32	8400	1143	3	4%	353	14904	7295	11100	3.55
524288	64	8378	1123	5	4%	214	7452	5881	6667	3.56
524288	128	8375	1125	10	4%	148	3726	4809	4268	3.56
524288	256	8368	1123	15	4%	95	1863	4066	2965	3.56
524288	1024	8365	1120	15	4%	114	466	5080	2773	3.56
1048576	32	8377	1121	4	4%	465	14904	5663	20567	3.56
1048576	64	8375	1125	5	4%	284	7452	5143	12595	3.56
1048576	128	8388	1136	4	4%	216	3726	5840	9566	3.55
1048576	256	8378	1133	15	4%	155	1863	5647	7510	3.56
1048576	1024	8382	1142	20	4%	101	466	4416	4882	3.56

The common part among all the results presented in the
Table 3.4 is that the speed increase is approximately 3.5 times. A complete image of the analyzed disk takes 5920 seconds, thus the partial image is created in approximately 1670 seconds, a savings of more than an hour, even on a relatively small disk. The overhead is consistent across the all runs and is within acceptable range.

We outlined several groups of data, based on the number of false negatives.

1. The group with the least amount of FN (highlighted as green). There is only one run with three false negatives. The following analysis parameters were used: $DT_{ref} = 512Kb, DS_{max} = 32Mb$. This run

completed in 353 seconds, which is higher than the average. Because the FBW discovered the chunks 7609 and 7631, it had to drop the speed and speed up again, which caused longer processing time. The chunks 7609 and 7631 have very low data density, therefore they are not consistently discovered across multiple runs.

2. The group with $FN = 4$ (highlighted as light-green). There are two entries with four false negatives. The first has parameters $DT_{ref} = 1Mb, DS_{max} = 32Mb$, the second has parameters $DT_{ref} = 1Mb, DS_{max} = 128Mb$. The both cases have identified chunk 26501, yet couldn't capture the chunks 7609 and 7631 that were discovered by the first group. On the other hand the first group is missing chunk 25601.
3. The group with $FN = 5$ (highlighted as yellow). The given group is large and spans all runs with $DS_{max} = 32Mb$, and four of the five runs with the analysis parameters $DS_{max} = 64Mb$. The group is missing the chunks that have very low density and are located in the middle of the large zero regions. We have discussed these chunks earlier, when analyzed results of the allocation-based strategy.
4. All the remaining runs have missed more than 5 chunks. The following series of chunks were overlooked:
    - 14904, 14905, 14906, 14907, 14908.
    - 16250, 16251, 16252, 16253, 16254.
    - 27404, 27405, 27406, 27407, 25408.

    The density of the aforementioned chunks is moderate, but due to large DS_{max} values, the cumulative data density on the data steps that span the sequences of the chunks is very low.

The average analysis time across the highlighted groups took approximately 215 seconds.

Despite the fourth group produced more false negatives, it still accurately identified large data regions. For example if an investigator wants to find out whether it worth to do selective imaging on a disk with an unknown content, he can specify large the maximum data step size $DS_{max} = 1024Mb$ and DT_{ref} between 64Kb and 256Kb. In this case the analysis would complete faster than if it was running with small step sizes. The resulting chunk map could be interpreted via the *chunk map visualizer* into a PNG picture, so the investigator could evaluate the disk layout and decide whether it needs to be further analyzed, or not. If the resulting chunk map has large zero regions then the selective imaging renders suitable. If the resulting chunk map is fully marked than the disk most likely has very dense data allocation patterns. It can be rescanned with smaller *join distance* values, in order to lower the amount of false positives. If a disk has a dense allocation pattern it is a subject for complete imaging.

The last part of the experiment is to run the full analysis on the 500GB disk with valid filesystems, and on a 2TB disk with two valid filesystems. The results for 500GB disk with four valid NTFS filesystems are presented in the Table 3.5. The results for the 2TB disk with two valid filesystems are provided in the Table 3.6.

Table 3.5. Results of the allocation-based analysis of 500GB disk with 4 valid NTFS filesystems.

Ground truth chunk map size: 7526 Disk size in chunks: 29809

DT_{ref}	DS_{max}	Chunk subset	FP	FN	OH	Time	CBW read, ops	FBW read, ops	Read, MB	Boost
65536	64	8370	846	2	3%	44	5661	8899	910	3.6
65536	1024	8370	846	2	3%	17	375	8899	580	3.6
131072	64	8370	846	2	3%	42	5661	8899	1820	3.6
131072	1024	8370	846	2	3%	17	375	8899	1159	3.6
262144	64	8370	846	2	3%	42	5661	8899	3640	3.6
262144	1024	8370	846	2	3%	18	375	8899	2319	3.6
524288	64	8374	849	1	3%	41	5661	8579	7120	3.6
524288	1024	8370	846	2	3%	19	375	7933	4154	3.6
1048576	64	8371	846	1	3%	47	5661	7479	13140	3.6
1048576	1024	8370	846	2	3%	20	375	6787	7162	3.6

The results of analyzing the 500GB disk with valid filesystems vs. analysis as raw data is almost identical for all the parameters that we tried. In particular, the false negatives are still the same as for the raw disk (7609, 7631). The major difference between the analysis of the raw version of the 500GB disk and the version with valid NTFS filesystems is that the time taken to analyze the second disk is noticeably lower. Also, the number of reads during the CBW is much lower, because it didn't have check the regions captured during the bitmap-based analysis. The longest analysis time for a precise run with the parameters for $DT_{ref} = 1MB, DS_{max} = 64MB$ is 47 seconds. The quick run with the parameters for $DT_{ref} = 64KB, DS_{max} = 1GB$ completed in 17 seconds.

The bitmap capture for this experiments well as the ground truth chunk map were presented earlier in the Figure 3.3 and the Figure 3.4 respectively. The resulting chunk map for the given experiment is presented in the Figure 3.14.

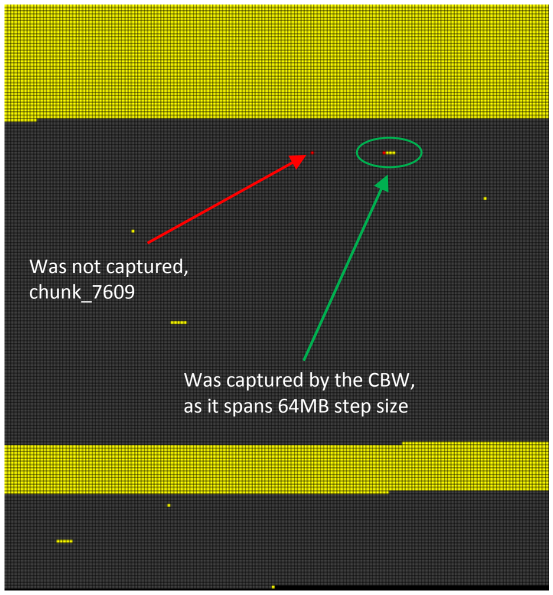

Figure 3.14. The chunk map produced by the allocation-based strategy for 500GB disk with 4 valid filesystems.

The analysis results for the 2TB disk are interesting because this disk is an ordinary disk that has been in use for approximately one year. It was reformatted once, and it had massive deletions. The bitmap-based strategy was able to capture only the allocated data subset, whereas the allocation-based analysis discovered the unallocated data lingering on the disk. The chunk map produced by the bitmap-based analysis is shown in the Figure 3.15. The ground truth bitmap is shown in the Figure 3.16. The resulting chunk map produced during the disk analysis is presented in the Figure 3.17.

Table 3.6. Results of the allocation-based analysis of 2TB disk with 2 valid NTFS filesystems.

Ground truth chunk map size: 51043 Disk size in chunks: 119234

DT_{ref}	DS_{max}	Chunk subset	FP	FN	OH	Time	CBW read, ops	FBW read, ops	Read, Mb	Boost
65536	64	51868	827	2	1%	244	21912	5350	1704	2.3
65536	1024	51867	826	2	1%	65	1400	5365	423	2.3
131072	64	51867	826	2	1%	377	21912	5361	3409	2.3
131072	1024	51867	826	2	1%	144	1400	16203	2200	2.3
262144	64	51867	826	2	1%	414	21912	5361	6818	2.3
262144	1024	51867	828	2	1%	154	1400	16251	4413	2.3
524288	64	51867	826	2	1%	489	21912	4830	13371	2.3
524288	1024	51924	883	2	1%	108	1400	4724	3062	2.3
1048576	64	51867	826	2	1%	634	21912	4129	26041	2.3
1048576	1024	51867	826	2	1%	255	1400	8988	10388	2.3

Analyzing the Table 3.6, we see that all the runs share some common outcomes, specifically, the false negative rate, false positive rate, overhead, and the speed increase of 2.3 times. The longest analysis time a for precise run with the parameters for $DT_{ref} = 1MB, DS_{max} = 64MB$ is 634 seconds. The quick run with the parameters for $DT_{ref} = 64KB, DS_{max} = 1GB$ completed in 65 seconds. The scan with the moderate parameters $DT_{ref} = 256KB, DS = 64MB$ took 414 seconds. As it can be seen comparing the Figure 3.15 with the Figure 3.16, our 2TB has very large unallocated region that contains data.

The two false negatives, namely chunks 28369 and chunk 87982, are located in the middle of large zero region which makes the data density on a given step to be very small. The data density of the chunks themselves was also very small, 2-4 sectors per 16MB chunk.

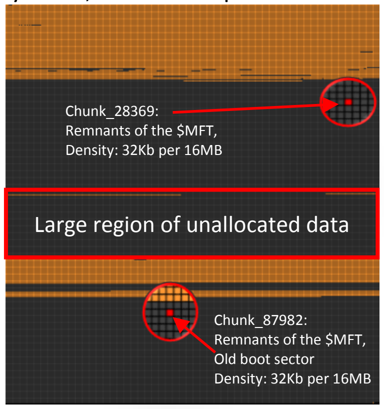

Figure 3.15. The bitmap-based chunk map for 2TB disk with two valid NTFS filesystems

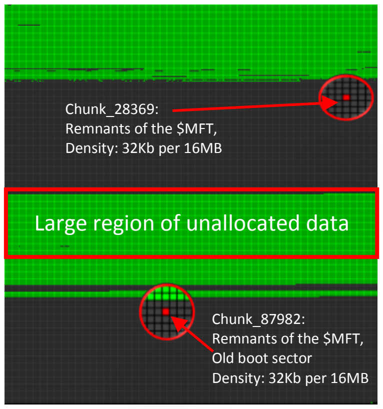

Figure 3.16. The ground truth chunk map for 2TB disk with two valid NTFS filesystems.

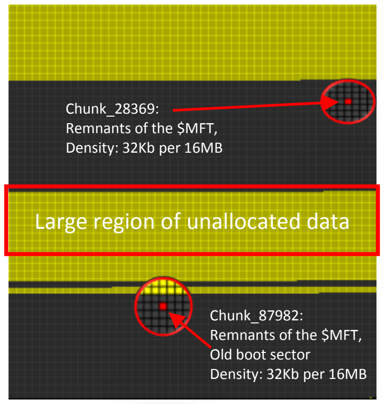

Figure 3.17. The allocation-based chunk map for 2TB disk with two valid NTFS filesystems.

Having analyzed the 2TB disk we discovered that more than 50% of its space consists of zeros, which should not be captured during the imaging. In the ideal case, when there is an appropriate hardware capacity and the disk works at high bitrates of approximately 150MPbs, the imaging time for a 2TB would about 4 hours. Even if we ran the longest analysis it would still decrease the imaging time by approximately two hours. However if a storage device is an old USB2.0 external hard disk with 2TB space, the imaging would proceed at the speed of approximately 40MB/s, which would turn into almost 14 hours. Of course the analysis of slower disks will take more time as well. Our strategy in this case would be to run a quick scan to see if there is any data on it other than the legitimate allocated subset. After the analysis has completed the tool produces the subsets for each of the phase that took place during the disk examination.

An investigator may use either the *chunk map visualizer* in order to interpret the chunk maps as a picture, and evaluate them visually, or as an alternate method, he can use the *chunk map viewer,* which has multiple functions, one of which is to compare any two chunk maps produced for the same disk.

In any event, even if the investigator spends a modest amount of time performing disk analysis, if this drastically reduces the imaging time, then a large amount of time can be saved.

Chapter 4

Selective Imager

After the final chunk map has been created, the data must be acquired from the source disk (based on the chunk map) and stored on a destination drive in an efficient format. As a part of the research, we developed a high performance selective imager which creates *AFFv3* format images. While some attention is being generated around AFFv4, we used AFFv3 it has much wider support among commonly used digital forensics tools. Images produced with our selective imager can therefore be readily analyzed by the majority of other existing digital forensics tools, without modification.

AFFLIBv3 Constraints

AFF is a well-known and widely adopted digital forensics image format and AFF offers attractive features, such as maintaining attributes and metadata for an acquired image. However the AFFLIBv3 cannot be used by our selective imager straight out of the box. The major drawback of the AFFLIBv3 library is that it is not optimized for concurrent data flow. When a chunk of data is written to a AFF file, the data is first compressed and hashed, unless compression and/or hashing are turned off, which is typically not the case, and only after all formatting operations have completed is it written to the output file in the form of an AFF page. It is clear that such an approach leads to inefficient resource utilization. Specifically, the destination storage device must always wait on the CPU to finish compression and hashing prior to the data being written to the AFF image.

The AFFLIBv3 supports two compression algorithms: *deflate*, *LZMA*, and three hashing algorithms: *MD5*, *SHA1* and *SHA256*. For compression used only the *deflate* algorithm with compression level 1, as higher compression levels impose larger compression overhead. The compression is the most intensive task, consuming by far the most CPU time. Nevertheless hashing also impacts the aggregated throughput. To assess the performance of the AFFLIBv3 implementation, we used deflate level 1 for compression and MD5 for signing data. The test platform was a desktop with a Intel i7 processor running at 4.7GHz and a PlextorM5 Pro SSD as the source disk. As a destination disk we used an HDD 7200RPM and an SDD Crucial MX100. The source disk contained a Windows 8 filesystem which was 70% full with user data. With a sufficiently fast destination disk, the throughput for the original AFFLIBv3 implementation was approximately 60-80 MB/s, which is inadequate for high-performance imaging.

AFFLIBv3 Tuning: Selective Imager Design

In order to relax the constraints imposed by the original implementation of the AFFLIBv3, we have implemented our selective imager in the way that it outsources the formatting of the data from the AFFLIB onto itself. The design of the imager allows us to fully utilize available CPU power for compressing and hashing the chunks of data by parallelizing the formatting procedure. The design of the selective imager is presented in the Figure 4.1.

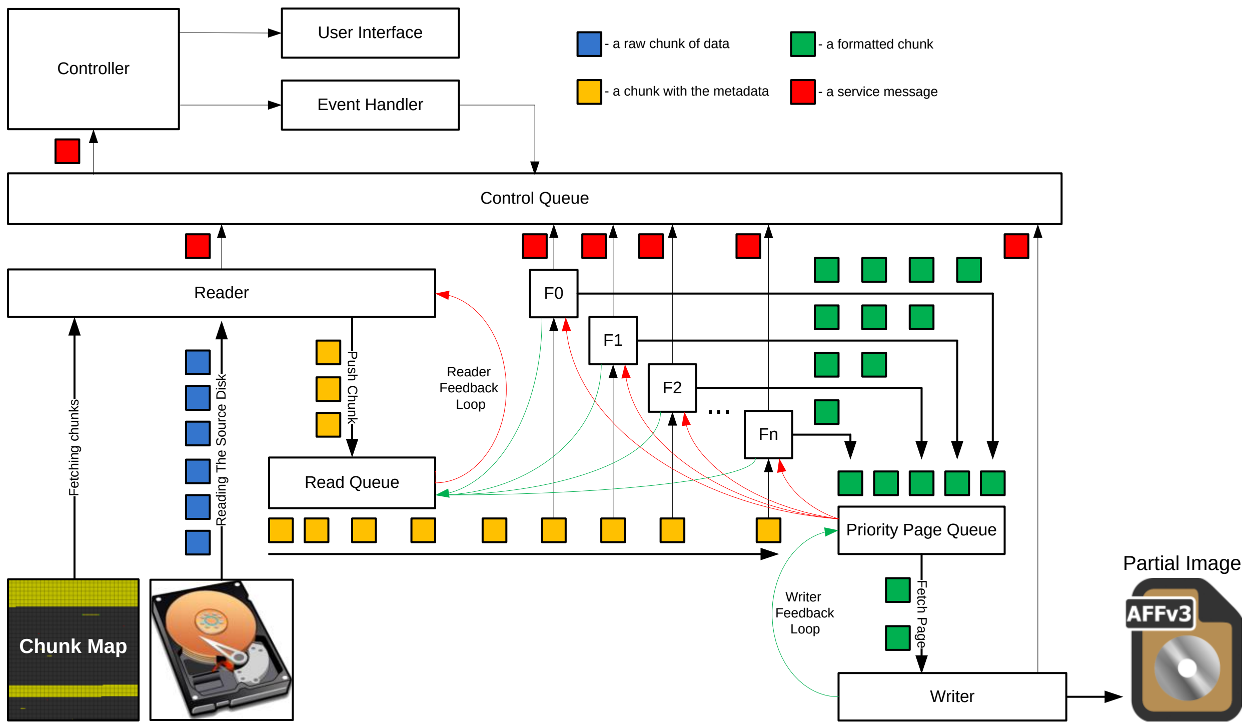

Figure 4.1. The design of the selective imager.

The imager has three major components:

- *The Reader* - iterates through the chunk map and for each set of chunks, it reads the data from the corresponding disk location. After a chunk has been read, metadata, i.e., chunk number and ordinal number, is associated and the chunk is pushed into the *Read Queue*. If the *Read Queue* reaches the size limit, it notifies the reader via the *Reader Feedback Loop* that it needs to wait until the *Read Queue* is unloaded and it is notified by the formatters (curly green arrows).
- *The Formatter* - queries the *Read Queue* and fetches chunks with their metadata. The *Formatter* performs compression and hashing according to parameters set by the user, and pushes the formatted chunk with its metadata into the *Priority Page Queue*. The system may have multiple formatters, depending on the number of CPU cores. The *Priority Page Queue* notifies the *Formatters* if it is full (curly red arrows). The *Formatters* wait until the *Writer* unloads the queue so they can proceed.
- *The Writer* - queries the *Priority Page Queue* and fetches formatted chunks in-order based on the order they have been read from the disk. When a formatted chunk is fetched it already has the compressed buffer and the hash signature(s) that are supplied via AFFLIBv3 facilities, which were modified to support the external formatting. After the imaging is completed the writer has produced a partial image in AFFv3 format, which is fully functional and can be readily consumed by other existing digital forensics tools.

The imager, due to its concurrent nature, maintains its state in a centralized manner. All the asynchronous modules, e.g., the *Reader,* the *Formatters,* and the *Writer*, transmit their states via the *Control Queue*. The controller monitors the state, maintains the user interface, and in case of an error, gracefully terminates all the on-going routines, displays a description of the error, and exits the imager.

Throughout the tuning of the AFFLIBv3 we also discovered an interesting fact, specifically, that write performance was degrading over time with the growth of AFF pages in the image. The problem was caused by the page placement policy the standard AFFLIBv3 uses to accommodate the next AFF page. The more pages an image contains, the longer time it takes to iterate through all of them to check whether there is enough space to fit the current page. We did not change the allocation policy, but we optimized it such that if a new image is created, then there is no need to check whether there is a space in-between the pages. As the AFF image grows, the pages are therefore allocated contiguously. For incremental imaging, the AFFLIBv3 will try to find a hole that would fit the current page, but once the previous AFF image size has been exceeded, the hole search ceases, and the imaging will proceed without the placement policy overhead.

Selective Imager Performance Evaluation

We performed an evaluation to assess the performance of our imager. As a source disk, we used a 500GB Samsung EVO 850 SSD, attached to the same system we used to evaluate the original AFF implementation. We created four partitions on the source disk. The first partition contained the installation of Windows 8, extended to 167.8GB. The second and fourth partitions were allocated 209.7GB and 80.7GB respectively, which were just fresh NTFS filesystems without any user files. The third partition contained 41.9GB of random data. The performance chart is presented in Figure 4.2.

For sufficiently powerful computers, the bottleneck will generally be source disk read speed, unless the source disk is an SSD with high transfer rates. In that case, performance can be limited by CPU speed or the write speed of the destination device. For computers with weak CPUs, the overhead will mostly be imposed by formatting, which includes compression. The write speed of a destination storage device will likely be a problem only when dealing with high entropy data, which resists compression. Other than this case, write speed is not really a problem, as the compressed data is substantially more compact and does not overload the I/O interface.

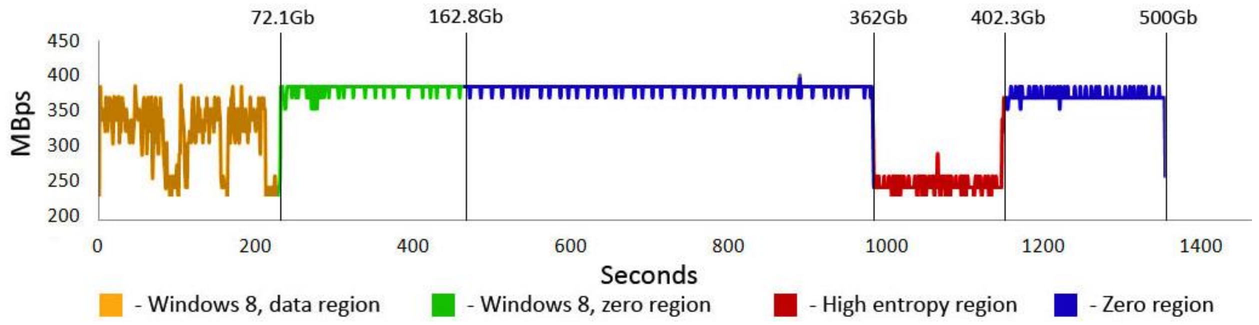

Figure 4.2. The imaging performance chart for 500GB SSD with different types of content.

Average imaging speed on the data region (the orange part) is 316 MB/s. On zero regions (blue parts) the speed is bottlenecked by the maximum speed of the SSD. On the high entropy region (red part) of the plot, the performance is bottlenecked by the CPU. The spikes are caused by the logging facility, as the result is being updated every few seconds.

Chapter 5

Conclusion

The volume challenge severely impedes forensic investigations by imposing delays during the imaging process that impact the ability of investigators to perform investigations in a timely manner. In the present research, we have developed the disk analysis techniques that accurately locate regions on disk that contain potentially important data. Using multiple analysis strategies in conjunction we are able to analyze a disk within a few minutes and yet reduce the imaging time of large storage media by many hours. Our imager produces AFFv3 images, while relaxing some of the performance issues present in the original AFFv3 implementation, all while maintaining compatibility with existing tools that support the AFF image format. The proposed approach can be easily incorporated into the workflow of practicing forensics investigators.

Limitations and Future Work

Analysis of the solid state drivers might require alternate techniques for maximum efficiency, since SSDs might utilize features such as TRIM and garbage collection. King and Vidas discussed in detail the retention properties of SSDs from different vendors[19].

The AFFLIBv3 uses the compression algorithms that provide high compression ratio at the cost of time. Since the storage is cheap and available, it would be more efficient to use high-performance compression algorithms, e.g., snappy [20], which would give higher bitrates with lower amount of the CPU cores.

For the future directions, we have outlined the support for other popular filesystems, e.g., the ext filesystem family, HFS+, and exFAT. Also we look forward to implement the entire tool kit with intuitive front end interface. In the future work we may consider to incorporate other image formats that support high performance compression algorithms.

References

[1] "dd (Unix)," [Online]. Available: https://en.wikipedia.org/wiki/Dd_(Unix). [Accessed 31 05 2016].

[2] "Forensic Tool Kit (FTK)," Access Data, [Online]. Available: http://accessdata.com/solutions/digital-forensics/forensic-toolkit-ftk. [Accessed 31 May 2016].

[3] "X-ways," X-Ways Forensics, [Online]. Available: http://www.x-ways.net/forensics/. [Accessed 31 May 2016].

[4] "ADF Solutions," ADF Solutions, Inc., [Online]. Available: http://www.adfsolutions.com. [Accessed 31 May 2016].

[5] S. Garfinkel, D. Malan, K. Dubec, C. Stevens and C. Pham, "Advanced Forensic Format: An Open, Extensible Format For Disk Imaging," *Advances in Digital Forensics,* vol. II, p. Chapter 2, 2006.

[6] S. Garfinkel, "AFFLIBv3," [Online]. Available: https://github.com/simsong/AFFLIBv3. [Accessed 31 May 2016].

[7] P. Turner, "Selective and intelligent imaging using digital evidence bags," in *DFRWS*, Lafayette, Indiana, USA, 2006.

[8] Stuttgen J, Dewald A, Freiling FC, "Selective imaging revisited," in *Seventh international conference on it security incident management and it forensics*, Nuremberg, 2013.

[9] Jonathan Grier, Golden G. Richard III, "Rapid forensic imaging of large disks with sifting collectors," in *DFRWS*, Philadelphia, PA, USA, 2015.

[10] B. Carrier, "Open Source Digital Forensics," [Online]. Available: http://www.sleuthkit.org. [Accessed 31 May 2016].

[11] "Sampling (statistics)," [Online]. Available: https://en.wikipedia.org/wiki/Sampling_(statistics). [Accessed 31 May 2016].

[12] Kloet, Robert-Jan Mora and Bas, "Digital forensic sampling," Almere The Netherlands, 2010.

[13] J. K. Taguchi, "Optimal sector sampling for drive triage," Naval Postgraduate School, Monterey, 2013.

[14] S. Garfinkel, "Fast Disk Analysis with Random Sampling," in *CENIC*, Montery, 2010.

[15] B. L. Schatz, "Bradley L. Schatz. Wirespeed: Extending the AFF4 forensic container format for scalable acquisition and live analysis," in *DFRWS*, Philadelphia, PA, USA, 2015.

[16] B. Carrier, File System Forensic Analysis, Addison-Wesley, 2005.

[17] B. Carrier, "The Sleuth Kit (TSK) Library User's Guide and API Reference," [Online]. Available: http://www.sleuthkit.org/sleuthkit/docs/api-docs/4.3/. [Accessed 31 May 2016].

[18] "Binomial distribution," [Online]. Available: https://en.wikipedia.org/wiki/Binomial_distribution. [Accessed 31 May 2016].

[19] Christopher King, Timothy Vidas, "Empirical analysis of solid state disk data retention when used with contemporary operating systems," *digital investigation,* vol. VIII, pp. 111 - 117, 2011.

[20] "Snappy," Google, [Online]. Available: http://google.github.io/snappy/. [Accessed 31 May 2016].

[21] R. Botchek, "Benchmarking Hard Disk Duplication Performance in Forensic Applications," Tableau, 2008.

[22] S. Garfinkel, "Anti-Forensics: Techniques, Detection and Countermeasures," in *ICIW*, Mauritius, 2007.

[23] Roussev V, Quates C, Martell R, "Real-time digital forensics and triage," *Digital Investigation,* vol. 10, no. 2, pp. 158-176, 2013.

[24] Bogen PL, McKenzie A, Gillen R. Redeye, "Redeye: a digital library for forensic document triage," in *JCDL*, NDIANAPOLIS, INDIANA, USA, 2013.

[25] Garfinkel S, Nelson A, White D, Roussev V, "Using purpose-built functions and block hashes to enable small block and sub-file forensics," *Digital Investigation,* vol. VII, pp. 13-23, 2010.

[26] Shaw A, Browne A. A, "A practical and robust approach to coping with large volumes of data submitted for digital forensic examination," *Digital Investigation,* vol. 10, no. 2, pp. 116-128, 2013.

Vita

The author was born in the city of Penza, Russia. He obtained his bachelor's degree in computer science from Penza State University in 2011. He joined the University of New Orleans computer science graduate program to pursue a master's degree in computer science in 2014. The author become a graduate research assistant in 2015, where he worked in a research team with Professor Golden Richard III.