Advanced File Format Validation for File Carving

WITH APPLICATION TO THE PST FILE FORMAT

by

Mart Peters

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, Faculty of Science
Master Software Engineering
to be defended publicly on Tuesday July 13th, 2021, 14:00.

Student number:	852116739	
Course code:	IM9906	
Thesis committee:	Dr. Bastiaan Heeren (chairman),	Open University
	Dr. Ir. Hugo Jonker (supervisor),	Open University
	Ir. Vincent van de Meer (2nd supervisor),	Open University

Open Universiteit
de beste! www.ou.nl

Contents

1 Introduction **2**

1.1 Contributions 3

2 Background **4**

2.1 NTFS file system 4

2.1.1 Fragmention 5

2.1.2 File deletion 5

2.1.3 Sparse files 5

2.1.4 File recovery (undelete) 6

2.2 File carving 6

2.2.1 Current challenges for file carving 7

3 Related work **8**

4 Methodology **11**

5 Current state of file validation **13**

5.1 File validation techniques 13

5.2 Current challenges with file format validators 15

5.3 File format validation reference implementation 16

6 File format analysis **18**

6.1 File format specifications 18

6.2 Concepts used in file formats 22

6.3 File life cycle 27

7 File format validation feasibility **28**

7.1 File format validation requirements 28

7.2 File format validation principles 29

7.3 Mapping between validation principles and file format concepts 32

7.3.1 Relation between validation principles and validation techniques . . . 32

7.3.2 Relation between file format validation techniques and file format concepts 34

7.4 Necessary file validation principles 37

7.4.1 File signature 37

7.4.2 Recognizable data structures 37

7.4.3 Ability to match data with the same file 38

7.5 Additional file validation format principles 38

7.5.1 Ability to detect invalid/corrupt data 39

7.5.2 Ability to check consistency across the complete file 39

7.6 Challenging file format concepts for validators 39
7.7 Validation case study: WAD file format 40
7.7.1 WAD file format specification. 41
7.7.2 WAD file format validation feasibility 42
7.7.3 Usability of the feasibility method 44

8 PST file format validation **46**
8.1 PST file format specification. 46
8.1.1 Logical organization of the PST file format 46
8.1.2 Physical organization of the PST file format 48
8.2 PST file format validator feasibility. 50
8.2.1 Identified file format concepts 50
8.2.2 Minimum set of required file format concepts for validation 51
8.2.3 Additional file format concepts for validation 52
8.2.4 Challenging file format concepts for validators 52
8.2.5 PST file format validation feasibility. 53
8.3 File validator for the PST format 54
8.3.1 Design of a PST format file validator. 54
8.3.2 Functionality and limitations 61
8.4 Proof-of-concept implementation 63
8.4.1 Verification 64
8.4.2 Results 65

9 Conclusion, discussion and future work **67**
9.1 Answers to the research questions 67
9.1.1 RQ1: Which existing file carving techniques can be used in a file format validator? 67
9.1.2 RQ2: What kind of concepts are used in file formats?. 67
9.1.3 RQ3: What concepts are necessary for a file format validator? 68
9.1.4 RQ4: How can this be used to design a file format validator for a complex format? 68
9.2 Conclusions 69
9.3 Discussion 69
9.4 Limitations and future work. 70

Bibliography **i**

A File format investigation **iii**
A.1 Image iii
A.1.1 JPG. iii
A.1.2 PNG iv
A.2 Audio v
A.2.1 MP3 v
A.3 Video. vi
A.3.1 AVI vi
A.3.2 MKV vii

A.4 Documents ix
A.4.1 Office Open XML File format (docx, pptx, xlsx) ix
A.4.2 EPUB ix
A.5 Archive x
A.5.1 Tar x
A.6 Database xi
A.6.1 SQLite xi

B PST File format **xiii**
B.1 Grammar xiii
B.2 List of fields xiv

ABSTRACT

File validation is a technique to recognize and validate file formats in an arbitrary stream of data. File validation thus can be used to recover deleted files without relying on metadata of the file system, as file validation directly analyses an arbitrary stream of data. Furthermore, this technique can be used to recognize valid combinations of file fragments. Recovery and reconstruction of fragmented files is very challenging, however, file validation offers a potential path to success.

In this thesis we investigate how file format specifications can guide file format validation. We propose a method to determine whether file format validation is feasible and how this can be achieved using existing validation techniques.

To answer this question we approached this problem from a file format perspective, because file format validation relies on properties of a file format. We analyzed popular file formats of commonly used file types to identify and generalize commonly used file format concepts across the different file format specifications. The analysis resulted in the identification of commonly used file format concepts.

Existing file validation techniques rely on properties of a file format. Our findings were that these properties can be translated into the identified file format concepts of our research, this resulted in the identification of a relation between file format concepts and file validation techniques.

A file validator is required to recognize and validate files. We identified a list of necessary validation principles to support these requirements. A validation principle can be implemented by using specific validation techniques, this dependency provides the linking pin between the file validator requirements and file format concepts.

This resulted in a method to determine the feasibility of file format validation. The method consists out of identifying the used file format concepts by analyzing a file format specification. Based on the identified file format concepts, the corresponding file validation techniques are determined.

To verify the proposed method we apply the method on a complex file format. The PST file format is identified as a suitable candidate, because related work found out that PST files are frequently fragmented on a system. The PST file format is used for storing e-mails and calendar items of Outlook.

The conclusion of the method is that file format validation is feasible for the PST file format, because the file format contains sufficient file format concepts. We implemented a PST file validator using the suggested validation techniques provided by the method. The implemented PST validator was able to recognize file fragments and can be used to reconstruct file fragments into the original file.

Our proposed method can determine file format validation feasibility and identifies which validation techniques can be used in the implementation of a file validator. This means that a file format specification can provide guidance on the implementation of a file validator. We consider the proposed method a starting point, since it might not be complete. However, our method allows the addition of new validation techniques and file format concepts in case these are identified.

1

Introduction

File carving is a technique to recover deleted files without using file system metadata. However, file carving is limited in handling corruption in its input stream. In such cases, a file carver will determine the input is corrupt or, in the worst case, recover a corrupt file. In general, file carvers are not always able to locate the specific point of corruption, only to ascertain that the result is corrupt. Interestingly, there is a technique which is able to locate such corruption: *file format validation*. File format validation is to validate a file by checking whether it adheres to the specification of its format. The last point where it matches the specification is then the last point where the file was still considered valid.

File format validation relies on the specification of the file format being validated. While specifications of different files obviously differ, the set of 'ingredients' used in specifications overlaps – though specific concepts may go by different names. Moreover, similar concepts will likely support similar validation strategies, even for wildly different file formats. This raises the question:

To what extent can a file format specification guide file validation?

The following research questions are composed in order to answer this main question:

RQ1. Which existing file carving techniques can be used in a file format validator?
RQ2. What kind of concepts are used in file formats?
RQ3. What concepts are necessary for a file format validator?
RQ4. How can this be used to design a file format validator for a complex format?

RQ1: Which existing file carving techniques can be used in a file format validator? File carvers recover deleted files from a file system, in order to do this file carvers must be able to identify files. A file carver must determine the beginning and ending of a file and therefore should recognize and validate the contents of a file. This functionality is closely related to the behavior of a file format validator. Therefore, it is interesting to investigate which existing file carving techniques are used and how these can be applied in the context of file format validation. We performed a literature research to answer this question, because this question is about what existing file carving techniques are available, finding out what the status quo is.

RQ2: What kind of concepts are used in file formats? File format validation relies on the internal file structure that is dictated by a file format specification. A file format specification thus affects how a file is stored on a storage medium and therefore interesting to investigate the specifications of different kinds of file formats. The goal is to investigate whether file formats use similar concepts and to identify which concepts are used. Furthermore, these discovered concepts are analyzed with regard to what extent these concepts affect the added value of using file format specific knowledge. This provides insight on the feasibility of creating a file validator for a specific file format. In order to answer this question we performed an investigation on different file formats of common file types. The reason of using common and different file types is to get a complete and representative view of how file formats are organized.

RQ3: What concepts are necessary for a file format validator? Once the concepts used in file formats are identified, the next step is to investigate how the identified file format concepts affect file format validation, since file format validation might not always be possible for a given file format. The requirements and functionality of a file validator is studied to form a view on which validation principles are used to support the requirements. These necessary requirements form the baseline that is used to determine whether file validation is feasible. The next step is to investigate the relation between file format concepts and file validation principles, because if there is such a relation this could give an answer to the main question: to what extent a file format specification guide file validation. File validation techniques rely on specific properties of a file format. The file format concepts required to enable validation techniques are investigated. This approach forms the basis for a method to investigate the file format feasibility of a file format.

RQ4: How can this be used to design a file format validator for a complex format? The method to determine file validation feasibility is based on the findings of research questions RQ1, RQ2 and RQ3. The proposed method is applied on a complex format to determine the usability and possible limitations of the proposed method.

1.1. Contributions

In summary, this study has the following contributions:

- Assessment of known validation techniques
- Identification of commonly used file format concepts in file format specifications of popular file formats of common file types.
- Method to determine how a file format specification can guide file validation.
- Application of the method on a complex file format which uses a lot of commonly used file formats.

2

Background

File validation is a method that can be used to recover deleted files and file carving is the practice of recovering deleted files. Therefore, studies related to file carvers can be interesting from a file validation point of view. To understand how deleted files can be recovered, it has to be clear how files are stored on a storage medium (Hard disk, SSD, USB Flash drive, etc.). File systems are responsible for storing and deleting files on a storage medium.

Furthermore, certain file system features can affect the way a file is stored on a storage medium. To recover deleted files, certain techniques and principles are used. The techniques and principles that are currently used and known are introduced.

2.1. NTFS file system

File systems are responsible for storing files on a storage medium. NTFS is a file system and uses the following concepts: a hard disk is divided into sectors (sometimes referred to as blocks), the size of a sector is a property of the hard disk. Most of today's hard drives use a sector size of 512 bytes [Lin18]. Russon et al. [RF04] and Lin et al. [Lin18] documented how the NTFS file system operates. Similar to other file systems, NTFS uses clusters to allocate disk space for files. Clusters consists of a number of sectors, usually a number of the power of two. The number of sectors used for a cluster is fixed when the volume is formatted. NTFS has two types of cluster numbers:

- LCN: Logical Cluster Number. The first cluster number of the file system starts with 0. The LCN is the number of a cluster relative to the NTFS file system.
- VCN: Virtual Cluster Number. Clusters belonging to a file are referenced in the Master File Table (MFT) using VCNs. The MFT is used by the file system to store metadata of a file, metadata of a file stored in a MFT record. A VCN is the number of a cluster relative to a file. VCNs start from 0, which are sequentially increased by 1 until the last cluster allocated to the file.

Files of which the data is stored in the MFT record of the file are called resident files, these files do not use allocated blocks [vdMJvdB20]. Files of which the data is too large to fit in the MFT record of the file are called non-resident files. The data of non-resident files is stored in allocated blocks by the file system. References to these blocks of data are called data runs and are stored in the MFT record of the file. Data runs are a sequential list of cluster sequences. A data run consists of a starting cluster and a length of consecutive clusters.

The starting cluster of a run is the offset relative to the starting cluster of the previous run. Data runs are stored in the MFT entry of a file.

2.1.1. Fragmention

As explained in the previous section, the NTFS file system stores files on clusters. When a file is stored in a sequence of consecutive clusters, the file is not fragmented. If this property is not satisfied, than the file is fragmented. Fragmentation can occur in different manners and can lead to different degrees of fragmentation. There are two aspects to take into account [vdMJvdB20]:

- Amount of fragments
- Amount of fragments which are out of order

A sequence of consecutive clusters belonging to the same file are called fragments. These fragments can be stored in order on disk, with sections of empty data or data from other files in between. However, these fragments can also be present out of order on the disk. A fragmentation point is the last cluster belonging to a fragment.

Another aspect of fragmentation, is the amount of fragments of the file. The amount of fragments and the amount of fragments that are out of order, affects the complexity of recovering deleted files.

2.1.2. File deletion

The following happens on NTFS when a file is deleted: a flag in the corresponding MFT entry of the file is set to indicate the file has been deleted. Other data located in the MFT entry is not changed yet. MFT entries are reused, so until the MFT entry is reused, the original data of the deleted file remains in the MFT entry. In case of a resident file, the file data is completely stored within the MFT entry and is still present in case the MFT entry is not reused yet. In case of non-resident files the clusters in which the data is stored is administrated in the MFT entry using data runs. Clusters used by the non-resident files can be overwritten once non-resident files are deleted. Until the clusters are overwritten the data of deleted non-resident files is still present on the storage medium. Because of this property it is possible to recover deleted files.

2.1.3. Sparse files

A file can consist solely out of zeros or can contain a part that consists of a sequence of zeros. The areas containing zero data is called sparse data [Lin18]. NTFS can save disk space, by only writing nonzero data to disk and does not allocate disk space for storing sparse data [Lin18]. To make a file sparse on NTFS, the sparse attribute of the file needs to be set [Lin18]. A normal data run in a MFT record of NTFS contains a starting cluster address and the length of clusters of the data run. Sparse data is stored using a sparse data run. A sparse data run only specifies the length of the sparse data run, without a starting address of a cluster. A sparse data run does not occupy disk space, this means that a sparse data run can only be used if the sparse data is contiguous and at least the size of a cluster, since the length of a data run is specified in the number of clusters. This is because a cluster is the smallest segment of disk space NTFS can allocate. This property is also confirmed during testing with marking areas as sparse data using the fsutil tool (tool from Microsoft that allows file system operations [1]). If the specified range of sparse data is not big enough, the data on disk does not change. A file is either sparse or not sparse by changing the sparse attribute. The amount of sparseness can be expressed by indicating how much data of the complete file consists of sparse data. Please note that the sparse data can be scattered across the file and can exist out of multiple sections of sparse data. All the required administration of sparse sections is located in the MFT. This means that if a sparse file is deleted, the locations of sections which contain sparse is lost. Since the sparse data itself is not written on disk, it becomes a challenge to recover a file containing sparse data.

2.1.4. File recovery (undelete)

Data runs specify at which locations (clusters) the data of a file is stored. Therefore, the original file can be restored, by copying the data from the clusters mentioned in the data runs. However, this can only work in case the referenced clusters have not already been overwritten by another file. When a file is deleted the used clusters are made available in the $BITMAP attribute of NTFS. So after deletion the $BITMAP attribute of NTFS indicates that the used clusters of the deleted file are available. However, if the $BITMAP attribute indicates that one of the clusters of the deleted file is in use again, it is highly likely that the data of the deleted file is already overwritten by another file, because clusters of a deleted file should be free (not allocated). In this case that specific part of the data of the deleted file is lost, this makes it impossible to completely recover the deleted file. The procedure mentioned above describes how an undelete action can be implemented for NTFS. This only works if the MFT records of the deleted file and the corresponding clusters containing the data are still intact. Since the MFT entry also contains information regarding the sparse data (since this information is stored in the data runs of the MFT records), this method can also work for sparse files. In case the MFT record of the deleted file is no longer present, an undelete action cannot be performed. Since sparse data locations are stored in the MFT, this information is also lost. In this case we have to resort to a practice called file carving to restore the deleted file.

2.2. File carving

File carving is the process of reconstructing files without using information regarding the file structure of the files from a file system [Lin18]. File carving only uses the available block(s) of binary data. File carving globally consists of two steps:

1. **Acquisition** consists of finding the fragments that belong to a file that is been reconstructed.
2. **Validation/Verification** To rule out false positives, the reconstructed files have to be validated and verified whether the files have valid file structures. Fragments of files under recovery can be stored out of order on the disk. Thus in order to successfully reconstruct the file, the correct sequence of fragments has to be found. Validation can also be used to reorder recovered fragments of a file under reconstruction in case of a fragmented file.

[1] https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/fsutil

2.2.1. Current challenges for file carving

File carving restores the data of deleted files by going through the raw data of a storage medium, without using the metadata of the file system. During file carving the clusters of the storage medium are analyzed sequentially. In case a file is stored sequential in consecutive clusters, files can easily be restored. However, due to the use of file systems and optimizations of file systems, this is not always the case. If files are not stored sequentially in consecutive clusters, the complexity of restoring the file is increased. The following concepts affect the complexity of restoring files using file carving [vdB+14]:

- Fragmentation
- Fragment recognition
- Fragment reordering (out of order fragmentation)
- Performance (limiting the search space)

According Garfinkel [Gar07] conventional file carvers have two limitations:

- Only restore non fragmented files, these are files with data stored consecutively and in order.
- No extensive validation is performed on the recovered data. As a result the recovered data also contains invalid data that cannot be used (false positives).

Fragmentation Conventional file carvers use magic strings to identify the header and footer (or file size) of a file and restore the file starting from the header up to the data of the footer (or the file size) [PM09, Gar07]. All the data located between the header and footer is also included, even if this data contains data that belongs to another file. Thus this method can only restore data that is stored consecutively and in order. Fragmented files can be accessed by file systems, since they keep track of where the fragments are stored on the disk. However, in case of file carving the information from the file system is not available.

The degree of fragmentation affects the complexity of restoring and reconstructing files. Files can be fragmented in 2 or more fragments and the fragments can be stored out of order on the storage medium. Van der Meer [vdMJvdB20] investigated fragmentation on NTFS using a corpus of 220 laptops. Findings were that the degree of fragmentation and out of order fragmentation where non-negligible.

Fragment reconstruction Fragment recognition is the ability to recognize data fragments. When data is fragmented, the fragments of a file need to be recognized in order to identify to which fragment the data belongs. Recognized fragments can be used to reconstruct the files under recovery. During the reconstruction of a file the fragments of the same file need to be recognized and possibly also need to be reordered in case the file was fragmented and out of order.

Performance Another aspect that forms a challenge is performance. Performance is related to how much effort, which can be computational power or time, it takes to perform an analysis. For instance, if the file carver returns a lot of false positives, it will potentially cost additional time to filter out the false positive results. However, if the file carving is very thorough and returns a lot of valid results, this can take a lot of time to perform. This long duration can be a problem. Trying out all the possible combinations of two fragments until the file is valid, can potentially cost a lot of time. Limiting the amount of possible combinations (search space) would improve the performance.

3

Related work

This thesis investigates file format validation using file format structures. Knowledge of file format structures could be used during file carving to recover deleted files. Golden and Roussev introduced the use of file format structures in a file carver [RIR05]. Their file carver, which is called Scalpel, used a database of header and footer data structures of specific file formats. Scalpel analyses the raw data of a storage medium under recovery, without using information from the file system. When a known file header is recognized and the file size is determined, which can be based on file size metadata or the presence of a file footer, the data can be recovered. The data is recovered by copying the header structure and the data directly after the file header until the size of the file is reached. One limitation of this approach is that this method only works for files that are contiguously stored on the storage medium. Fragmented files cannot be recovered with this method [PM09, RIR05, Gar07].

As mentioned above, the first generation of file carvers can only recover deleted files that are stored contiguously. A limitation of the first generation of file carvers, like Scalpel, is that it is not able to recover fragmented files. To address this limitation, Garfinkel et al. [Gar07] introduced the concept of object validation. Object validation is the process of determining which byte sequences form a valid file that adheres to a specific file format specification. Object validation uses the internal file structure to determine whether files are valid. Object validation is a superset of file validation, since valid byte sequences of files embedded in other files can also be recognized. For instance, a JPEG picture that is stored in a Word document. The approach to recover fragmented files is to provide byte sequences to the validator. The validator provides feedback whether the provided combination of data sequences is valid. Garfinkel describes a method, bifragment gap carving, for carving files that are fragmented into two fragments. Bifragment gap carving assumes that in case a file is fragmented into two fragments, that data not belonging to the file under recovery is between the two fragments of the file under recovery. This data in between the two file fragments is called the gap. By shifting the dimensions of the gap until the validator succesfully validates the two fragments combined, the gap can be identified. Using this method a fragmented files with two fragments can be recovered. Garfinkel introduced the concept of combining a file carver with an object validator to recover and reassemble fragmented files.

A continuation on the research of Garfinkel et al. [Gar07] for carving fragmented files using validators is provided by Cohen [Coh07]. Cohen considers the carving problem as a mathematical problem by creating a mapping function between the file bytes (file under recovery) and the image bytes (storage medium). The mapping function is determined by using a discriminator. The discriminator uses the internal file structure to determine whether a proposed mapping function is valid or has to be modified. The discriminator is also known as validator, as introduced by Garfinkel [Gar07]. In the approach of Cohen, the discriminator is used to efficiently determine the correct mapping function. In the end the mapping function is used to recover the file.

Pal et al. [PM09] identified the following limitations using the bifragment gap carving method as introduced by Garfinkel [Gar07]: the method does not scale with fragmented files that have large gaps in between and the method is not designed for fragmented files with more than two fragments. To address these limitations, Pal et al. introduced the smartcarver architecture. The smartcarver architecture supports file carving of both fragmented and unfragmented files. The smartcarver architecture divides a file carver into 3 components: preprocessing, collation and reassembly. File format validation can be used in the collation component in which raw data is identified as a specific file type. File format validation can also be used during the reassembly component in which the discovered fragments of a file are reassembled, by merging data together based on file structure of a file format. Poisel et al. [PT13] performed a literature study on current file carving techniques. Poisel et al. state that the main challenges for existing file carving applications are accuracy and performance. Poisel et al. state that approaches that use the structure of file formats, also known as file signature approaches, are highly efficient for recovering fragmented files. File signature approaches that are based on discriminators and mapping functions are particularly useful for file formats with a fixed structure and recognizable features. This thesis investigates which concepts a file format should have to apply file format validation with file signature based approaches.

Related work provides several proof of concepts for using information of file format structures for file carving. Garfinkel et al. successfully applied the approach of file format validation in combination with file carving for the following formats: ZIP, JPEG and Microsoft OLE [Gar07]. Cohen also successfully applied his mathematical approach with using the mapping function for the ZIP and PDF file format [Coh07]. Wei et al. [WZX10] created a file carver to recover fragmented RAR files. The file carver uses a validator that uses the internal file structure of the RAR file format. The validator is used as a discriminator for determining the mapping function as introduced by Cohen [Coh07]. Chen et al. [CZX+08] created a file carver to recover fragmented PDF files. This file carver uses file validation techniques that also use file format specific structures, similar as Garfinkel [Gar07] and Cohen [Coh07]. Yang et al. [YXLS17] created a file carver that uses file format information for the AVI file format. The AVI file carver can identify and reorder frames (fragments) from an AVI file to carve fragmented AVI files. The identification of frames is based on file format information and the reordering of the frames is based on data structures that contain properties (metadata) of frames sizes and frame locations.

Furthermore, related work shows that file format specific approaches provide an added value when compared with generic file type approaches. Veenman et al. [Vee07] introduced two models for recognizing file types: Type-All and type-X recognition. The type-All model tries to recognize multiple file types, the type-X model only tries to recognize one specific file type. The conclusion of Veenman was that the type-X model had better results, since the type-X model was better able to distinguish the features of similar file types. A similar conclusion was drawn by Roussev and Garfinkel [RG09], who performed a study with a bottom-up approach, by starting with examining specifications of popular file formats. Based on this approach, Roussev et al. concluded that specialized methods are required for each specific file type to correctly distinguish different file formats. Distinguishing different file formats is a requirement for reassembling fragmented files, because if file formats of fragments are incorrectly recognized, this can lead to difficulties when the fragments are reassembled into the original file.

Related work identified that restoring fragmented files using file carving is a challenge, since this requires recognition of fragments and reassembly of these fragments [PM09, Coh07, Gar07]. To what extent fragmentation actually forms a problem in the real world, depends on the degree of file fragmentation. Van der Meer et al. [vdMJvdB20] investigated file fragmentation on the NTFS file system by analyzing file fragmentation on a corpus of 220 Windows laptops. Van der meer et al. investigated the degree and presence of out of order fragmentation for file formats. The majority of the files are stored contiguously. When files are fragmented, the majority of the files are fragmented in two fragments. Files that are fragmented in two fragments can be restored by using the bifragment gap carving method as introduced by Garfinkel [Gar07]. However, Van der Meer et al. found out that a specific group of file formats (jpeg, tif, flv, pdf, ppt, pptx, pst, accdb, db, SQLite , 7z, rar and zip) are fragmented more often and out of order when compared with other file formats. According to Van der Meer et al. the reconstruction of this specific group of file formats is not sufficiently supported by current tooling. One of these identified file formats is the PST file format. Our work investigates how to create a file format validator using data structures for the PST file format.

4
Methodology

In order to answer the main research question, we have chosen to approach this from a file format specification perspective. Because file validation relies on properties of a file format and these properties are defined in file format specifications. The expected outcome of this approach is to identify general concepts that are commonly used across different file format specifications. If we are able to identify general concepts, we can investigate whether a relationship exists between the general concepts and the currently known validation techniques. We expect that there is such a relation, since validation techniques also rely on specific properties of a file format. If these specific properties happen to be part of the identified general concepts, we can answer our main research question. Because a file format specification can be studied for the presence of commonly used file format concepts. Once these file format concepts are identified, the concepts can be used to determine which validation techniques are suitable. As a result the file format specification thus can potentially guide file format validation.

An alternative approach would be studying existing file validator implementations. We did not choose for this approach, because we expect that this does not give a complete picture of the situation. If file validators of only one specific file format were analyzed, we expect to recognize commonly used practices for validating the same file format. However, if the chosen file format does not cover a lot of potential file format concepts, we potentially miss concepts that do exist in other file formats.

Another approach would be to analyze file validator implementations of different file formats. In this case we potentially miss common practices among file validators, since the chosen file formats might be completely different from each, which potentially results in the use of very different and specific file validator implementations. Another aspect might be that a file specific property is used by a validator, which in general cannot be applied in file validators of another format.

The target is to identify common practices and commonly used file format concepts to identify which approaches can be reused in the implementation of a new file format validator. In order to overcome the limitations of the two mentioned alternative approaches and accomplish the target, a third option was chosen: identify general concepts that are commonly used across different file formats by analyzing file format specifications of different common file types of popular file formats. As mentioned before, this method allows the identification of general concepts that are commonly used in different file format specifications. This method also has limitations, since the list of identified concepts can be incomplete. The completeness of the list of identified concepts depends on the file format specifications that are selected for analysis. During the selection of the file formats this aspect was taken into account, by using a diverse selection of file formats of different file types and multiple file formats for each file type. This reduces the risk, however it still does not guarantee a complete list. We consider this risk acceptable, due to the used selection of file formats. Furthermore, the list of identified concepts can always be extended once more file formats are investigated. The chosen approach allows this flexibility towards additions in the future.

Once the commonly used file format concepts are identified, the relation between the existing validation techniques and the file format concepts is identified. We choose for this approach because we expect it is very likely this relation is present, because the validation techniques rely on properties of file formats and we expect these properties can be identified as general file format concepts. If new file format concepts are discovered during the analysis of new file format concepts in further investigations, these new file format concepts can also be mapped to validation techniques. Therefore we expect our proposed method allows flexibility with regard to newly discovered file format concepts. This flexibility is another reason why we have chosen for a method that uses the approach from a file format specification perspective.

Once we identified a method to determine how a file format specification can guide file format validation, this method still needs to be validated. This is achieved by applying the proposed method on a complex file format. We expect the proposed method provides insight on which validation techniques can be used in the design of a new file format validator. Once this design is created based on the validation techniques that are identified by the proposed method, we actually implement the designed file format validator. This implementation is validated by testing the behavior of the implemented file validator. If this implemented file validator behaves correctly we consider this a proof of concept for our proposed method.

Since the implemented validator is only a proof-of-concept, we are not going to extensively test the implementation. Each functionality that can detect a different type of corruption is tested to verify whether the validation works. For this we make use of a relative simple and small valid file and under analysis and the validator must detect this is valid. In further testing, the valid file is manually altered to introduce corruption at specific and known locations. It is verified whether the validator is able to detect the introduced corruption at the expected location of several cases. This functionality is also tested with a larger PST file on which several file operations are executed, such as deletion of items, in order to test the resilience of the validator against the dynamic properties of the PST file format.

5
CURRENT STATE OF FILE VALIDATION

We perform a literature research in order to understand what the current state of the art with respect to file validation is and answer RQ1:

> **RQ1. Which existing file carving techniques can be used in a file format validator?**

The literature research focused on two areas:

- File validation techniques and strategies which are applied during file format validation
- File format specific implementations of file format validators

Studies of file validation techniques and strategies provide insight and an overview on what is currently known and used in the field. This current state of the art provides insight on which concepts of file formats are used by the file validation techniques and strategies. This knowledge is useful when analyzing the different concepts that are present in file formats. In this thesis an analysis of used concepts in different file formats is executed. The currently known file validation techniques and strategies provide context and insight on what to look out for during the analysis of the different file formats.

Studies of file format specific implementations of file format validators are applying the currently known techniques and strategies for file validation. This literature provides a reference on how the techniques and strategies can be applied on a specific file format. Additionally, these studies give insight on which other file format specific optimizations are possible. Therefore, studies of file format specific implementations of file format validators provide additional context on which file format properties to look out for when investigating the file format specifications.

5.1. FILE VALIDATION TECHNIQUES

Garfinkel [Gar07] uses the term object validation for the process of validating sequences of carved bytes of a specific file format. Object validation is considered a super set of file validation, because a file can be embedded in another file. For instance, a JPEG file that is embedded in a PST file. Object validation tries to check whether a sequence of bytes is still a valid file that can still be opened by, in the case of PST files, Outlook. Outlook should be able to open the file without any error messages and should be able to display uncorrupted information.

In file carvers the steps to perform acquisition and validation are not always clearly separated. Sometimes these steps are embedded within the file carver itself. In case of the term object validation, the separation between acquisition and validation becomes more apparent. In this concept a file carver can consist out of an acquisition part combined with several object validators for each supported file format.

The following summary of the current state of the art validation techniques is compiled based on the work from Lin et al. [Lin18], Van den Bos et al. [vdB+14] and Garfinkel [Gar07]:

Header and footer validation if a file format contains static sections that can be recognized as header and footer, a file validator can use these sections to validate a file. A shortcoming of this method is that the data between the header and footer is considered as part of the file, without actually validating this data. So this method only works if the file is not fragmented and stored contiguously.

Magic Number Matching A technique used for recognizing the data structure of binary file formats, magic numbers are used. A specific file format, for instance GIF files, always start with the ASCII string "GIF" and end with byte 0x3B.

Container structure validation uses the internal structure that the file format dictates, of container files to validate files. A file format can use a specific layout of the file and have specific sections at specific locations. This layout can be validated against the file format definition to validate the file.

Data Dependency Resolving The file structure of file formats can contain fields that provide information regarding the contents of the file. For instance, a length field that specifies the size of the entire file. A file format can also contain references to other parts in the file, for instance using an absolute file offset within the file. This information can be used by validators to make sure that the file does not contain inconsistencies.

Validating with decompression Once the structure is validated, the actual contents within these structures still have to be validated. It could be that the file format encodes or compresses data in specific sections. These compressed/encoded data sections can be validated by checking whether they can be successfully decompressed/decoded. This allows the validator to verify whether the contents of the sections are corrupt. For instance, when the decoding of a MPEG file is successful it is likely that the movie is (partly) view-able. If decoding of the file fails, the location of the error can also provide information regarding the location of corrupted or missing data. Theoretically it is still possible that data is successfully decoded or decompressed but does not contain any information that makes sense. It depends on the type of information whether automatic detection of sensible information is possible. For instance, if it is known that the data should contain human readable text, this property can be verified by checking the presence of invalid characters (non human readable text).

Algorithm Output Analysis File formats can use encoding or compression on the data stored in the file. When analyzing a block of data it is possible to determine whether it is likely that the data was compressed or encoded in a specific way by using bit sequence matching.

Internal verification checking Internal verification uses the actual contents of the file to verify parts of the file under recovery. For example, files can contain information about length and a cyclic redundancy code (CRC) of the contents. This CRC field from the raw binary data, can be used during the verification of the file that is recovered from the file carving process. The validator can compute the CRC value of a specific section of data and compare this value with the stored CRC value of the file.

Semantic validation Uses the semantic information that is stored in the file. For instance, if it is known the file contains text about a certain topic in a specific language, this information can be used to determine whether it is likely that other recovered text sections also belong to the same file. Garfinkel also indicates this approach is currently hard to automate [Gar07].

Manual validation It is possible that a recovered file passes all the validation checks and has a valid file structure and can successfully be opened, but still does not contain human readable text. The process of inspecting the recovered data with human eyes is called manual validation. This is required to detect false positives that slip through automatic validation.

Bifragment Gap Carving (BGC) [Gar07] tries to identify the header and footer of the file under recovery. If the header and footer including the data in between is not valid, it is assumed a gap of data is in between the two fragments of the header and footer. The method each time tries a different gap size between the two fragments in order to find the gap size. Once the gap size is found, the file can be recovered by leaving out the gap between the two fragments. This technique is used at file carver level and not directly by a file validator. However a file validator can assist the bifragment gap carving technique in identifying when the file is valid.

Metadata Based Data Recovery [PT13] uses information from the file system about the file under reconstruction that might be left after the file was deleted. In case a file is deleted this information is not always available, in this case we have to resort to the techniques mentioned above.

5.2. Current challenges with file format validators

Cohen states that general purpose programs (such as Outlook for PST files) may be used as validators, however the usability of the results of this approach is limited [Coh07]. This is due to the following factors:

- General purpose programs might not expect that provided files are corrupt. It could be that general purpose programs do not perform a complete integrity check of the file, as a result corrupt files might not be detected.
- Another approach of a general purpose program when handling a corrupt file, is to attempt to recover the provided file. Detected errors in the file might not be reported.
- Additionally most general purpose programs fail to pinpoint the location of the errors that caused the corruption in a file.

This indicates that there is room for improvement with regard to the issues mentioned above, for validators that provide more detailed feedback regarding the state of a file. A file under reconstruction, ideally is checked sector by sector until the point of fragmentation is detected, since from that point the file structure is no longer valid. The reconstruction process can try appending different blocks of data until the file structure is valid again, the number of combinations can be very big which make it impractical to recover the file. Additional information and assumptions from the environment can be used to reduce the number of possible combinations, this improves the performance of reconstructing the file. Cohen mentions that these assumptions can be derived from the file structure of the file format and lead to two types of constraints:

Positive constraint A positive constraint limits the amount of possibilities by identifying a sector of data that belongs to the current sequence of data in the file and can be appended to the file is reconstructed (for instance an identified file header).

Negative constraint A negative constraint limits the amount of possibilities by ruling out candidates that belong to the current sequence of data in the file. For instance when its likely that the sector belongs to another type of file.

Additionally, information regarding the used file system can provide information about sector sizes, which may lead to other assumptions that files only begin at sector boundaries and that fragmentation only can occurs on sector boundaries.

5.3. File format validation reference implementation

Chen et al. [CZX+08] build a PDF file carver, by applying the findings of Garfinkel [Gar07] and Cohen [Coh07] in practice. The lessons learned and techniques used in the research of Chen et al. are interesting for creating a new file format validator, because this study is focused on creating a file carver except targeted at another file type. The PDF file carving method of Chen et al. consists of five validation methods:

1. **Header/file length/maximal offset of objects/footer validation** The validator searches for the header of the PDF file, by searching for a magic string. Also the footer is searched by using a magic string. These magic strings are properties of the PDF file format. PDF files can be organized using two different methods: linearized and non-linearized. The following approach is applicable for linearized PDF files: The length of the PDF file is stored in the data. The length value is used to check whether at the end of the file the magic string indicating the end of the file is present. If this is the case it is validated that the PDF file is intact. For non-linearized PDF files another approach is applicable: the internal cross-reference table of the PDF file is used to determine the maximal offset of the objects in the table. If an object is found at the location of the maximal offset, the PDF file is considered complete. If the the validations mentioned above fail, the PDF file is marked as fragmented.
2. **Internal structure validation** The internal structure of the PDF file format contains a cross-reference table referring to objects that are present at certain locations in the PDF file. The objects in the cross-reference tables are tested for completeness and availability. Another step is seeking for fragments of a PDF file. To successfully find a fragment of the PDF file a fragment must be large enough to contain at least one complete object of the cross-reference table and one or two fragmented objects. Incomplete objects are put together by appending fragments, if the validator now detects a complete object the two fragments are placed in the correct sequence. This process is repeated until the PDF file is complete or is marked broken, in case the next fragment of the sequence cannot be found.
3. **Entropy difference validation** Entropy is the amount of randomness or uncertainty of a piece of information. This entropy value can be used as a signature to identify different file types. The entropy value can be used to indicate which data belongs to each other and identify boundaries of data. For instance, compressed data has a specific entropy value, when some non-compressed data occurs after the sequence of compressed data this can be detected, because non-compressed data has another entropy value. The difference of the entropy value between two segments of data determines whether it should be considered as data that belongs together. However

entropy cannot identify the exact fragments of the PDF file.

4. **Validating with zlib/deflate decompression** Data in PDF files is most of the time compressed using the Flate data compression method. Two fragments of data segments are put together and decoded using the decompression method, if this succeeds the two segments belong together. This process can be repeated to find other fragments that belong to the same data sequence.
5. **Character table validation** Boundaries of fragments of a fragmented PDF file contain characters. Based on the knowledge of the file format it can be determined whether the last couple of characters of a fragment and the first characters of a fragment belong to each other.

6 File format analysis

In this chapter we investigate RQ2:

RQ2. What kind of concepts are used in file formats?

In order to answer this research question we perform an investigation on a selection of file formats, to find out which concepts a file format typically contains and how these concepts can affect the accuracy of file format validation. The file formats selected for this analysis consist of popular and well documented file formats from the following categories defined by Van der Meer [vdMJvdB20]:

- **Image:** JPG, TIFF, PNG
- **Audio:** MP3, OGG
- **Video:** AVI, MKV
- **Documents:** Office Open XML, EPUB
- **Archive:** ZIP, TAR
- **Database:** SQLite

The categories are the same categories as introduced by Van der Meer. Requirements for the investigated file formats are that each category had to be represented by one or more file formats and the availability of documentation for each file format. Furthermore, the file formats must be commonly used in order to acquire a representative view of which file formats are typically present on a system.

The goal of the file format investigation is to map the identified concepts in the physical layout of a file format to validation techniques in this chapter.

Please note that this chapter only contains a selection of the investigated file formats in order to understand which concepts are present in a file format. However, more file formats were investigated in order to find file format concepts and the other investigated file formats can be found in Appendix A. More file formats were investigated to come to a more substantiated conclusion with regard to the identified file format concepts and to prevent that we miss certain file format concepts that were not present in selection of file formats.

6.1. File format specifications

To identify which concepts are used in each different file format, not every detail of each file format has to be understood. Therefore, only the aspects of a file format that are relevant in the context of file format validation are described.

Please note that the terminology of the file format descriptions follows the file format specification of each file format. As a result similar concepts in different file formats can use different names or naming conventions in this chapter. This is intentional, because this allows the reader to easily locate the original terminology in the corresponding file format specification. Another reason is to identify which file formats use similar concepts and mapping this to the original terminology of each different file format.

Each file format is described by a schematic diagram that represents the physical layout of a file format. Figure 6.1 contains all the used concepts that are used to describe the schematic diagram of a file format. The following notation is used in the schematic diagrams:

Dark grey area Sections of data that contain recognizable byte sequences, such as magic strings.

White area Sections of data that do not contain recognizable byte sequences.

Black arrows Indicate the presence of a reference in a section to a specific other location in the file.

Boxed white sections Indicate the name of a section, which is a specific area in a file. Sometimes a section consists of multiple other sections, for instance in case of a list or a parent-child relation within a file.

Green arrows Indicate the data covered by a corresponding CRC value, the black line indicates the location of the CRC value.

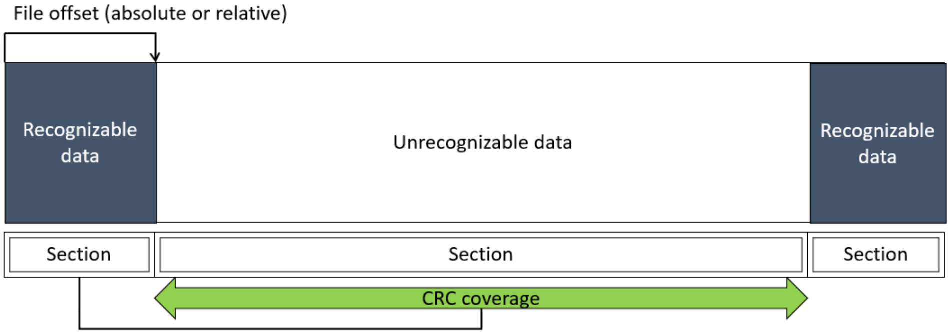

Figure 6.1: Overview of file format diagram concepts

Please note that not all details of each file format are discussed. For instance, not every field of a header or section is explained. Only information relevant in the context of file validation or required to understand how the file format works is mentioned. This in general also applies to omitting the details of how information is encoded, since this does not provide an added value for understanding the organization of a file format.

TIFF The TIFF file format[1] consists of different sections. These sections are not identifiable by recognizable byte sequences. Instead, the TIFF file format uses absolute file offsets to refer to each different section within the file.

Figure 6.2 provides a schematic overview of the file format. A TIFF file starts with an image file header at the start of the file. Only the image file header is recognizable due to a specific byte sequence at the start of the image file header. The image file header contains

[1] https://www.itu.int/itudoc/itu-t/com16/tiff-fx/docs/tiff6.pdf

an absolute file offset to the first Image File Directory (IFD). The location of the image file directory can be anywhere in the file, as long as it is after the image file header. There can be multiple IFD's, the file format dictates that at least one IFD has to be present. The IFD structure is not recognizable based on the byte sequences. Each IFD contains a reference to the next IFD (if another IFD is present), by using an absolute file offset.

Each IFD contains at least one directory entry. A directory entry, also known as TIFF field, has a specific tag and datatype and a reference to the value data. The value section contains the corresponding data of a TIFF field. There is a vast number of different tags available, the order of the tags must be ascending in the IFD entries. The TIFF fields required in the IFD depend on the image type stored in the TIFF file. Certain data values of the file can be compressed or encoded, for instance values containing the image data. TIFF supports multiple methods of compression: LZW, JPEG, Huffman, PackBits or no compression.

The TIFF file format does not make use of recognizable byte sequences to identify sections within the file format, only the image file header at the beginning of the file is recognizable. Instead of recognizable byte sequences, the sections can be found based on the use of absolute file offsets. From a file validation point of view these offsets cannot be used as identifiers for the different sections, since an absolute file offset is an arbitrary bit string. Furthermore, it is difficult to verify an absolute file offset to another field, because it is difficult to verify whether the referred field is correct, since the field itself is also an arbitrary bit string.

The TIFF file format contains a large number of different field types (Directory entry and value). Not each different field type of the file format is explained, because fields are structured and used the same for each type of field. Only the type of information stored in each type of field is different. This can also be seen in Figure 6.2, there are no differences between the different directory entries, which are also known as TIFF fields. Therefore, the knowledge of all the different field types is not required to understand how the file format is organized.

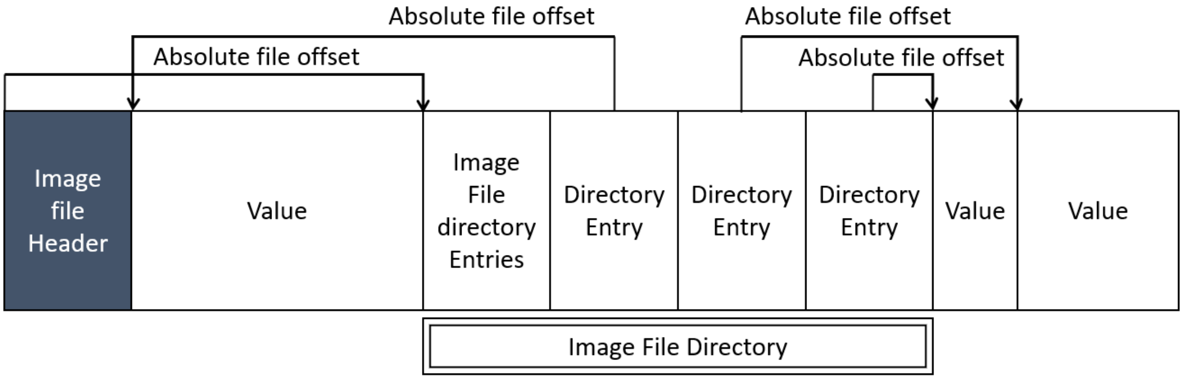

Figure 6.2: TIFF file format diagram

OGG The OGG file format[2] is a container format that encapsulates data, for instance audio data (using the Vorbis codec). The OGG file format consists of pages that contain data, these pages are called ogg pages. Pages can be used stand-alone and contain all the infor-

[2] https://tools.ietf.org/html/rfc3533

mation required for a decoder to recognize, verify and handle the page, without requiring the other pages of the bitstream.

Figure 6.3 provides a schematic overview of the ogg file format. As mentioned before, an ogg file consists of a sequence of ogg pages. Each logical stream contained in an ogg file starts with a beginning of stream (BOS) page and ends with an end of stream (EOS) page.

Each page starts with a page header followed by one or more ogg segments. Page structures are recognizable and contain the following information: the current position of the data in the stream, the meaning of this position depends on the used codec of the encapsulated data. A bitstream serial number is a unique serial number that identifies the logical bitstream, since an ogg file can contain multiple tracks in case of an audio file. A page sequence number, which is the sequence number of a logical bitstream, in case of audio, an audio track. Also a CRC checksum of the page is present in the page header, this checksum can be used to verify the contents of the page including the page header (excluding the value of the CRC field itself). An encoder puts the audio data in the page data structures, in case of the vorbis encoder the data is Huffman encoded. The ogg encapsulation process splits the audio data into different segments, the maximum size of a segment is 255 bytes. An OGG page contains a sequence of segments, segments do not have a header. However, the number of segments and length of each segment contained in the page, is stored in the page header of. The data of the segments is accessible by using the lengths of the segments that are stored in the page header.

The ogg segments stored within the pages are stored directly after each segment and do not contain any specific recognizable byte sequence between the segments. As a result the ogg segments itself are not recognizable. The data stored within the ogg segments depend on the used codec when creating the ogg file. Pages itself are recognizable and verifiable using the CRC value, the maximum size of a page is 64 KB and are usually between 4-8 KB. This could be a challenge in case the file system uses smaller clusters than the maximum size of a page. Because, it is not possible within pages to detect from which exact part the data is no longer valid, it is only possible to verify the complete page.

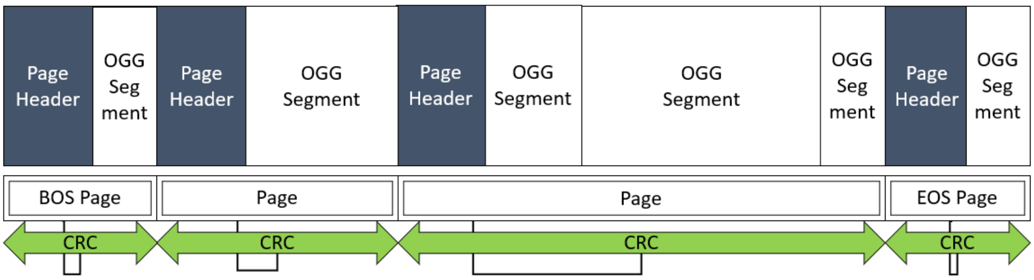

Figure 6.3: OGG file format diagram

Zip Zip files are used to contain (compressed) files.

Figure 6.4 contains a schematic overview of the zip file format. The zip file format[3] mandates a local file header for each file contained in a zip file. Each file stored in a zip file has a preceding and recognizable local file header. The local file header contains the following metadata: the CRC value of the uncompressed data of the file stored in the zip file. Also the files sizes of both the compressed data and compressed data of the file and the file name are stored in the local file header.

[3] https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

After each local file header, the compressed or non compressed data of the file entry is located. Each file entry in a zip file has a corresponding central directory header record entry in the central directory section. The central directory structure is located at the end of a ZIP file. The central directory consists of file header entries. A file header entry is recognizable and contains the following metadata: the compression method of the entry and a CRC checksum of the data. Also the compressed size and uncompressed size and file name of the file is contained in the file header entry. The relative offset of the local header with respect to the start of the zip file is also stored in the file header entry.

At the end of the central directory structure, a Zip64 end of central directory record is located. The structure is recognizable. This record contains the following metadata: the total number of entries in the central directory. The size in bytes of the central directory. Furthermore, the relative offset of start of central directory with respect to the start of the zip file is stored in the Zip64 end of central directory record.

Data compression may be used for the data segment of each file entry. The method that is typically used for compression is deflate.

The data sections in Figure 6.4 are marked as unrecognizable, because of the following reasons: the data can be stored uncompressed and the size of the data section is arbitrary. The arbitrary file size can be larger than the cluster size of a file system, this can lead to challenges during validation. If the data is uncompressed the validator cannot always detect which data belongs together and from which point the data is no longer valid. Invalid data can be detected due to the presence of a CRC value of the uncompressed data. However, the validator can still not always pinpoint the exact location from which the data is no longer valid, because it is not always possible to detect from which point encoded/compressed data no longer is valid. Therefore, the data section is marked as unrecognizable.

Although almost every field present in the zip file format specification is described in the preceding text. Only the fields that are relevant with respect to file format validation or fields that are required to explain how the zip file format works are mentioned in the text above and Figure 6.4.

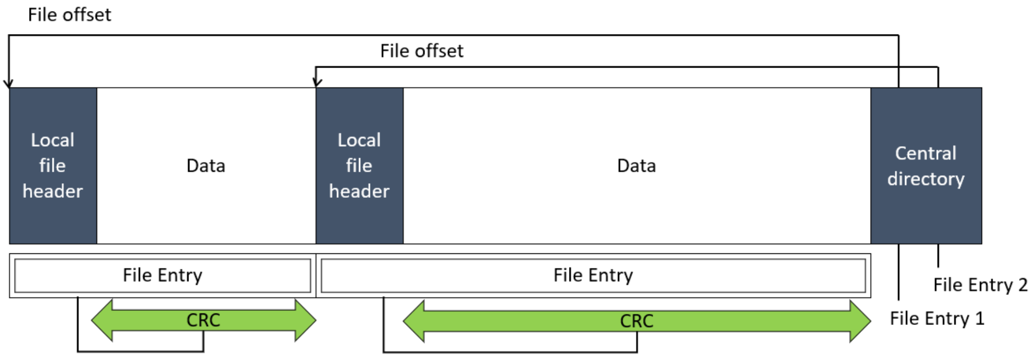

Figure 6.4: ZIP file format diagram

6.2. Concepts used in file formats

Analysis of the different file formats indicates that file formats use similar concepts. It could be that not all concepts are present in each file format. However, in general similar concepts and structures can be found in different file formats. These concepts typically lead to the use of certain structures of the data in the file format. These structures can be recognized and verified by a validator. These structures typically contain metadata of the file, this metadata can be used by a validator to validate the file under analysis. The concepts shown in Table 6.1 can be found in file formats.

Table 6.1: Concepts used in file formats

Sections	Hierarchy/Structure
References	Error detection
Encoding	Metadata

The concepts mentioned in Table 6.1 are typically implemented using magic strings or specific byte sequences. This can range from fixed hard-coded values that are always present in a specific area in the file format, to a limited set of valid values at specific locations. Checking all the possibilities of the limited set of valid values for the fields of the data structure can be used as a byte sequence signature to detect the presence of the specific data structure. These specific byte sequences can typically can be found in the header or other sections of a file format. The file format itself can also contain information from which other references or identifiers of the file can be retrieved or extracted. This information can be used to find other parts of the file under recovery. The following list summarizes the applications each identified concept in the analyzed file formats:

Sections clusters (mkv), segments (jpg), pages (ogg, SQLite), frames (mp3), blocks (pst) or chunks (png, avi), file entries (zip, tar, epub, ooxml)

References Absolute file offsets (tiff, zip, avi, mkv, ooxml, epub), page numbers (SQLite)

Encoding image data compression (jpg, tiff), Deflate (png, ooxml, epub, zip), Huffman encoding (mp3, ogg/vorbis), codec specific compression in case of containers (avi, mkv, ogg).

Hierarchy/Structure Specific ordering (jpg, tiff, png, mp3, ogg, avi, mkv, ooxml, epub, zip, tar), hierarchy (avi, mkv, ooxml, epub), recurring structures (mp3, ogg, avi, mkv, ooxml, epub, zip, tar)

Error detection CRC validation (png, mp3, ogg, ooxml, epub, zip), other type of checksum (tar)

Metadata file size (avi, SQLite), section size (png, avi, mkv, ooxml, epub, zip, tar, SQLite), information regarding the structure of the contents (ogg, avi, mkv, ooxml, epub, zip, SQLite)

Sections. Sections are specific areas in a file format. Typically, sections have a specific and sometimes recognizable data structure that forms the section. Examples of sections are file headers and file footers (trailers). A File header is a section specified by the file format that is located at the beginning of the file. A file footer is a section specified by the file format that is located at the end of the file. File headers and footers are very commonly used within file formats, even if the file contains almost no structure. File headers typically also contain a file signature, this can be used to recognize a specific file format. Therefore, the file header and footer are identified as a separate concept.

Typically, a file format also has sections to store data that is contained in the file, such as audio/video data. These sections are sometimes called clusters (mkv), segments (jpg), pages (ogg, SQLite), frames (mp3), blocks (pst) or chunks (png, avi). These sections can have a specific signature, dimensions or location, this depends on what is specified by the file format.

Sometimes these sections contain a CRC value. The CRC value can be used to validate the content of the section, in this case the validator can validate the contents of the section. In case there is no CRC value available, the validator can only check whether the section adheres to the specification of the file format.

The size of a section can vary between file formats. It is possible that the file format allows long sections, this could affect the accuracy of a validator. The validator can detect when a section is not valid, but to indicate from which point the section is no longer valid could be a challenge. For instance, the length of a field containing arbitrary data is known and the file format dictates that at the end of the data section a specific data structure has to be present. In case the structure at the end of the section is not found, this could mean that somewhere between the expected data structure and the beginning of the data section, the data is no longer valid (possibly due to fragmentation). Since the data is arbitrary, it is difficult to pinpoint from which point the data is no longer valid. In case the sections are small, this uncertainty can be acceptable. However, when big data sections are allowed this also implies a higher uncertainty, thus less accuracy to indicate from which point the file is no longer valid.

Sections could contain information regarding the length of a section. This can be in the form of a specific field that contains the length of a section, however the file format specification can also specify that the length of a section has a specific length. In this case the length of a section is a property of the section. The length property is important to identify sections, because if the length of a section is unknown it is difficult to identify the start and end of a section. In case the start and ending of a section are recognizable, the length property can also serve as a double check to verify that the data between the start and ending of a section is consistent with the specified length.

A file format can also contain unrecognizable sections. Unrecognizable sections are sections that do not have recognizable byte sequences to recognize the location of a section, when looking at the raw data. SQLite for instance uses page structures of which is known to have a specific size. However, some of these pages can contain a specific page type that does not have a recognizable byte sequence at the start of a page. In terms of validation it is not possible in this case to distinguish the difference between data belonging to the same SQLite file or whether the data is from another file. Another example is used in the tiff file format, the Image File directory can be seen as a section that contains directory entries, however this section does not contain recognizable byte sequences to recognize this structure. The tiff file format uses file offsets to locate the different sections within the file. Thus, in case these offsets are invalid, for instance due to fragmentation, there are no means to find the section that is referred to.

References. The file format can specify structures that contain references to other parts of the file. The file format can use for instance an absolute file offset from the start of the file or use identifiers to refer to other parts of the file. The zip file format for instance uses relative file offsets in the central directory to refer to the location of the local file headers of each file entry.

Another type of reference is the use of unique identifiers. Unique identifiers can be used to recognize and identify a specific section. An example of the use of unique identifiers is used in the ogg file format. The ogg file format uses page sections that contain a bitstream serial number, which is a unique serial number that identifies the logical bitstream. Furthermore, a page sequence number, which is the sequence number of a logical bitstream (in case of audio, an audio track) is located in the page data structure.

References can be checked and validated by a validator. These references can also be used to identify and combine fragments of a fragmented file together.

Encoding. A file format can specify that data, which can be located in sections, has to be encoded using a specific algorithm. Also compression can be used in specific sections of the data. A file format validator can check whether sequences of data are encoded by decoding the data sequence.

An example of the use of compression algorithms is the use of the deflate method in zip files. An example of encoding can be found in the ogg file format: data stored in ogg files is encoded depending on the used encoder during the creation of the ogg file. Also other formats like JPG, PNG and MP3 use encoding. However, file formats do not always have to use encoding or compression. An example of this is the tar file format, this archive format can store other files within the tar file. The files stored within the tar file are not modified by compression or encoding, instead the original raw data of the archived file is stored in the tar file.

Hierarchy/Structure. The file format can dictate a specific structure or hierarchy with regard to the sections mentioned earlier. For instance, a file format can specify that a file always starts with a header section, followed by a data section and ends with a footer section. So not all possible combinations of sections are allowed by the file format in this case. The structure can be defined ranging from very detailed and strict to very simple and straightforward. The more structured a file format is specified, the more this structure can be validated by a validator.

A file format can specify a hierarchy for the sections that are stored within a file. For instance, the ooxml file format uses a specific naming scheme that has a defined hierarchy for the file entries. The ooxml file format is layered on top of the zip file format, but has specific rules regarding the names of the file entries and directory structure for the files stored in the zip file. Another example that uses hierarchy is the AVI file format: stream headers are always stored in the hrdl list structure and video frame data is always stored in te movi list structure.

A file format can specify a specific order in which data sections have to be present in a file. For instance, the JPG file format specifies a specific ordering, the file starts with a SOI section followed by a frame section and ends with a EOI section. The JPG file format also specifies, the contents and order of the segments within the frame segment. This file format does not allow another sequence of sections. However, file formats do not always have a strict order for each section and allow more freedom with regard to the ordering of sections. For instance, the PNG file format uses sections called chunks. Some chunks can be placed in any order and other chunks have to be in a specific order/location. For instance, the image header chunk (IHDR) and the image trailer chunk (IEND) need to be in a specific order and location. There are also file formats that have almost no restrictions on the ordering of sections. For instance, the SQLite file format uses pages as sections and only specifies that the file has to start with a database header structure, there are no rules regarding the ordering of the other pages in the file.

Note that the difference between ordering and hierarchy is subtle. Hierarchy can be seen as a constraint that results in a specific ordering, however ordering does not have to be hierarchical. Hierarchy is a parent-child relation between different sections in a file format. This hierarchy is a concept that is layered on top of the physical data. This means that the physical data first has to be parsed to be able to verify the rules of the hierarchy. The ordering of data sections can already be verified by analyzing the physical data. This is the difference between hierarchy and ordering.

Another type of ordering is the recurrence of specific data structures. For instance, a file format can specify that specific structures have to occur at certain intervals, for example the mp3 file format have a recurring sequence of frames. These frames start with a frame header and is followed by frame data, this is a very strict order of data section sequences.

The structure property (hierarchy/ordering/recurrence) can be checked by a validator. If the structure no longer adheres to the format this could be an indication that there is a fragmentation or corruption point reached in the data.

Error detection. The file format can specify fields that contain information of internal verification mechanisms, such as CRC checksums of specific parts of the data. For instance, sections or even the complete file. These CRC values can be used by a validator to validate parts of the data of the file under recovery, by calculating the CRC checksum of the raw data and by comparing the calculated value with the CRC value stored in the file.

The data coverage of the CRC checksum can vary between different file formats. For instance, the zip file format contains CRC checksums in each local file header for each file that is stored within the zip file. In this case the CRC checksum can be used to verify each file entry separately. If the file contained in the zip file is very big, the span of the CRC checksum is also big. However, there are also file formats that have CRC checksums on smaller amounts of data. For instance, the ogg file format has CRC checksums for each page section and also the mp3 file format can contain a CRC checksum for each frame section. The usability of the CRC checksum for pinpointing the location in a file that is corrupt, depends on the span of data of which the CRC checksum is calculated.

Metadata. The file format can specify fields that contain information regarding the data contained in the file. For instance, the size of a field or data section or information regarding the amount of data contained in the complete file. Examples of this are, information about the amount of data structures used in a file or the last issued unique identifier within the file.

For example, the database header in the SQLite file format contains the following metadata: the database page size and the amount of page sections stored in the database file. Another example is the header structure for each file in the tar file format, this header contains the size of the file and the name of the file which is contained in the tar file. Each local file header in the zip file format contains the compression method of the entry and the compressed size and uncompressed size and file name of the file contained in the zip archive. The AVI file format uses an AVI main header structure, this structure contains metadata of the AVI file, such as the file size, the total amount of frames stored in the file and the video dimensions in pixels.

As indicated by the investigated file formats, the range of information stored in the metadata can be very broad. The properties stored in the metadata of the file can be verified by a file format validator to detect inconsistencies.

6.3. File life cycle

Another relevant concept with regard to file formats, is the file life cycle of a specific file format. There are two types of life cycles:

- **Single write:** Files that are only written once upon creation of the file. For instance, audio and video files.
- **Multiple write:** Files that are rewritten multiple times during the life time of the file. For instance, documents, presentations and e-mail archive file formats.

A file format can contain specific concepts, in order to support multiple write life cycles. A file that is updated during its life cycle can contain sections which can be rewritten. This can be due to performance considerations. For instance, if a new email is added to an email archive, you want to prevent that the whole archive is rewritten from scratch. If a file format supports the reuse of data sections, the file format can also contain administration regarding allocated and available data sections. This could introduce another layer of complexity with regard to fragmentation. If a file format supports the reuse of data sections and does not clean the old contents of the data sections, fragmentation on a file level occurs (internal fragmentation). When restoring a file that can contain old and incomplete data sections of deleted data of the file itself, this introduces additional complexity during validation of the file.

In order to correctly validate files that support its own data allocations, the internal administration that contains the allocated and free data sections need to be parsed. In order to parse the allocation administration, the validator must be file format aware. The file format validator must take into account that non allocated data sections can contain valid, invalid, incomplete or no data. The file format validator must only verify the data sections which are allocated according the allocation information of the file.

File format concepts that can be used to support a multiple write life cycle are sections and metadata. The metadata concept is used to store the internal administration that is required to support a multiple write life cycle. Another property of the multiple write life cycle is that the file is only partially updated or data is appended to the file, to support this behavior the file must be organized in different sections. Thus the sections file format concept is typically used in file formats that support a multiple write life cycle.

Please note that the majority of the investigated file formats in Section 6.1 has a single write life cycle. Additional research might be required to further identify the relation between file format concept usage and the life cycle of a file format.

7

File format validation feasibility

This chapter introduces a method that can be used to determine whether the creation of a file format validator is feasible for a specific file format. The concepts used in file formats that were identified in the previous chapter are used for this. A mapping between file format validation and the identified file format concepts is described in this chapter. As a result an arbitrary file format specification can be analyzed for the presence of specific file format concepts. The identified file format concepts in combination with the method, can be used to determine the feasibility and added value of file format validation.

The previous chapter analyzed files from a file format specification point of view to identify file format concepts. This chapter focuses on the concepts that are required to construct a file validator, from a file validation point of view.

7.1. File format validation requirements

File carving is the technique of retrieving deleted files without using information from the file system. During file carving the storage medium is read cluster by cluster. Typically, a file header is used to determine whether a specific file is detected [PM09, Gar07]. The file can be stored contiguously on the storage medium, but can also be incomplete, corrupt or fragmented. The added value of a file format validator is to provide information about the validity of the detected file during file carving. Ideally the file format validator indicates from which point in the data the file is no longer valid to detect fragmentation points. However, if the data is no longer valid according the validator, this could also indicate that the file under recovery is corrupt. In this case the reason for this does not always has to be fragmentation. Fragmentation only occurs at cluster boundaries of the file system, this property can be used to distinguish between corruption and fragmentation. If a validator can distinguish between corruption points and fragmentation points, this information can be used to identify fragments of a file of a specific file format.

File format validation is a technique that can be used to recover fragmented files. In order to do this, a file format validator must be able to indicate whether a file under recovery is valid or not. In this case a file carver needs to provide data to the validator until the data forms a valid file, this approach potentially takes a long time, because of the size of the search space. However, if a file validator is able to pinpoint from which location the data is no longer valid, this information could be used to identify fragments. When file fragment identification is possible, this reduces the amount of combinations (search space) the file carver must check to recover a file, this improves the performance. Thus when using a validator in the context of file carving, file fragment identification and reconstruction is the added value of a validator in terms of performance and usability.

The majority of the files under recovery are stored non fragmented. In these cases the files under recovery consist out of one part and there are no fragments that have to be reorganized in order to be recovered, only the beginning and ending of the file needs to be identified. However, files that are fragmented and thus consist out of multiple fragments, need to be reconstructed to recover the fragmented file. The problem that file format validation for fragmented files must solve is:

- File fragment identification
- File fragment reconstruction

A file validator needs to be able to identify which file format is provided in order to validate the file according the corresponding file format specification. Recognizable data structures can be used by a validator to identify information within a file itself. Furthermore, the validator must be able to detect until which point the data belongs to the same file, this ability improves the detection of fragment boundaries. This ability can assist during the reconstruction of different fragments and differentiating the fragments of different files of the same file format.

One step further is to be able to verify the contents that are stored in the file, because a file can be valid according to its file format specification, but can still contain invalid data.

7.2. File format validation principles

The previous section introduced two problems that must be solved by file format validation: file fragment identification and the reconstruction of a file using identified fragments. The following file validation principles are identified to address these problems:

1. **File signature:** Allows the file format to be recognized by a file format validator.
2. **Recognizable data structures:** Allows a file format validator to recognize data structures within a file.
3. **Ability to match data with the same file:** Allows a file format validator to match data to a specific file.
4. **Ability to detect invalid/corrupt data:** Allows a file format validator to detect invalid or corrupt data within a file.
5. **Ability to check consistency across the complete file:** Allows a file format validator to perform consistency checks on file level.

The file signature and recognizable data structures validation principles are used to support the file identification requirement. The ability to match data with the same file, corrupt data detection and the consistency check validation principles are used for supporting the identification and reconstruction requirements.

File signature First of all a file format has to be recognizable, because this enables the selection of the corresponding file format validator and the possibility to detect files of a specific file format. In order to do this, the presence of a file signature is required. A file signature can be a sequence of bytes that is specific for the file format. This is the absolute minimum requirement to be able to recognize a file format.

Recognizable data structures The next principle to look for in a file format are data structures. Examples of data structures are:

- File header
- File segment within the file
- File footer

Recognizable data structures can be used to detect and identify parts of a file. File headers and file footers can for instance be used to determine the start and end of the file. Furthermore, the file format can dictate specific structures at specific intervals or locations in the file. Examples of these are headers and footers.

A file format can also specify data structures between the header and footer, these data structures can be seen as sections (sometimes also called: clusters, segments, frames, blocks or chunks) that have their own structure. If these structures are strict, they could potentially be validated by a validator. If the data no longer adheres to the structure or required interval defined by the file format, this could be an indication that there is a fragmentation or corruption point reached in the data. Data structures need to have a specific signature (byte sequence / magic string) in order to be recognized by a validator, otherwise the validator is not able to validate the data structures. An example of recognizable data structures are fragments (sections) containing audio data in the MP3 file format, these fragments appear at a specific interval and have a specific structure.

Ability to match data with the same file This ability is required to detect fragmentation or corruption points of the file during file carving. This ability is used to identify fragments and can also be used to reconstruct fragments. During file carving the data of a file is provided cluster by cluster, the file format validator must be able to determine whether two successive clusters belong to the same file, in order to identify fragments. This ability can also be used during the reconstruction of a file by validating two successive fragments.

There can be several concepts present in a file format that can be used to determine whether the next data still belongs to the same file:

- Encoding
- File format structure/hierarchy
- Fields that contain the length/size of data segments

A file format can specify that specific segments contain encoded data. This property can be used to check until which point the data is still encoded, if this check fails this can be an indication of a fragmentation or corruption point. Another property that can be used, is the presence of a physical layout specified by a file format. A file format can specify a global structure, to which a file has to comply. The physical layout can be used to identify whether a file is still valid with respect to the file format specification. The required physical layout of the file format can specify structure and/or hierarchy that can be verified for consistency. If the structure or hierarchy is no longer consistent with the file format, this could also be an indication of a fragmentation or corruption point. A file format can specify that only specific hierarchies are valid or dicatate a specific sequence of data structures. These properties can be validated by a validator. Another example is that a file format can specify that specific fields contain data regarding the length of data segment or that fields have a specific lengths. A validator can check these length properties to validate whether the data is still adheres to the file format.

Ability to detect invalid/corrupt data This property has overlap with the previous ability, since this ability can also be used to determine whether the next data still belongs to the same file. However, this ability specifically refers to the availability of internal verification mechanisms specified by the file format. The difference with the previous ability is that this ability not only checks for valid hierarchy or data structure sequences, but actually verifies the data contained in the segments. This is a significant difference, since a data segment can be valid according the specified data structure (for instance, a valid footer and header section), but the actual data contained in the data segment can be invalid. The following concepts can be used for internal verification:

- Internal verification mechanisms
- Restrictions imposed by the file format on the content of data structures

Internal verification mechanisms are typically present in the form of CRC values. The file format can specify the presence of a CRC value of a specific section of the file. These CRC values can be used to compare the CRC value with the computed CRC of the data. If the check passes, this is an indication the data is still valid and part of the file. If the check fails this can be an indication of a fragmentation or corruption point. If the data format specifies data structures in a specific format, these data structures can also be validated for consistency with respect to the restrictions imposed by the file format. If this consistency check fails, this can be an indication of a fragmentation point. An example of this is that a specific field is only allowed to have a limited set of valid values. If this field does not contain a valid value, this could be an indication of a fragmentation or corruption point.

Ability to check consistency across the complete file A file can consist out of valid segments, however combining valid segments in a random order does not have to result in a valid file. The file format can contain metadata which enables validation on file level. The following concepts can be used on as check on file level:

- CRC checksum of parts (or complete) of the file
- Absolute (or relative) file offsets within the file
- Unique identifiers (numbers, names) for data sections
- References to other sections in the file
- File size metadata
- Data structure/hierarchy at file level

A file format can specify CRC values of larger parts of the file than the earlier mentioned segments of a file. Once more parts of the file are recovered or even the complete file, this allows validation on file level instead of on section level.

A file format can use references to other sections in the file, references can be implemented by using absolute or relative file offsets within the file. These file offsets can be used to check the consistency within the file, by comparing the physical position in the file under validation with the file offset stored in the file.

References can also be used to identify and combine fragments of a fragmented file together. References to other sections in the file can also be implemented by using unique identifiers. A file format can specify that data structures have a unique identifier, these unique identifiers can be used to validate the consistency of the file.

A file format can specify areas that contains file size metadata. File size metadata can be regarding segments of the file or even the complete file size. This metadata can be used by a validator to validate the consistency of the file. An example of this metadata, is the complete file size. In case the complete file size metadata is present, this value can be compared with the amount of data that is recovered.

The techniques described above can also be used on fragment level, this can be seen as the ability to match data with the same file, however some metadata can only be verified in case the complete file is available. For instance, the verification of a reference is only possible if the location that is referred to is part of the recovered data, this is not always the case. Furthermore, some metadata, like file size or the amount of sections, can only be verified in case the complete file is available. Therefore this ability is identified as a separate ability.

7.3. Mapping between validation principles and file format concepts

Section 5.1 contains a list of existing file format validation techniques. In order to apply these validation techniques, certain file format concepts need to be present in the file format. Section 6.2 contains a list of identified file format concepts that were found during the file format analysis. Table 7.1 contains the mapping between validation techniques and the required file format concepts. This table also contains the mapping between validation techniques and the identified file validation principles, that are introduced in Section 7.2. These file format validation principles are used to recognize and reconstruct fragments, in order to successfully carve fragmented files using file format validation. In order to bridge the gap between file format validation principles and file format concepts, there are two intermediate relations identified: the first one is the relation between file format validation principles and file format validation techniques. The second one is the relation between file format validation techniques and file format concepts.

Table 7.1 provides an overview about which file format concepts can enable the use of specific file validation principles. Section 7.4 provides the minimum set of required file validation principles for file validation, using Table 7.1 the corresponding file format concepts that enables these file validation principles can be found.

7.3.1. Relation between validation principles and validation techniques

Section 7.2 introduced the file validation principles. These file validation principles can be supported by file validation techniques. This sections describes which file validation techniques can be used to support a specific file validation principle.

File signature A file signature is something that can identify a file format. A file format can use a specific header and/or footer, therefore the header/footer validation technique can be used to identify a file format. Furthermore, a file specific magic number (or string) is typically present in a file format specification, therefore the magic number matching validation technique can also be used to identify a file signature.

Recognizable data structures In order to recognize and identify data structures of a specific file format, the following validation techniques can be used: Header/Footer validation, magic number matching and container structure validation.

Header/Footer validation relies on the specific data structure of a header and footer, therefore this technique can be used to recognize data structures.

Typically, the recognizable part of a data structure are specific sequences or locations of values in the data. These recognizable data values can be identified by using the magic number matching technique.

Container structure validation checks the properties of data structures that are specified in a file format, for instance the length of a field or the type of data that is stored in a field of a data structure. As a result, the container structure validation technique can be used to identify recognizable data structures.

Ability to match data with the same file Several validation techniques can be used to match data with the same file, it depends on the data under analysis which technique is most suitable.

In case data the is compressed, the validation with decompression technique can be used. If the data is encoded, the algorithm output analysis technique can be used.

If the data is not encoded or compressed, the data dependency resolving technique can be used, this technique uses information retrieved from the file data to check the consistency of the file under analysis, for instance by checking references or the length of a section.

If the file format uses container structures, the container structure validation technique can also be used. Container structure validation checks the specified properties of a container data structure. For instance, the length of a field or a value within a field of a container data structure. When data is matched to the same file, container data structures need to be valid. If this is not the case, this could be an indication the data does not match with the same file.

Ability to detect invalid/corrupt data In order to detect invalid/corrupt data, internal verification mechanisms are used, this can be accomplished by using the internal verification checking validation technique. An example of an internal verification mechanism is a comparing a CRC value of the data with a calculated CRC value.

Another method of detecting invalid data is by using the container structure validation technique. This technique can be used to verify whether the structure of the data structures is correct and contains valid values in case these are specified by a file format. For instance, a field in a data structure can have a specific length and value.

Ability to check consistency across the complete file Multiple validation techniques can be used to check the consistency of a complete file. The following techniques can be used to verify the consistency across the complete file or parts of the file: semantic validation, internal verification checking, algorithm output validation, validation with decompression, data dependency resolving and container structure validation.

Semantic validation can be used to verify whether the semantic information is consistent across the complete file. Internal verification checking can be used in case there is information available of the complete file (or parts of the file), for instance a CRC value of the complete file. Algorithm output validation can be used to verify whether the complete file can be decoded and is consistent. Validation with decompression has a similar approach except it tries to decompress compressed data of the complete file.

If the complete file is available this also allows the data dependency resolving technique on a file level, instead of on a fragment. If a complete file is available, all references can be validated or metadata of the file on file level. Metadata can for instance contain information about the amount of data structures of a file or the file size, these can only be checked if the complete file is available.

Container structure validation can be applied within sections itself, but can also be applied on structures that exists out of multiple sections across the complete file. For instance, if a file format consists of multiple container structures that are nested in each other.

Since a file can consist out of different sections with different properties for each section, the used validation technique depends on which part of the file is analyzed. For instance, a zip file consists of a section containing references and sections of compressed data. Validation with decompression can be used for the compressed data sections and data dependency resolving technique for verifying the references.

7.3.2. Relation between file format validation techniques and file format concepts

The next step in mapping validation principles to file format concepts, is the dependency between the file format validation techniques and file format concepts as introduced in Section 6.2. In order to apply validation techniques, specific file format concepts need to be present. Section 5.1 contains the list of validation techniques, the following techniques are not applicable since they do not apply in the context of file format validation: manual validation, Bifragment Gap Carving and Metadata Based Data Recovery.

Manual validation relies on manual actions and is not used in combination with a (automatic) file format validator. Bifragment Gap Carving (BGC) is a technique than can use a file format validator to identify a gap between two file fragments of the same file, this technique is a method to solve the reconstruction problem of file fragments on a file carver level. Thus Bifragment Gap Carving is not directly applicable within file format validation. Metadata Based Data Recovery relies on the availability of metadata of the file system on which the data under recovery is stored. This technique cannot be applied in a file format validator, since a file format validator handles the data of the file under recovery and does not use information of the file system. Furthermore, metadata based recovery is also a technique which is applied on a file carver level, not on file validation level.

Header and footer validation Relies on the use of recognizable data structures that are required to recognize the header and footer sections that are used to identify a file under recovery. This means that the required file format concept to apply header and footer validation is the sections concept.

Magic Number Matching Relies on the usage of recognizable values (magic numbers, magic strings) in a file format. Typically magic numbers are used in combination with recognizable data structures. Data structures are recognizable due to the usage of magic numbers. Therefore, if a file format uses the sections concept, the magic number matching validation technique can be applied to recognize data structure.

Please note that magic number matching can also be used without sections, for instance in order to recognize a file signature. So recognizable data structure requires magic numbers, in order to be recognizable. But magic numbers do not necessarily require data structures, since magic numbers can appear anywhere in a file format.

Container structure validation Validates the structure of container data structures. This requires recognizable data structures to validate the content of a data structure, thus the sections file format concept is required.

The container structure validation technique can also be used to verify a hierarchy within a file format, this allows container structure validation on file level.

Data Dependency Resolving Multiple file format concepts can be used to apply data dependency resolving: references, metadata and hierarchy/structure.

The references file format concept can be used to verify the reference, by checking whether a specific data structure is present at the location the reference is referring to. Metadata can also be used for applying the data dependency resolving technique, for instance the size of a specific data structure can be checked. If a file format uses a specific hierarchy, this hierarchy can also be used by data dependency resolving to check whether the hierarchy is valid within a file.

Validating with decompression Relies on the usage of data compression. This technique tries to successfully decompress data to validate the data. In order to do this the data must be compressed, thus the encoding file format concept is required.

Algorithm Output Analysis Is a different approach to validating with decompression. This technique tries to match data together by applying bit sequence matching of data that was encoded using a specific algorithm, for instance encoded video data. Thus this method relies on the presence of the encoding file format concept.

Internal verification checking The internal verification checking technique relies on the presence of error detection, for instance CRC checksums or other types of checksums that can be used to validate the contents of the data. The technique computes the checksum of the data and compares this with the checksum that is present in the file itself. Thus the internal verification checking technique requires the error detection file format concept.

Semantic validation Uses semantic information from the file format. For instance, the language that is present in a file. In order to apply semantic information the following file format concepts can be used: metadata and sections. Metadata can be used to extract semantic information, such as the language. This semantic information can be used to check whether the data contains the same language. Semantic information can differ among different sections, therefore the sections file format concept can be useful to differentiate which semantic information applies to that specific section.

Table 7.1: File validation and file format concepts mapping

Validation principle	**Validation technique**	**File format concept**
File signature	Header/footer validation	Sections
	Magic number matching	Sections
Recognizable data structures	Header/footer validation	Sections Hierarchy/Structure
	Magic number matching	Sections Hierarchy/Structure
	Container structure validation	Sections
Ability to match data with same file	Validating with decompression	Encoding
	Algorithm Output Analysis	Encoding
	Data Dependency Resolving	References Metadata Hierarchy /Structure
	Container structure validation	Sections
Invalid/corrupt data detection	Internal verification checking	Error detection
	Container structure validation	Sections
File consistency	Semantic validation	Sections Metadata
	Internal verification checking	Error detection
	Algorithm Output Validation	Encoding
	Validation with decompression	Encoding
	Data dependency resolving	References Metadata Hierarchy/Structure
	Container structure validation	Hierarchy/Structure

7.4. Necessary file validation principles

In order to determine whether file format validation is feasible and to what extent it has added value are two different subjects. The added value depends on the level of detail the feedback is provided by a file format validator, regarding from which point the file is no longer valid. Creating a score model that gives an accurate indication of the added value is difficult, since this analysis is not straightforward and highly depends on the context and conditions. However, it is possible to specify an absolute minimum set of requirements to perform file format validation.

As mentioned in Section 7.1 a file validator must solve the following problems: file fragment identification and file fragment reconstruction. This means that in an arbitrary bit stream a file format must be recognized by a validator. Furthermore, the validator must also be able to indicate from which point the data is no longer valid, possibly due to corruption or fragmentation. In order to fulfill these requirements, the following concepts are the minimum requirements that have to be present in a file format to perform file format validation:

- File signature
- Recognizable data structures
- Ability to match data with the same file

A false negative is rejected data that actually belongs to the file. A false positive is accepted data by the validator that actually not belongs to the file.

The concepts mentioned above are required to verify whether data still adheres to the file format specification, on a structural and hierarchical level. These concepts do not provide validation on the contents stored within the file. This means that if data contains the correct structures with respect to the file format specification, the information stored in the file could still be invalid or unreadable.

7.4.1. File signature

The file format needs to have at least recognizable byte sequences or magic strings that can be used to recognize the file format. These values do not necessarily need to be present in the header, but also other data structures can be used, as long as they can be used to recognize the file format. If the file format cannot be recognized from a file's raw data, it is not possible to validate the data with regard to a file format specification, since it is not clear which file format specification should be used to perform the validation.

7.4.2. Recognizable data structures

Recognizable file format structures can be used to verify whether the data still adheres to the file format. If the file format contains no distinguishable features, the data can be anything, this makes it impossible to create a file format validator. Please note that the definition of recognizable is as broad as possible, this does not has to mean that the structures need to have headers and footers. For instance, if it is known that a specific section of the file format is always encoded in a specific manner, this property can also be used to recognize the data structure and sections within the file.

7.4.3. Ability to match data with the same file

This concept is a continuation of the previous concept. Since the previous concept can be used to determine from which point the data no longer belongs to the same file. However, if these data structures allow big sections of arbitrary and undefined data, the file format validator has no means to verify this data. For instance, if a file format specifies a file structure that has a recognizable header and footer and allows an arbitrary data section with an undefined length in between, it is impossible to indicate from which point the data section is no longer valid. This can even be a problem if the length of the data section is specified by the file format, the validator can check the presence of the footer at the expected location using the length attribute. However, the problem arises when this footer is not present at the expected location, because this means that from some point in the data section the data of the file is no longer valid. In case the data section contains no properties that can be verified by the validator, the corruption point can be anywhere in the data section, this creates a degree of uncertainty. The amount of uncertainty a file format validator has, should be at least in the range of the cluster size, because fragmentation occurs at cluster boundaries. If the uncertainty of the file format validator is a multitude of the cluster size, the added value of the file format validation is restricted.

7.5. Additional file validation format principles

The remaining principles can be used as additional checks on top of the mentioned principles in the previous section, the remaining principles are not required:

- Ability to detect invalid/corrupt data
- Ability to check consistency across the complete file

The validation principles above are not part of the minimum required set of validation principles, because file validation is also possible without application of the additional validation principles. The additional principles offer more precision and information, however they do not have to be present in order to validate a file format. The availability of more validation principles is always better, however this is not always realistic. Since a file format does not always allow the usage of each validation principle. In order to prevent that validation is considered unfeasible for a lot of file formats, we differentiate between minimum and additional validation principles. Validation is feasible if only the minimum requirements are present, but the level of detail might be restricted compared to a file format that also allows the additional validation principles.

Both validation principles mentioned above can be used to check the contents of the file. The proposed minimum set of validation principles is used to validate the file on standalone data structure (segment) level, but does not state anything about the contents stored in the file. The two remaining additional principles provide means to check the file on a content level. If a file is validated according the minimum set of validation principles the file adheres to the structure that is mandated by the file format specification. The file however can still contain corrupted data stored in the data structures itself, since this can occur without breaking the rules of the file format specification. The additional validation principles also verify the contents stored in the file.

7.5.1. Ability to detect invalid/corrupt data

This ability is not a minimum requirement for file format validation, because if there are no internal verification mechanisms specified by the file format, the file format validator could still depend on other concepts to validate the file. For instance, the validity of data structures and hierarchy of data structures. Internal verification mechanisms typically provide information about the consistency of the data stored in the file. If internal verification mechanisms are not available, the file format validator cannot verify whether the data is corrupt or valid. If a corruption point occurred somewhere in the data stream, this could be detected by an internal verification mechanism, however this can also be accomplished by verifying data structures. For instance, if a data section has a header and a footer (or a specified length) these data structures can be used to detect corruption points. An internal verification mechanism goes further than this, since it can check whether data is corrupt or valid and thus can be used to provide a confirmation on top of the data structure consistency check (double check). Internal verification mechanisms are very useful since they provide confirmation about the correctness of the data of a specific section, but from an absolute minimum requirement viewpoint, they are not required. The trade-off of omitting internal verification mechanisms is that there is no guarantee that a file that adheres to the file format specification, contains valid contents. It is possible that a file contains valid ranges of data structures, but the data contained in those data structures is invalid.

7.5.2. Ability to check consistency across the complete file

The techniques and principles mentioned in this category, provide additional means to validate the integrity of a file on top of the validation of standalone data structures, therefore this ability is not a requirement for file format validation. A trade-off of omitting this principle is, that a file validator cannot longer detect the following case: a file that consists of valid segments does not have to result in a valid file, because combining valid segments in a random order can be valid according the file format specification but does not have to result in a valid file. The ability to check the consistency across the complete file can be used to check the consistency of the file on top of (double check) the data structure validation. Therefore, this principle is not a minimum requirement.

7.6. Challenging file format concepts for validators

The principles mentioned above all contribute to the identification of positive and negative constraints as introduced by Cohen [Coh07]. A file format can provide structure and restrictions that specify positive and negative constraints that can be used during validation by a file format validator. However, if the file format does not specify recognizable structures and does not impose structures and restrictions, less constraints can be identified. The absence of negative and positive constraints, potentially results in a less effective file format validator. The following concepts in a file format can be a problem for file format validators:

1. **Undefined length of sections:** If the file format allows sections of undefined length that are not marked with recognizable structures. The length of a section can be specified by field or dictated by the file format, in this case the completeness of a section can be checked. However if the data can be of arbitrary length and does not recognizable data at the beginning and ending, the dimensions sections cannot be identified.

This can for instance form a problem in case sections are validated using a CRC value, but also during the validation of structures with regard to the file format specification.

2. **Uninitialized/Undefined data:** If the file format allows sections of undefined data, the validator cannot verify the data since the data can be anything. For instance, if the file format can reuse data sections, but does not specify that the data of a section has to be cleared when the section is freed. Or when a file format can allocate new data sections that still contain old (undefined) data.
3. **Non-encoded data:** Decoding can be used to check which data belongs together. If the file format allows long sections of non-encoded data, it is more difficult to identify from which point the data no longer belongs together. For instance, if a file format allows data sections with non encoded data that has the size of a couple of clusters and a CRC value is available, the CRC value can be used to check if the data is consistent. However, the exact corruption point cannot be identified. Since the corruption point can be at any place in the non-encoded data stream. This property increases the amount of uncertainty of a file validator.
4. **Unrecognizable data structures:** File format validation relies on the recognition of data on byte sequence level. If the file format contain sections, that cannot be recognized on byte sequence level, this forms a problem for file format validation. Since if the file format validator cannot recognize the sequence, it can also not check whether the sequence is valid with respect to the file format.

To what extent these file format concepts actually form a problem during validation depends on the situation in which the concepts occur in the file. As mentioned in the concepts above, if the size of the data affected by the concepts described above is smaller than the cluster size of a file system, the impact on file validation is limited.

The presence of uninitialized or undefined data does not have to be a problem in case the file validator is aware of the location and presence of these sections. In order for the file validator to be aware of this, the file format specification must specify the location of uninitialized or undefined data or specify in which section of the file the location of the uninitialized or undefined data is stored (metadata file format concept). In case the location of uninitialized or undefined data is stored in the metadata of the file, the validator must be able to parse this information from the file data.

Unrecognizable data structures form a problem that cannot be solved by applying other concepts, because file validation relies on the recognition of byte sequences in a stream of arbitrary data. If the byte sequences are not recognizable the file validator cannot function. The same applies to the presence of sections with an undefined length, because in this case the validator can also not recognize the presence of a specific data structure and as a result cannot validate the conformance to the file format specification.

7.7. Validation case study: WAD file format

This chapter introduced a method on what to look out for when analyzing a file format specification. The method can be used to determine whether file format validation is feasible and can have an added value for identifying corruption points. In this chapter the WAD file format is analysed to determine whether the identified concepts and categories are usable and complete enough to do a feasibility study for creating a validator for a file format.

The method to analyze a file format specification with regard to the feasibility of file format validation is applied to the WAD file format. The categories for analyzing the feasibility identified in Section 7.4 and 7.5 for creating a file format specific validator are used in combination with Table 7.1. The WAD file format specification is analyzed to identify which file format concepts, as introduced in Section 6.2, are present. The identified file format concepts of the WAD file format are mapped to the different file validation principles using Table 7.1.

The goal of this exercise is to verify the usability of the identified validation principles in combination with the file format concepts, by using Table 7.1. The WAD file format is used as a case study, if the method of combining file format concepts with file validation principles is correct, this serves as a proof of concept of the method.

7.7.1. WAD FILE FORMAT SPECIFICATION

The WAD file format is used by doom and games based on the doom engine for storing data. The following information is retrieved from the Doom Wiki[1]. WAD is an acronym which stands for Where's All the Data. The WAD file format consists of 3 segments:

- Header
- Directory
- Raw data of the resources stored in the WAD file (lumps)

Figure 7.1 provides a schematic overview of the WAD file format.

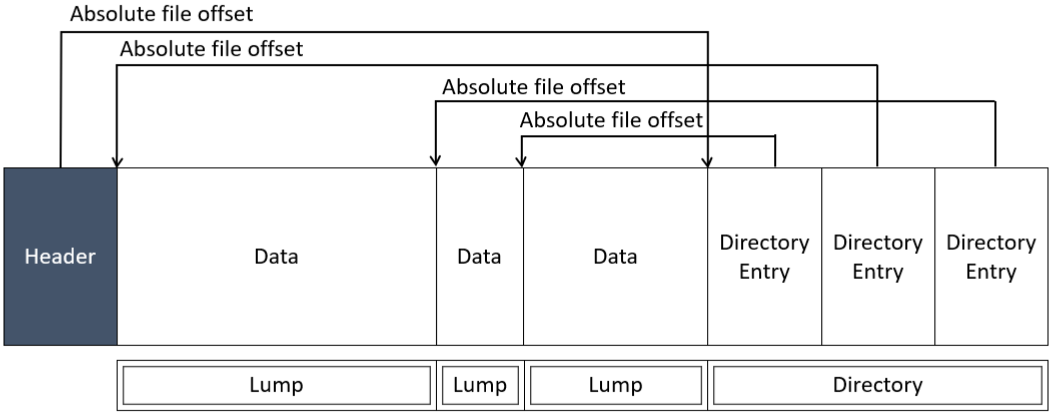

Figure 7.1: WAD file format diagram

Header data structure The header consists of 3 fields:

- Magic string, ASCII string ("PWAD" or "IWAD")
- Number of entries in the directory (Integer value)
- File offset to the location of the directory structure (Integer value)

The ASCII string ("PWAD" or "IWAD") can be used for recognizing the file format and detecting the header of a WAD file. The header also contains a field that contains the number of entries in the directory structures and contains a file offset to the location of the directory.

Directory data structure The directory data structure contains file entries of the resources that are stored in the WAD file. Each entry in the directory structure has a fixed length and

[1] https://doomwiki.org/wiki/WAD

the number of directory entries is specified in the header of the WAD file. A directory entry contains three fields:

- File offset to the start of the data of the resource (4 bytes integer value)
- File size of the resource (4 bytes integer value)
- Name of the resource (8 bytes ASCII string)

The directory structure does not contain byte sequences which can be used to identify and recognize directory entries. The only restriction the file format specifies is a specific length and that each entry contains 2 integers followed by 8 bytes containing an ASCII encoded string. Also the number of entries can be derived from the file header. So the only information the file format provides regarding the directory structure is the number of entries and the length and data types of the entries. Only this information can be used to validate the directory structure. Both the header and directory do not contain CRC values that can be used to validate the data. There seem to be restrictions and naming conventions for the naming of the resource. If there is a limited set of valid values for the name of the resource this string can be used as a magic string to recognize directory entries.

Resource data Resource data must be of a specific type (for instance, a level map or a sprite). A data resource is called a lump. Lump items are of a specific type, lump types may use markers that can be used as magic strings to recognize. The format of most of the data lump types consists of a series of integer values, without any magic strings or specific byte sequences. The file format sometimes specifies a hierarchy for data lumps, for instance when using map data lumps. However, the sequence of data lumps is not always restricted in the WAD file format.

7.7.2. WAD FILE FORMAT VALIDATION FEASIBILITY

The WAD file format contains the following file format concepts as introduced in Section 6.2:

- Sections: header, directory and lumps
- References: file offsets in directory to lumps and file offset of the directory in the header
- Metadata: number of file entries, data resource lengths and resource name

Necessary file validation principles The minimum set of required file validation principles for validation are: file signature, recognizable data structures and the ability to match data with the same file.

- **File signature:** The file format uses sections and the file signature of the WAD file format is present in the header.
- **Recognizable data structures:** The WAD file format contains sections that use data structures: file header (recognizable), file directory (recognizable to some extent, since this depends on the available possible valid entries for the name field) and data lumps. Data lumps can be identifiable to some extent, this highly depends on the type of data and to what extent the file format specifies identifiable sections for each type of data lump.
- **Ability to match data with the same file:** The WAD file format uses references and sections. These references cannot always be verified, since the section it is referring

to is not always recognizable. Furthermore, the WAD file format does not use encoding or compression, so this property cannot be used to match data together. The file format structure is limited and difficult to recognize based on byte sequences. The file format does not specify a strict hierarchy. There are fields that contain the length of data segments. However, the identification of those field can be challenge, since the file format does not contain any magic strings to identify the beginning and ending of data structures.

The WAD file format thus satisfies the file signature validation principle, but not every data structure is recognizable. Furthermore, the ability to match data with the same file is limited, due to the lack of file format concepts that can be used to support to ability to match data with the same file.

Additional file validation principles The additional set of file validation principles are: the ability to detect invalid/corrupt data and the ability to check consistency across the complete file.

- **Ability to detect invalid/corrupt data:** The WAD file format does not contain internal file verification mechanisms (error detection), such as CRC values. The sequence of data structures is not always restricted and depends on the type of data. For instance, the map format contains requirements with regard to the data structures that are required and also the sequence is specified.
- **Ability to check consistency across the complete file:** The WAD file format does not contain a CRC value on the complete file contents. However, the WAD file format does contain references (file offsets) to refer to other parts of the file. The directory structure contains references to the start of data lumps stored in the file. These references can be verified to determine whether a known data structure starts at this location. However, the possibility to verify references is limited, since the identification of the start of a data lump might not always be possible. The ability to recognize the start of a data lump varies between the type of data lump.

The WAD file format is very limited in recognizing invalid or corrupt data, due to the lack of error detection. Furthermore, the ability to check consistency across the complete file is also limited. Despite of the use of references by the file format, these references cannot always be used for consistency checking. Since this references cannot always be verified, because the referred section is not always recognizable.

Challenging file format concepts for validators

1. **Undefined length of sections:** The file format specifies the length of data structures. In case the length of a data lump is arbitrary, the lump contains a field that contains the length. However, the recognition of the field containing the length of the raw data can be a challenge.
2. **Uninitialized/Undefined data:** The file format specification does not mention anything about allocating new data. It looks like the WAD file format is used as an archive that does not contain unused data sections and therefore does not contain uninitialized data.
3. **Non encoded data:** The file format does not use encoding or compression.
4. **Unrecognizable data structures:** The file format contains unrecognizable data structures. The file header is recognizable by using a magic string, but the other data

structures like the raw data lumps and the directory are not recognizable. At best the directory can be recognized, by detecting a series a ASCII strings at certain intervals. However, the data lumps sections cannot easily be recognized, since most of the data lumps contain a sequence of integers without any recognizable markers or magic strings. Typically, the majority of the content in the WAD file is lump data, which can be difficult to recognize.

WAD file format validation feasibility

The minimum required set of concepts for file format validation is not met. The file signature is present, but the used data structures within the file are not always recognizable. The ability to match data with the same file is also limited and not always possible. The additional validation principles can also not be supported for the WAD file format.

Furthermore, the file format contains properties that make it more difficult and complex to perform file format validation.

Therefore, the creation of a WAD file format validator that is able to provide detailed information of the location of fragmentation points or the recognition of WAD file fragments is not feasible. The file format does not provide enough structure and concepts that can be used by validation techniques to verify the validity of a WAD file.

7.7.3. Usability of the feasibility method

The validation principles identified in Section 7.2 to analyze the feasibility of creating a file format validator, that can identify fragmentation or corruption points, were applied on a new file format that was not included in the file format concept identification investigation in Chapter 6. During the analysis of the WAD file format specification, the used file format concepts were identified. These file format concepts could be used to identify validation principles by identifying the corresponding validation techniques. The validation techniques provided a mapping between the used concepts in the file format specification and the consequence with regard to validation and verification.

The file validation principles are derived from a file validator point of view, by answering the question what should a file validator be able to do. The other point of view is from a file format specification perspective. By analyzing a file format specification, file format concepts could be identified. File format concepts are used by file validation techniques, thus a link between file validation techniques and the required file format concepts can be made. File validation principles can make use of different file validation techniques to achieve a file validation principle. As a result another link between file validation principles and file validation techniques can be identified. These two connections result in a mapping between file validation principles and file format concepts.

Due to this mapping, the file validation principles can be used to come to a substantiated conclusion with regard to the feasibility of creating a file format validator. The file validation techniques are the available methods that can be used to perform file validation. If there is no available file validation technique to verify a specific part of a file format, file format validation is not feasible. The unavailability of a validation technique can be detected by using the mapping in Table 7.1.

The proposed method seems to be complete for analyzing the WAD file format specification. However, the WAD file format used to test the method is relative simple and has a single write life cycle. In order to further validate the completeness of the proposed method, the method is applied on a complex file format with a multiple write life cycle: the PST file format.

8

PST FILE FORMAT VALIDATION

The Personal Storage Table (PST) file format is an open proprietary file format, which is developed by Microsoft. Documentation for the PST file format is publicly available [Mic20]. The file format is used by Microsoft Outlook to store e-mails and calendar items.

The PST file format is an interesting target for file format validation, because research of Van der Meer [vdMJvdB20] discovered that PST files are frequently fragmented. A corpus of 220 Windows laptops was analyzed and the findings were that 35.8% of the present PST files were fragmented. Furthermore, a significant amount of PST files was fragmented in more than 2 fragments and these fragments were stored out-of-order. The amount of fragments and the degree of fragments that are out-of-order increase the complexity of recovering a deleted file using file carving. The combination of the complexity and the relative high amount of fragmented PST files, makes the PST an interesting target from a file carver point of view.

Another reason that makes the PST file format an interesting target is in the context of digital forensics. PST files can contain e-mails and calendar items that can contain useful information or evidence in case of digital forensics.

8.1. PST FILE FORMAT SPECIFICATION

8.1.1. LOGICAL ORGANIZATION OF THE PST FILE FORMAT

A PST file contains a message store, which is used for storing folder objects. Folder objects contain message objects, an example of a folder object is a mailbox. A message object is a set of properties that represent an email message, appointment or contact. A message object has an attachment table that represents the attachments attached to the message object. An attachment is represented by an attachment object, this object is a set of properties that represent a file, structured storage or a message object. The properties of the different objects (Folder object, Message object, Attachment Object) combined provide all the information of a specific item stored in the message store. The PST file format contains the following logical layers:

- Messaging layer: Message store, Folders, Messages, Attachments
- LTP Layer: Heap, BTree, Property bags, Tables
- NDB Layer: Node database, lower-level storage of the PST file format

The logical layers have a hierarchy and are build on top of each other. The lowest logical

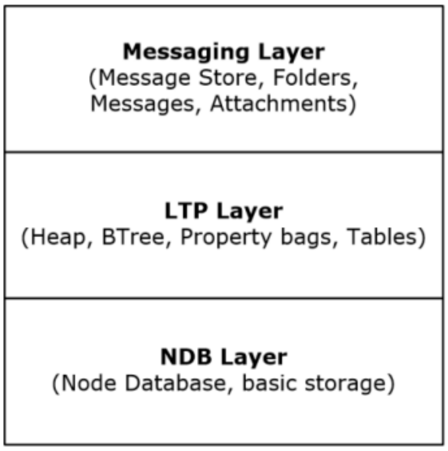
Messaging Layer
(Message Store, Folders, Messages, Attachments)
LTP Layer
(Heap, BTree, Property bags, Tables)
NDB Layer
(Node Database, basic storage)

Figure 8.1: PST file logical structure, source: [Mic20]

level is the Node Database (NDB) layer. In order to understand how the different logical layers interact and depend on each other, the layers are introduced from bottom to top.

Node database layer The Node Database (NDB) layer is a logical representation of the lower-level storage of the PST file format. The data stored in the PST file is accessible using nodes. Nodes provide an abstraction that is used to reference to data that is stored in the PST file that is interpreted from higher logical layers, thus not by the NDB layer itself.

Nodes are used to divide the data stored in the PST file into the different logical streams. A node consists of a stream of bytes and a collection of subnodes. The stream of bytes of the node is stored in the NDB layer as a data block. The collection of subnodes is stored using a subnode binary tree. The NDB layer contains a database of nodes. In order for higher logical layers to access the nodes that are required to access the lower-level storage of the PST file format, the NDB layer uses two binary trees:

- NBT: the Node Binary Tree contains the references to all of the accessible nodes of the PST file and contains information about which blocks need to be combined to form nodes. The NBT is optimized for locating a specific node efficiently.
- BBT: the Block Binary Tree contains the references to all of the data blocks that are used in the PST file. The BBT is optimized for locating a specific block efficiently.

By using both binary trees, the NDB layer can access the corresponding data that belongs to a specific node. The references to the roots of both the NBT and BBT are stored in the header of the PST file. The NDB layer consists of the following components:

- PST file Header
- File allocation information
- Blocks
- Nodes
- Node Binary Tree (NBT)
- Block Binary Tree (BBT)

Blocks are used to store data of the PST file, more information about blocks is provided in the block data structure section.

Lists, Tables and Properties layer On top of the NDB layer, the Lists, Tables and Properties (LTP) layer is implemented. The LTP layer adds higher-level concepts on top of the NDB layer to provide the structures that are required to represent messaging related objects, such as Folder Objects, Message objects and Attachment objects. The LTP layer has two core elements: the Property Context (PC) and the Table Context (TC). The PC is a collection of properties and the TC is a two-dimensional table, this table contains property-value pairs.

Messaging layer On top of the LTP layer the messaging layer is located. The messaging layer contains the higher level abstraction that uses both the LTP and NDB layers to handle folder objects, message objects and attachment objects. The messaging layer defines how the PST file is allowed to be modified. Furthermore, the messaging layer provides an interface that allows operations on the PST file.

8.1.2. Physical organization of the PST file format

The concepts introduced in the previous chapter are logical layers, which are abstract views on the actual data that is eventually stored in the PST file. This chapter discusses how the data is physically stored in the PST file. Figure 8.2 and 8.3 provide a schematic overview of the PST file format. Appendix B contains the grammar of the PST file format, this provides a more detailed description of the fields present in the PST file format. A PST file physically consists of the following parts:

- Header element: contains metadata of the PST file and information on how to access the data sections that contain the message store, which in its turn contains the e-mails and attachments. The header contains the location of the root of both binary trees, the NBT and BBT, which are used in the NDB layer.
- Allocation information at specific intervals (AMap, PMap, FMap, FPMap)
- Data sections, are sections of data of roughly 250 kilobytes in size. These data sections contains the allocations that are used by the PST file format to store data. Each allocation is aligned to a 64-byte boundary and is always a multiple of 64 bytes. The data section can contain non allocated (free) data and allocated data that is used to store two types of structures: the page and block data structures. The page and block data structures are used to store the metadata of the NDB layer and contain the data of the objects that are stored in the message store.

Each logical layer introduces new concepts and does not introduce new data structures, because it uses the existing data structures from the lower levels. As a result, there are only two different types of data structures that in the end contain all the information of the different layers. The NDB layer is the lowest logical layer and uses two types of data structures, the page and block data structure. These two structures are used to store the two binary trees, the Block Binary Tree (BBT) and the Node Binary Tree (NBT), and the nodes. Thus, from a physical organization point of view there are only two types of data structures present in a PST file, besides the PST file header: the page and block data structures. This overview can be seen in the schematic overview of the PST file format in Figure 8.3.

Please note that the PST file format allows the deletion and reuse of areas within a PST file, as a result free sections can be present. These free sections are marked as the grey shaded sections in Figure 8.3. The file format has to keep track of the allocation status of the PST file, because the file format supports the reuse of data sections, this explains

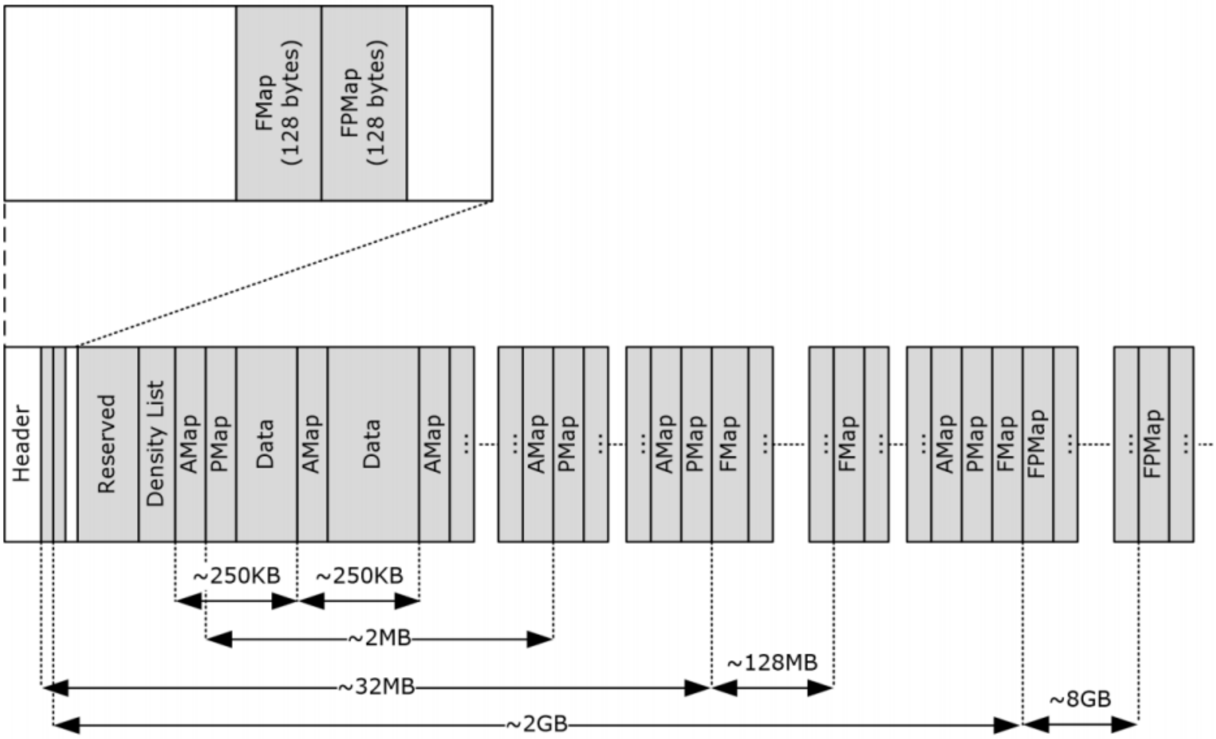

Figure 8.2: PST file format structure, source: [Mic20]

the presence of the allocation information (AMap, PMap, FMap, FPMap) that is present at specific intervals.

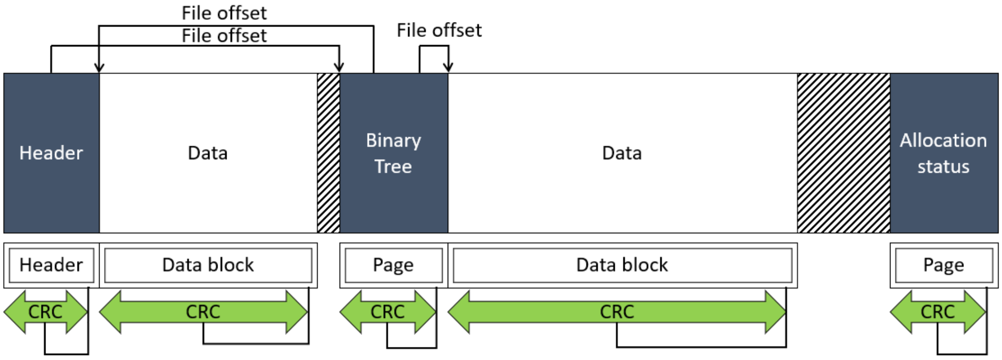

Figure 8.3: PST file format diagram

The PST file format only uses two types of data structures besides the PST file header: the page data structure and the block data structure. The page and block data structure are organized as follows:

Page data structure A page is a fixed size data structure of 512 bytes, which can represent several types of metadata. Pages are used to store allocation metadata and the binary tree data structures of the BBT and NBT, which are located in the NDB layer. A page trailer is located at the end of each page. In the page trailer is specified what type of page the data structure is. For instance, binary tree information or allocation metadata. The page trailer also contains an identifier of the page and a CRC checksum of the page contents. The page type determines what kind of information is stored in the page. The file format specifies the structure of the page for each type of page.

Block data structure A block consists of a variable size data structure between 64 and 8192 bytes. Blocks are the fundamental unit of data storage used in the NDB layer. Blocks are always multiples of 64 bytes and aligned on 64-byte boundaries. Blocks are used for storing two types of data: storing raw data and representing the sub nodes which can be contained within a node. The first part of a block structure contains the data that is contained in the block. At the end of each block, a block trailer is located. The block trailer contains information about the amount of data in bytes of the block, an identifier and a CRC checksum of the data stored in the block.

8.2. PST FILE FORMAT VALIDATOR FEASIBILITY

The method to analyze a file format specification with regard to the feasibility of file format validation is applied to the PST file format. First the file format specification is analyzed for the use of file format concepts. Table 7.1 is used to map these file format concepts to validation principles. The method in Chapter 7 specifies which validation principles are the minimum requirements to create a file format validator.

8.2.1. IDENTIFIED FILE FORMAT CONCEPTS

The previous section described the structure of the PST file format, by analyzing the PST file format specification [Mic20]. The following file format concepts are identified:

Sections The PST file format uses recognizable data structures: file header, page data structure and block data structure. The file header contains a file signature that can be used to identify the file header structure. The page and block data structures have a recognizable trailer structure at the end of a page and block data structure. This trailer can be used to identify a page or block data structure.

Hierarchy/Structure In order to access all the data that is stored in the PST file, the PST file format uses two binary trees: the NBT and BBT. A Binary tree is a hierarchical structure. Furthermore, the PST file format specifies that data structures regarding the allocation data are present at a specific location and interval. The interval of these data structures is also a property that can be seen as the hierarchy/structure file format concept. The PST file format thus contains two properties that make use of the hierarchy/structure file format concept.

References References are used in the PST file format to access the data that is stored in the different data blocks. The PST file format specifies the presence of two binary trees: the NBT and BBT. Both trees are used to get the data that belongs to a specific node.

The NBT contains references to all of the accessible nodes in a PST file. Each reference in a node is identified by a Node ID. Each node uses a data block structure that contains the references to the subnodes of the node, the block ID (BID) of the data block containing the references is stored in each node.

The BBT contains references to all of the data blocks that are used in a PST file. References to a data block contain the unique ID of the block and the absolute file offset of the data block.

Error detection All the data structures used in the PST file contain CRC values: Header, Block data structure and Page data structure.

The header data structure contains a field that contains the CRC value of the header, this CRC value covers the complete header except for the field containing the CRC value itself.

Both the page and block data structures have a trailer data structure located at the end of the data structures. This page and block trailer contain a field that contains the CRC value that covers the data of the page and block, the CRC value does not cover the data contained in the trailer.

Encoding The PST file format supports two different cipher algorithms to encode the data stored in data blocks. This feature is optional and can be enabled by the client that uses the PST file. The encoding is only used to obfuscate the data stored in the data blocks, the data block trailer is not encoded. Also other data structures, such as pages and the header are not encoded.

Metadata The header data structure of the PST file format contains the following metadata: encryption method (the encoding cyphers as mentioned in the encoding paragraph), file size and the locations (references) of the root of the NBT and BBT.

Furthermore, the allocation status of data sections in the PST files is stored in specific page data structures. Also the sizes of the data blocks and locations, absolute file offsets, are stored in page data structures.

Block trailers contain the length of the data stored in the block.

8.2.2. Minimum set of required file format concepts for validation

The PST file format uses the following file format concepts: sections, hierarchy/structure, references, error detection, encoding and metadata. In this chapter the file format concepts are mapped to the different validation principles using Table 7.1.

- **File signature:** The PST file format uses sections and has a recognizable file signature (magic string) that is located in the header data structure.
- **Recognizable data structures:** The PST file format contains several recognizable data structures:
  - File Header
  - Pages, which can be recognized due to the recognizable byte sequences that are present in page trailers
  - Data blocks, which can be recognized due to the recognizable byte sequences that are present in the block trailers.

  Only the following data structures are used in the PST file format, the file header, page and block data structures. The file header, page and block data structures make use of the sections file format concept. The PST file format uses the hierarchy/structure file format concept, due to the usage of the two binary trees (BBT and NBT) and the specific interval and locations of data structures containing file allocation information.

- **Ability to match data with the same file:** The sections file format concept is used in the PST file format, due to the use of the header, data block and page data structures. The file format can use encoding for the raw data of the data blocks, however this is not required. The PST file format also uses references in the form of absolute file offsets, these references can also be verified due to the use of recognizable data structures. The metadata file format concept is also used, the sizes of each section are specified are present in the data of the PST file. The hierarchy/structure file format concept is present in the form of two properties in the PST file format: the two binary trees and a specified interval and location of data structures containing allocation information.

The minimum set of required validation principles is covered by multiple file format concepts. This means that multiple validation techniques can be used to support the validation principles.

8.2.3. Additional file format concepts for validation

- **Ability to detect invalid/corrupt data:** The error detection file format concept is used in the form of CRC values that are present in the header, data block and page data structures. The CRC value can be used to validate the content of the data contained in the data structure, by computing the CRC value of the data present in the file under validation and comparing this value with the CRC value present in the data structure. The file format also specifies which values are valid for specific field, for instance the valid values allowed in the page type field of a page trailer data structure. The sections file format concept is also used, which enables the checking of the data structures of the sections.
- **Ability to check consistency across the complete file:** All file format concepts that can support the file consistency validation principle are present in the PST file format. As a result multiple validation techniques can be used to validate the consistency at file level. The file size (metadata) of the PST file is present in the header and can be checked. References can be checked across the complete PST file, both binary trees can be traversed to check for completeness and consistency. The interval of the allocation information (hierarchy/structure) can be validated across the complete file.

8.2.4. Challenging file format concepts for validators

The following categories increase the complexity to validate a file:

1. **Undefined length of sections:** The PST file format allows the presence of unused sections of data in a PST file. The length of the unused sections is undefined. The AMap structures indicate which parts of the data section are allocated, but in case of PST fragments, this AMap data structure might not always be available. Allocated data sections of the PST file always have a defined length (blocks and pages).
2. **Uninitialized/Undefined data:** PST files can allocate more storage space if required, when this happens the data that is present in the newly allocated area is undefined, the file format only specifies the initialization of the corresponding the AMaps and if required PMap, FMap and FPMap data structures.

3. **Non encoded data:** The PST file format allows encoding of data sections, however this is not required. Thus in theory the data sections of a PST file can be non encoded. This property can make it more complex to detect from which point the data in a data section is no longer valid, because decoding cannot be used to determine whether data still belongs together. This means that other techniques have to be used to find the corruption point.
4. **Unrecognizable data structures:** The PST file format has recognizable data structures (based on magic string and byte sequence matching) for the file header and page data structures. However the block trailers cannot directly be recognized, since these block trailers do not contain a fixed or limited selection of valid possible values, similar to page trailers. The contents of a block trailer can be derived once more information of the corresponding block is available.

8.2.5. PST FILE FORMAT VALIDATION FEASIBILITY

The minimum required set of concepts for file format validation are met. The file signature is present and the used data structures within the file are recognizable and contain a well defined structure. The ability to match data with the same file is present and several techniques can be used to accomplish this with file format validation. The header and page data structures are recognizable. The block data structures are also recognizable, but only if the contents of specific fields of the block trailer are known, if these are unknown it is not possible to recognize data block data structures.

The additional file format concepts are also present in the PST file format. It is possible to detect corrupt and invalid data in PST files. It is also possible to verify the consistency across the complete PST file by using file format validation.

However, the file format also contains properties that make it more difficult and complex to perform file format validation. These properties are present, because the PST file format has a multiple write life cycle. As a result of this, a PST file can contain data sections that are no longer in use, because data sections can be freed for reuse. Another aspect of the multiple write life cycle is that this property seems to have impact on the fragmentation of PST files. Van der Meer et al. [vdMJvdB20] found that the fragmentation rate of PST files compared to other file formats, is relative high (35.8%). A potential relation between fragmentation and the multiple write life cycle makes sense, because each time data is written to a file and a file increases this could lead to fragmentation. Since, if the file size increases of a PST file and there is no adjacent data cluster available on the storage format, this leads to a fragmented file.

Furthermore, the file format provides allocation information about which sections are freed and which sections should contain valid data structures. The PST file format validator can use the allocation information to rule out certain areas of the data, this strategy addresses the presence of challenging file format concepts to some extent.

4096 bytes is a common cluster size of a file system. The maximum data structure size within a PST file is 8192 bytes and page data structures have a fixed size of 512 bytes. This means that in the worst case a page data structure can be stored in 2 different clusters and a block data structure at most in 3 different clusters. This limits the uncertainty of combining fragmented files together. This makes the reconstruction of a block or page data structure straightforward, because of the limited amount of clusters involved and the presence of a CRC value that can be used to verify the reconstruction. In theory a PST file can consist out of reserved (allocated) areas that do not contain data structures yet, in this case only the allocation information is present at intervals of 253,952 bytes. This property can be used as an upper boundary for the amount of clusters that needs to be checked for detecting the end of a PST file. The limited size of the data structures in a PST file with respect to the cluster size, the amount of involved clusters is restricted, this makes validation less complicated.

The creation of a PST file format validator that can provide detailed information of the location of corruption points and the recognition of PST file fragments is therefore feasible. The file format provides enough structure and concepts that can be used to verify the validity of a PST file. The resolution of corruption detection can vary between 512 or 8192 bytes in case of corruption within a data structure. The corruption resolution is close to the most common cluster size, 4096 bytes, on which fragmentation point can occur. If there is an area of non allocated data in between the upper limit is 253,952 bytes, which is quite big in comparison with the cluster size of 4096 bytes. However, this also means that this part does not contain any data, as a result the validator could use this property to reconstruct the PST file by adding these sections automatically without any data. Furthermore, the added complexity of the multiple write life cycle make the PST file format an interesting use case.

8.3. File validator for the PST format

A file validator indicates whether provided data is a valid PST file with respect to the PST file format. Furthermore, the file validator indicates from which point the data is valid and from which point the data no longer contains a valid PST file format structure.

Messages and attachments are stored across separate data blocks that are interpreted by combining the information from the NDB layer and LTP layer, which are stored across separate pages in the PST file. As a result of these properties, the data of the messages and attachments are scattered across the PST file. This property provides challenges for restoring messages or attachments directly from the PST file itself.

The physical organization of the PST file is an interesting part from a file validation perspective, because in terms of file validation, the file might be corrupt or incomplete and as a result of that higher level views might not be available, since the required lower level data is missing. When validating the file format, only the lower level data structures can be validated at first. The lower level data structures must be complete and valid, otherwise it is impossible to validate the higher level layers. Therefore, the validator at first can only validate at the level of the header element, pages and block structures.

From data structure perspective the PST file only has three type of data structures on which other abstractions are build:

- Header element
- Page element
- Block element

8.3.1. Design of a PST format file validator

Figure 8.4 gives an overview of the different steps that are involved during the validation of a PST file. This figure also shows the relation between the identified data sections of the PST file and the different steps involved to accomplish this. The validator consists of the following steps:

- Step 1: Header and page recognition: used to identify and validate header and page data structures.
- Step 2: Reference extraction: used to find references, which are required to find block data structures
- Step 3: Block recognition: used to identify and validate block data structures
- Step 4: File consistency validation: validate the PST file based on the identified pages and blocks.

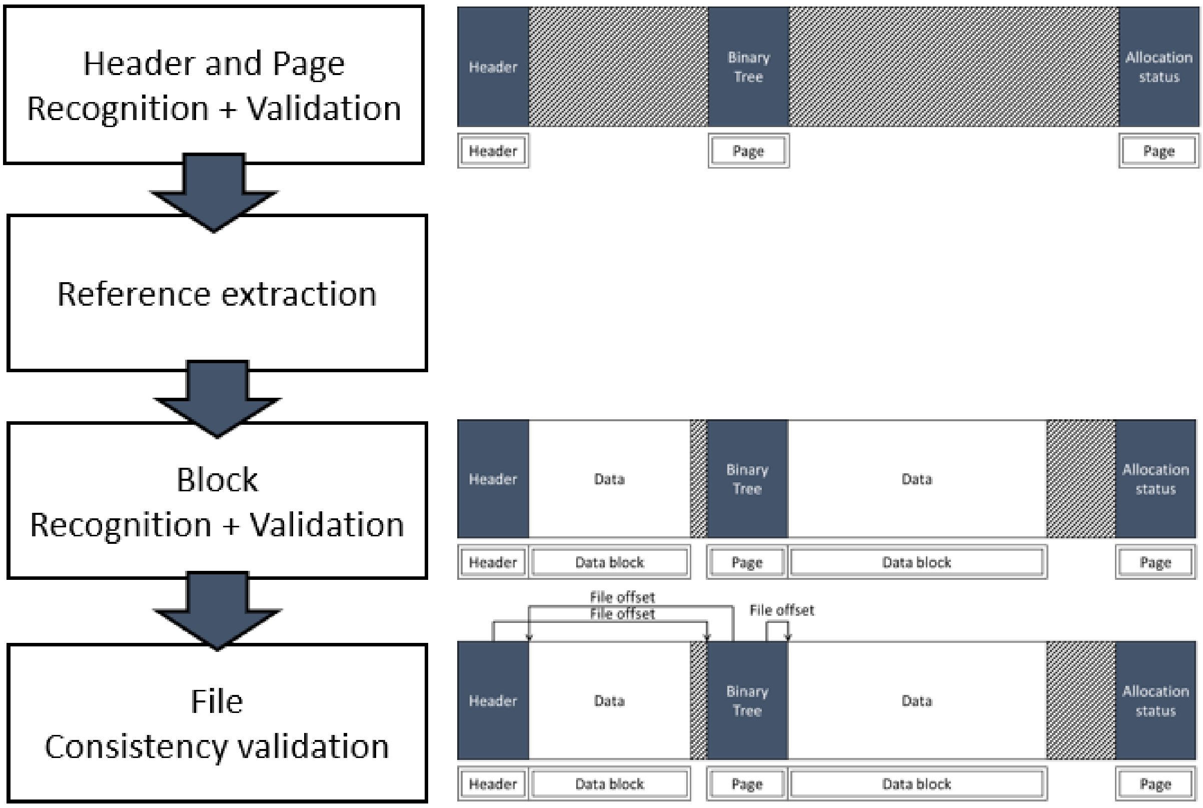

Figure 8.4: PST validation architecture

Step 1: Header validation The header element is the first data structure of each PST file. Figure 8.5 contains an overview of how a header structure is validated. The data is handled cluster by cluster, in each cluster the presence of a header is checked. A PST file header contains a recognizable byte sequence, a magic string, which is used to recognize the PST file format. The PST file header can be validated by calculating the CRC value of the data contained in the header and comparing this value with the stored CRC value in the header. The file header also contains metadata of the PST file: File size, references to the root of the BBT and NBT and the absolute file offset of the last AMap, which contains allocation information. The following validation techniques are used for recognizing the file header: Magic number matching and internal verification checking.

Magic number matching is used to identify the file header and file format. Container structure validation is used to retrieve the CRC value of the header element. Internal verification checking is used to validate the contents of the header. If the content of the header is valid, the metadata that is stored in the header element, such as file size, can be used by the validator.

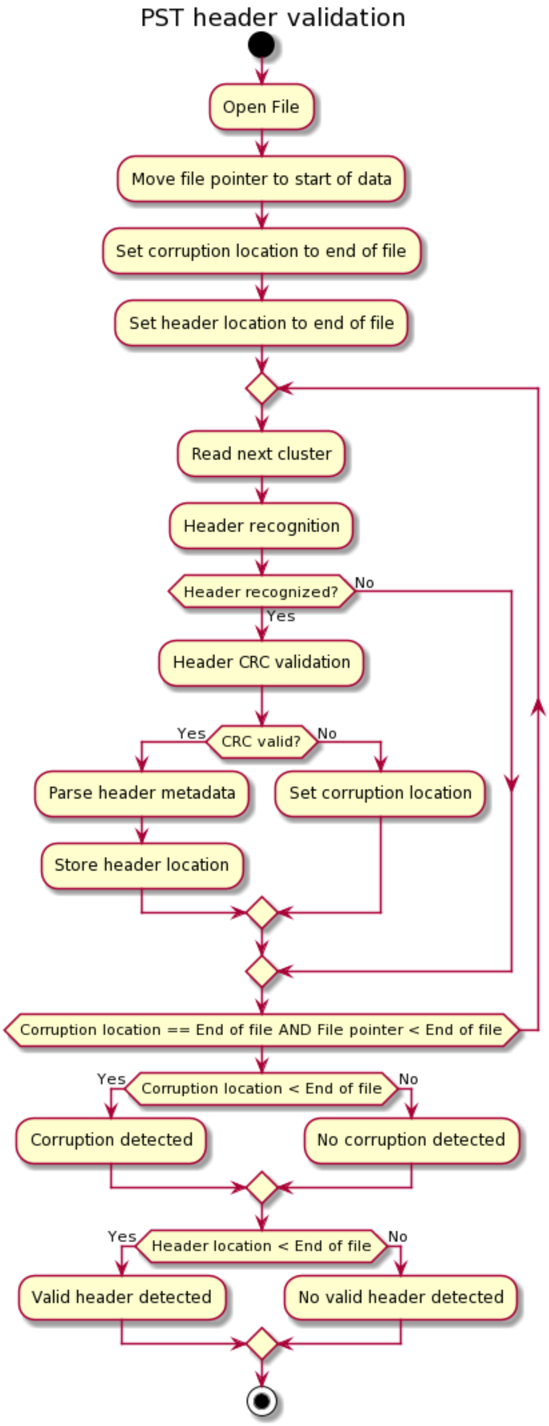

Figure 8.5: PST Header validation

Step 1: Page validation As mentioned in the previous chapter, a page element consists of the page itself and the page trailer. Figure 8.6 contains an overview of the different steps involved during page validation. The validator uses the page trailer to recognize and validate a page. A page trailer contains the following fields: ptype, ptyperepeat, wSig, dwCRC and bid.

The page trailer contains information about which type of page it is, the pagetype is stored in the ptype and ptyperepeat field. The page types are specified by the file format, there are 7 types of pages, so there are only 7 valid values possible for the fields related to the page type in the page trailer. These 7 values are used as signatures (magic strings) for recognizing page trailers. Thus the magic number matching technique is used to find page trailers.

Once these page trailers are found, the CRC checksum that is present in the page trailer is used to validate the content of the page data (internal verification checking technique). The size of the page is fixed and the trailer has a fixed size, as a result it is clear which part contains the data of the page. The value of the bid field depends on the page type. In case the page type is an AMap, PMap, FMap or FPMap the bid field contains the absolute file offset, otherwise the field contain the unique identifier of page (bid), these values are verified (container structure validation).

The wSig field of the page trailer contains the page signature of the page. This value is always 0 in case the page type is AMap, PMap, FMap or FPMap. For the remaining page type the page signature can be calculated by using the absolute file offset and bid value. The correct value of the wSig field is determined and verified (container structure validation).

As a result, all the values of a page trailer can be validated to check whether a page trailer structure is valid. Also the page data can be validated by using the CRC value stored in the page trailer. These two methods combined allow the validator the validate page data structures.

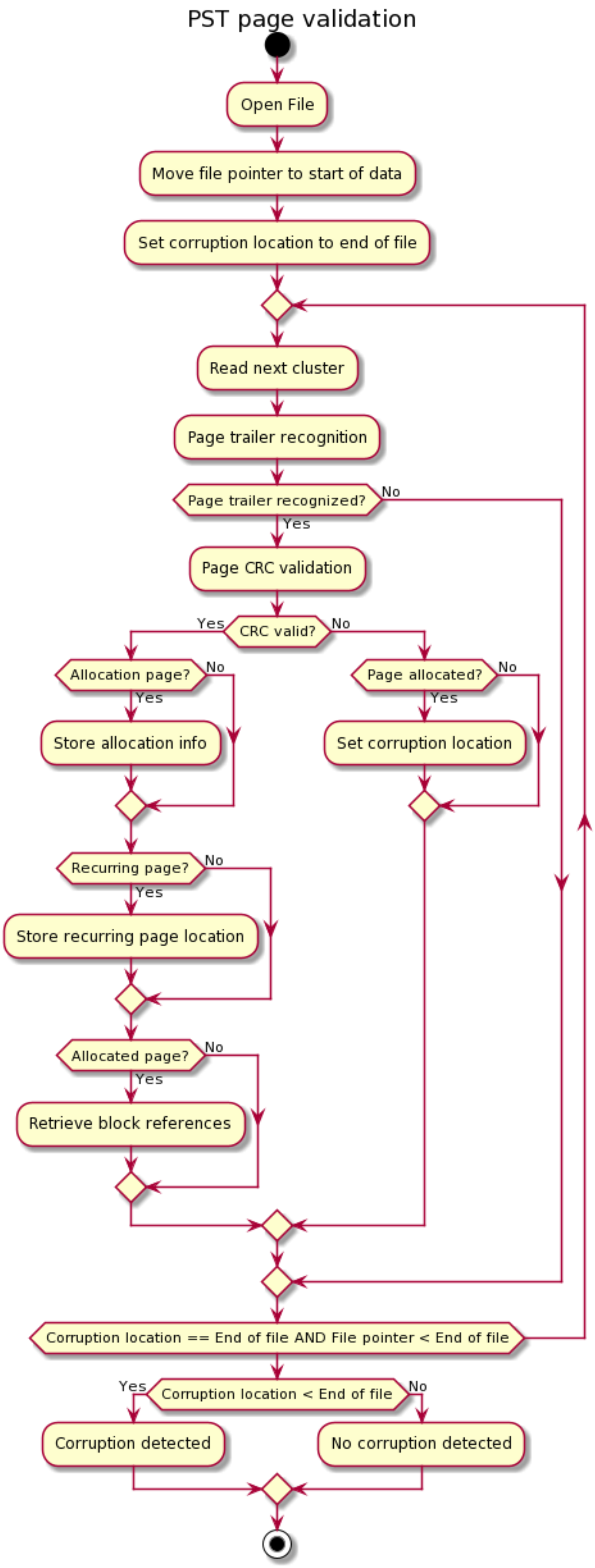

Figure 8.6: PST page validation

Step 2: Reference extraction References are used to recognize block trailers. Block references are stored in BBTENRTY records. A BBTENTRY contains a BREF record, the data size of the block and a reference counter. BBTENTRY records are stored in block binary tree (BBT) pages. These pages can be recognized by a specific ptype (ptypeBBT) in the page trailer. Only leaf BBT pages contain block references, these can be recognized by the value of the clevel field of a BBT page structure, which has to be 0.

A BREF record contains the mapping of a BID to the corresponding absolute file offset (IB) in the PST file. The BID and IB values are used for finding block trailers.

A BBTENTRY thus contains the bid, absolute file offset and data size of the block. The bid and absolute file offset can be used to calculate the wSig field of a block trailer. This means that the cb, bid and wSig values of a block trailer are known, these values can be used to recognize the corresponding block trailer (magic number matching and container structure validation).

When the validator finds a valid leaf BBT page, the block references are retrieved. These references are used by the validator to search for valid block trailers and thus finding data blocks in the PST file. Figure 8.6 contains the reference retrieval step that is executed during page validation.

Step 3: Block validation Blocks have a variable size between 64 and 8192 bytes and are always a multiple of 64 bytes. Like pages, blocks also have a trailer located at the end of the block which is called the block trailer. The block trailer contains metadata of the block, this metadata can be used by the validator to validate the block.

The block trailer contains the following fields: cb, wSig, dwCRC and bid. The cb field is the length of the data contained in block data structure. The wSig field is calculated by using the absolute file offset and bid value of the block. The dwCRC field contains the CRC value of the data of the block. The bid field contains the unique block ID of the block, the bid field contains the value that is used for calculating the wSig value.

Please note that block trailers do not contain a fixed signature, like the different page types in the page trailer. This means that block trailers cannot directly be recognized by a validator, since the block structure does not contain recognizable byte sequences, like the page trailer does. Blocks are thus harder to recognize and validate by the validator.

In order to find a block trailer, the values of the fields in a block trailer must be known. The PST file format uses references. References contain information from which the values of a block trailer can be determined. A reference contains the block identifier (BID) and the absolute file offset (IB) of a block. Once the absolute file offset and the bid are known, the value of the wSig field in the block trailer can be calculated by using the block identifier and absolute file offset. Also the bid field of the block trailer is known, since the block identifier is stored in a reference. The data size of the block data is also stored in a reference. As a result, all of the fields of a block trailer can be found or derived from a reference. The values of these fields are used to recognize a block trailer (magic number matching), the layout of the block trailer is used (container structure validation) to determine what the value of block trailer must be. Figure 8.7 contains the different the steps involved during the validation of blocks.

Once a block trailer is recognized by a validator, the CRC value of the block trailer is used to validate the data that is stored in the block (internal verification checking). If the block trailer is recognized, the block trailer is valid. If the CRC check passes, the data in the block is valid. As a result, blocks can be validated by using these two checks.

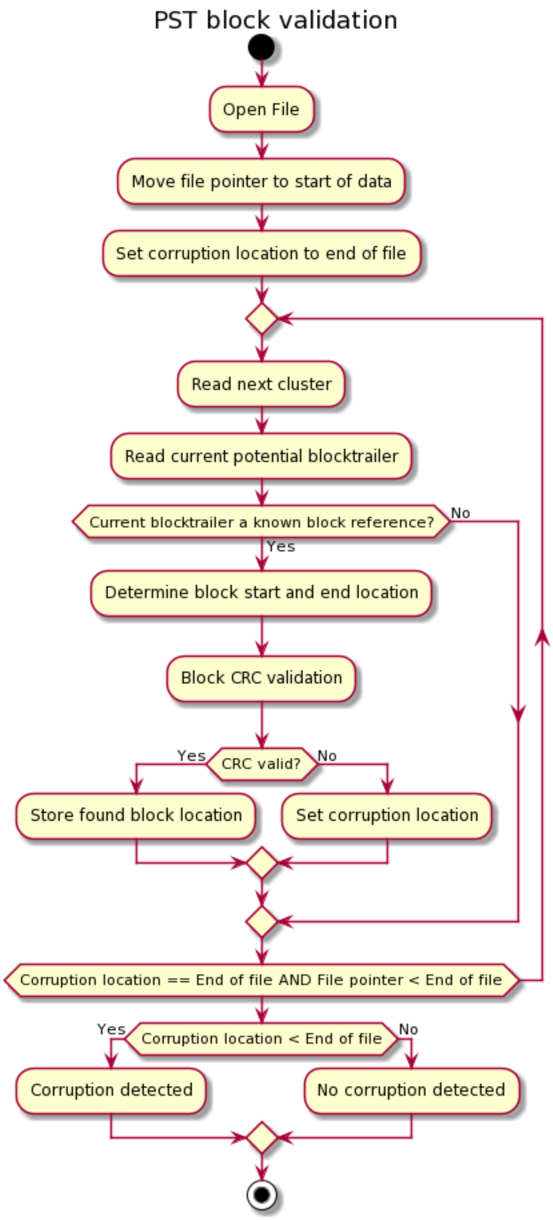
PST block validation
Open File
Move file pointer to start of data
Set corruption location to end of file
Read next cluster
Read current potential blocktrailer
Current blocktrailer a known block reference?
No
Yes
Determine block start and end location
Block CRC validation
Yes
CRC valid?
No
Store found block location
Set corruption location
Corruption location == End of file AND File pointer < End of file
Yes
Corruption location < End of file
No
Corruption detected
No corruption detected

Figure 8.7: PST Block validation

Step 4: File integrity The validator can check the integrity of the PST file by verifying various aspects for consistency:

- Specific data structures at specified intervals in the PST file, mandated by the file format
- Consistency of data structures by using CRC checksums in the header, page and block data structures.
- Verifying references: presence of a valid block structure at the location of the reference.
- Verifying absolute file offsets with respect to the start of the PST file.

The PST format mandates specific pages at specified intervals, this must be checked by the validator. The PST format mandates that a PST file starts with a header and expects the occurrence of four different data structures containing allocation information (AMap, PMap, FMap and the FPMAP page) at a specific interval. Depending on the size of the PST file, the file format expects a certain amount of these structures, please refer to the grammar of the PST file format to see the intervals of the data structures. If these page data structures not occur at the intervals specified by the file format, the PST file is not valid. This information can be used to give an indication from which part the PST data is no longer valid (corruption point). However the accuracy using this method is limited, since the data structure with the smallest interval has an interval of 253,952 bytes (AMap).

Between the data structures that are mandated by the file format, the file format specifies that the remaining available space is used as data section. Data sections can contain undefined data, when a part of the section is not allocated or contains page or block data structures. The allocation map (AMap) page structure indicates which parts of the data section are allocated. The information in the AMap can only be trusted if the AMapValid field, located in the root structure of the PST header, indicates that all the AMaps in the PST file are valid. Allocations in the data section are always multiples of 64 bytes, as a result each bit in the AMap represents 64 bytes and every allocation is aligned to a 64-byte boundary. The allocated parts of the data section are validated whether they contain valid page or block structures. If a specific part of the data section does not contain valid data and the AMap indicates the section is allocated, this is an indication that the data is no longer valid from that point onward.

The PST file contains data structures (BBTENTRY) that contain references to blocks. These references also contain an absolute file offset, that is relative to the beginning of the PST file. These references must be checked to verify the correct page or block is present at the specified file offset. This can be accomplished because each page and block has a unique identifier (BID) that can be found in the trailer of a page or block structure and this BID is also located in the reference. The validator checks every retrieved reference, by verifying whether a valid block data is present at the location specified by the reference. If there is no block data or corrupt block data present, a corruption point is identified.

8.3.2. Functionality and limitations

The file validator is able to recognize and validate file fragments of the PST file format. However, there are some limitations and properties of the PST file format which have to be taken into account.

Allocated data structures As mentioned in Section 8.1 the PST file format allows the reuse of sections. This means that a PST file can contain freed pages and blocks. The PST file format keeps track of which sections of a PST file are allocated and which sections are freed. It is possible that a freed page or block is partly overwritten with new data, as a result a valid page trailer or block trailer can be left, but the data part can be corrupt and the CRC check will fail. The validator has to take allocation into account in order to prevent false detection of invalid data structures. When pages and blocks are validated, they must be located in an allocated part of the PST file. The validator must allow invalid data structures in freed sections. Therefore, the validator can only report corruption in allocated sections of a PST file.

As a result allocation information has to be available for the validator to function correctly. The allocation information is retrieved during page validation, as can be seen in Figure 8.6. Allocation information is stored in AMap pages, these pages are stored in intervals of 253,952 bytes. The validator can recognize corruption in PST file fragments correctly when the fragment starts with a PST header or an AMap page. Without allocation information, the validator can only recognize valid page and block data structures, but the validator cannot detect invalid data structures, since this requires allocation information. This is a limitation of the validator.

Fragment recognition Because it is possible to recognize page data structures, the file validator is also able to recognize file fragments of a PST file. This means that the provided data does not have to contain a file header in order to recognize the PST file format. Data blocks cannot be directly recognized, since the recognition of data blocks requires additional information (retrieved from references) that has to be extracted from recognized pages.

However, a PST file is allowed to contain invalid pages and blocks, since sections can be reused and overwritten in a PST file. In order to make sure that a corrupt data structure is correctly identified as file corruption, the allocation information of the PST file must be available.

Fragment recognition is possible by identifying pages and blocks without allocation information. However, if the allocation information is not available, the validator cannot identify corruption correctly. For fragment recognition this might be sufficient, because we are only interested in recognition. In order to combine fragments and actually validate the data, allocation information is required, in order to make sure that corrupt data is detected in an allocated part of the file.

File format validity vs file validity Data provided to the file validator can contain valid PST file format data structures. However, this does not mean that the detected PST file format contains a valid PST file that successfully can be opened by an application which supports the PST file format.

The data within the PST file format can also be invalid. The file validator is limited in checking the consistency of the PST file with regard to the file format. As mentioned before, the PST file contains two binary trees, these trees can be checked for consistency. The data of the binary trees is stored in pages that are spread across the complete PST file, this is especially problematic when a fragment of a PST file is detected, since this means that not all the data of the binary trees is present, which makes the validation of the binary trees impossible. This means that PST file fragments are only validated on data structure level. The integrity of the complete NDB layer, the lowest logical level in the PST file format, can only be validated if the complete binary trees are available.

Even if the complete PST file is provided, only the lowest logical level (the NDB layer), is validated. The logical layers on top of the NDB layer are not validated.

In the context of file carving this limitation does not have to be an issue. The goal of file carving is recovering deleted files from a storage medium. If corrupt PST files are deleted, they are also corrupt if they are recovered.

8.4. Proof-of-concept implementation

The previous sections described how the header, page and block elements of a PST file can be recognized and validated. Also the validation of consistency at file level is discussed. All these techniques combined form the basis of the PST validator implementation. In order to verify the correctness of the validation strategies and concepts that are introduced in Chapter 8.3.1 a proof-of-concept version of the validator is implemented. Figure 8.4 contains an overview of the different steps that are performed in the PST validator.

Figure 8.8 gives an overview of the different steps that are involved in validating a PST file. The validator has to go through the data of the PST file twice.

The first time the validator can only find valid pages, since the references for finding blocks first have to be retrieved from recognized pages. Figure 8.6 provides an overview of the steps involved during page validation and please refer to Section 8.3.1 step 1 for more information regarding page validation. Please note that references may only be retrieved from valid and allocated pages, because the references might not be valid in case they are retrieved from non allocated pages.

Once all references for the blocks are retrieved, the second run can search for valid blocks, by searching for the block trailers using the information from the references that is retrieved from valid and allocated pages. Figure 8.7 provides an overview of the steps involved during block validation and please refer to Section 8.3.1 step 3 for more information regarding block validation. The recognized blocks do not contain references or other metadata that can be used by the validator, therefore it is not required to run the validator once more through all the data of the PST file, since this does not result in finding more blocks or pages of the PST file.

Once all the data structures of the PST file have been found and validated, file consistency validation can be performed. The validator uses several rules to find corruption. Once the first corruption point is detected, the validator looks no further than the first detected corruption point, however the validator still tries to find corruption before the first detected corruption point, since this might give a more accurate indication from where the corruption occurs.

The following rules are used for corruption detection:

- CRC check fail of allocated pages (Page validation).
- CRC check fail for identified blocks (Block validation).
- CRC check fail for an identified file header (Header validation)
- All found block references must refer to a valid block at the specified location in the PST file (Integrity check).
- Recurring structures mandated by the file format must be present at the specified locations in the PST file (Integrity check)

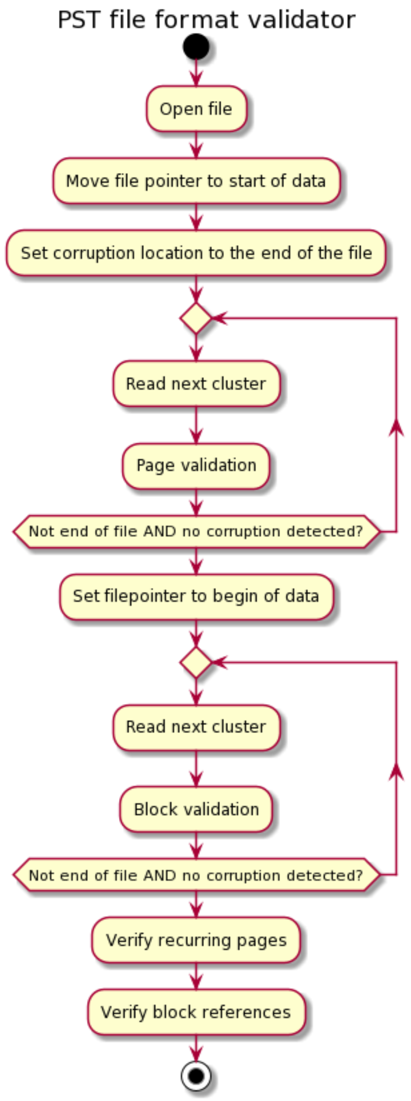
PST file format validator
Open file
Move file pointer to start of data
Set corruption location to the end of the file
Read next cluster
Page validation
Not end of file AND no corruption detected?
Set filepointer to begin of data
Read next cluster
Block validation
Not end of file AND no corruption detected?
Verify recurring pages
Verify block references

Figure 8.8: PST validator implementation

8.4.1. Verification

In order to verify that the implemented proof-of-concept PST validator is functioning as expected, the validator is tested. The goal is to verify that the strategies for validating the header, page and block data structure function correctly. In case a data structure is corrupt, the validator should be able to detect this.

Since the goal of the testing is limited to the basic functionality to verify the correctness of the validation strategies, stress testing and extensive performance tests are out of scope.

Tests were performed on a selection of PST files that are created using Outlook 2019, version 16.0. The PST Data structure viewer which is part of the PSTSDK from Microsoft[1] is able to display the complete structure of the PST file, this includes a view of both binary trees. This allows us to verify whether the proof-of-concept PST validator is able to detect

[1] https://archive.codeplex.com/?p=pstsdk

all the present page and block data structures, this check was done on a PST file of around 2 megabytes that contains 1 email message which contains a PDF attachment. The detected page and block structures are compared based on the reported file offset and unique identifiers of the pages and blocks. Also the references contained in the pages retrieved by the validator are compared with the references displayed by the PST Data structure viewer. Furthermore, corruption was added by manually altering a few bytes in the PST file at specific locations to test the header, page and block validation. The validator was able to detect the manually added corruptions in header, page and block data structures.

The PST file of 2 megabytes is not a representative open world example. In order to overcome this limitation we also tested with a more representative case. However, it is not straightforward to acquire a representative PST file due to the multiple write life cycle of a PST file. For instance, a PST file that has been in use for multiple years has undergone multiple write and delete actions, which result in the presence of artifacts of freed sections. It is a challenge to produce a PST file for testing which has undergone actions that are similar and representative to years of usage. Therefore, we decided to limit the testing to a manually created PST file that has undergone several file operations such a deletion and creation. This PST file contains contains 343 emails and is 64 megabytes in size. First the original valid file was validated by the validator, after that corruption was manually added in several random locations. The validator was able to locate the corrupted sections.

These tests show that the proof-of-concept validator can correctly detect corruption in the cases that were tested using a simple PST file and a more complex and representative PST file. The testing is sufficient to test the validity of the concepts that are introduced in the design of the header, page and block validation. However, we still recommend more extensive testing with open world examples in case the validator is going to be used in the field. We addressed this risk by testing the validator using a more complex file that has undergone multiple file operations, but this potentially still does not cover all possible scenarios of a open world PST file.

8.4.2. Results

The implemented PST file validator is a proof-of-concept, to verify that file format validation for the PST file format is possible. The implemented validator is able to detect corrupt pages and block data structures, furthermore the validator uses other properties of the file format such as the recurring data structures and the use of references to validate a file. As a result the validator is able to identify and validate fragments of a PST file. The validator can also be used to recognize valid combinations of different file fragments.

However, there is still room for improvement. Aspects of the validator that could be improved are the performance of the validator and file carver integration. We consider file carver integration and performance optimizations out of scope for a proof of concept validator, because our target was to demonstrate that file validation for the PST file format is possible and how this could be achieved. Furthermore, the implemented PST validator and introduced strategies can be used as a starting point for an implementation of a PST file validator. Therefore, we consider the current limitations with regard to performance and file carver integration acceptable.

Another conclusion is that the PST file format provides enough structure and file format concepts that enable the implementation of a validator that is able to verify the PST file format. Furthermore, the file format allows corruption detection at data structure level. The biggest possible data structure in a PST file is a block data structure which has a maximum size of 8192 bytes. A typical cluster size of a file system is 4096 bytes, thus the corruption detection is comparable in magnitude with a typical cluster size.

9

Conclusion, discussion and future work

9.1. Answers to the research questions

The research questions are answered in this section in order to answer the main question: to what extent can a file format specification guide file validation?

9.1.1. RQ1: Which existing file carving techniques can be used in a file format validator?

A literature research has been performed in Section 5.1 and the following list of existing file carving techniques exists and can be used in a file format validator:

Header and footer validation Searches for the header and footer of a file format and recovers the header, footer and the data in between.

Magic Number Matching Searches for recognizable bytes sequences of a specific file format in an arbitrary data stream.

Container structure validation Validates the internal structure of data structures as specified by the file format.

Data Dependency Resolving Dependencies and relations present in a file format are checked, for instance: reference checking and validation of the field lengths.

Validating with decompression Compressed sections in a file format can be decompressed to check whether the section can successfully be decompressed.

Algorithm Output Analysis Tries to match data together by determining whether the data is encoded by the same algorithm using bit sequence matching.

Internal verification checking Uses the internal verification mechanisms of a file format to validate the contents of a file.

Semantic validation Uses semantic information in order to match data together, for instance the language of a document.

9.1.2. RQ2: What kind of concepts are used in file formats?

In Section 6.1 and Appendix A different file format specifications of commonly used file types were analyzed. This analysis provides insight on how file formats are organized. Section 6.2 summarizes the identified concepts in different file formats. The conclusion was that file formats do use similar concepts and the following concepts are identified:

Sections Data structures that belong together in a file, such as a frame in a MP3 file.

References Data that refers to other locations within the file, for instance absolute file offsets or unique identifiers.

Encoding Data that is encoded using a specific algorithm, for instance audio/video encoding or compression.

Hierarchy/Structure Data structures or sections in a file that occur in a specific sequence or hierarchy.

Error detection Internal verification mechanisms that enable error detection in the data of a file, for instance CRC values.

Metadata Data stored in a file that contains information regarding the file itself, for instance file size, section size or the amount of sections. Please note that the type of information stored in this concept is very broad and can be very file specific.

9.1.3. RQ3: What concepts are necessary for a file format validator?

A file validator must be able to identify and validate fragments of a file of a specific file format. Validation principles are introduced in Section 7.2, these principles are used to support the identification and validation of fragments. The feasibility of a validator is determined by verifying to what extent the validator supports the different validation principles. Section 7.4 describes which validation principles are necessary to perform file validation.

In order to answer RQ3 a mapping is created between the validation principles and the validation techniques, which is discussed in Section 7.3.1. Since validation techniques depend on the availability of file format concepts, the relation between a file format specification and the feasibility of a validator is identified.

Table 7.1 provides the complete overview of the relation between file format concepts and the validation principles. This table combined with the knowledge of what validation principles are required for a validator as mentioned in Section 7.4, provide the answer to what concepts of a file format specification are necessary for a validator.

9.1.4. RQ4: How can this be used to design a file format validator for a complex format?

Chapter 8 describes the application of the introduced method on the PST file format. The PST file format specification is analysed to identify the file formats concepts that are present in the PST file format. Table 7.1 is used to identify which validation techniques could be used with the identified file format concepts in the PST file format. The conclusion of applying the method was that the creation of a validator for the PST file format is feasible.

The next step is to verify the conclusion of the method that the creation of validator for the PST file format is feasible. In order to verify this conclusion a PST validator is designed and implemented based on the validation techniques identified by the mapping described in Table 7.1. Section 8.3 describes the design and implementation of the PST validator. The implemented PST validator is able to identify and validate file fragments of the PST file format. These two properties are required for a validator. The proposed method for determining file format validation feasibility thus also works on a complex file format that has a multiple write life cycle.

The specification of the PST file format shows it provides for detailed structure. Moreover, the components used in the specification closely match those found in the analysis of file format specifications (Chapter 6). Thus, the PST file format specification is exquisitely suited to use as a litmus test for our approach. The resulting proof-of-concept implementation of a PST validator illustrate the feasibility of the approach. In more detail, the PST validator is able to correctly determine the section in which the corruption point occurs in a closed test.

9.2. Conclusions

The answer to the main question is based on the answers and conclusions that are drawn from the research questions.

Main question: To what extent can a file format specification guide file validation? A file format specification can be used to directly implement a file validator in an ad-hoc fashion, which is based on the properties of the file format. However, a more systematic approach is possible by applying the method we introduced in Chapter 7. The method gives insight on the feasibility of validation for a given file format specification, furthermore it also provides guidance on which existing file validation techniques are possible to validate the file format.

The introduced method can be used to guide the implementation of a file validator based on the file format specification. However, this also means that the extent of guidance depends on the file format specification. If the file format specification does not use file format concepts that can be used by a corresponding validation technique, the method cannot identify which validation techniques to apply in a validator. In this case the outcome of the method is that file validation is not feasible for that specific file format.

Thus, the extent to which a file format specification can guide validation depends on whether file validation is feasible for the given file format specification. If file format validation is possible for a given file format, the proposed method allows guidance in the design of a validator. The extent to which guidance can be provided by the file format specification depends on the amount of used file format concepts. If multiple file formats concepts are used in a file format specification this allows the usage of multiple validation techniques in the design of a file validator. The extent of file validation guidance thus depends on the amount of identified file format concepts in a file format specification.

9.3. Discussion

The method introduced in Chapter 7 opens up new doors for the field, the following cases can now potentially be handled:

The added value of the method introduced in Chapter 7 is that it allows re-usage of existing knowledge, by quickly identifying possible validation techniques, which improves the efficiency of developing validators for new file formats. Furthermore, a file format specification can be analyzed to determine whether file validation is possible at all, this allows identification of file formats that have an insufficient amount of concepts and structure.

Another view is that the identified file format concepts can be used to create a file format that allows easier recognition, validation and reconstruction of fragmented files. In particular, we advise to incorporate at least the following characteristics into a file format to make future file formats more recoverable: The use of recognizable sections which are smaller than the typical cluster size of a file system that contain an internal verification mechanism such as CRC validation. With regard to the reconstruction of fragmented files we recommend the use of unique identifiers which identify the file itself and the sequence of the sections within a file.

Finally, we see an application of our findings for anti-forensics by deliberately making file recovery hard. Since our method provides an overview of possible validation techniques and the corresponding required file format concepts. This allows the construction of a file format which lacks file format concepts that allow recognition, validation and reconstruction.

Related work of Roussev and Garfinkel [RG09] concluded that a specialized approach is required for recognizing a specific file type in order to correctly distinguish different file formats. Our method can be used for developing specialized validators for specific file formats, based on the present file format concepts.

Furthermore, there are already several successful implementations of specialized file format validators that can be used to reconstruct fragmented files, such as the ZIP and PDF file format [Coh07], RAR files [WZX10] and AVI files [YXLS17]. Our method can be used to identify new file formats that potentially also allow reconstruction of fragmented files, based on the analysis of file format specifications. This is interesting because currently one of the challenges for file carving is the recovery of fragmented files. Van der Meer et al. [vdMJvdB20] investigated which file formats are typically fragmented on a system. The identified file formats in this list are a good starting point to apply our method on to verify whether validators could be designed that allow reconstruction of fragmented files of other file formats.

The conclusion of the PST file format investigation we performed was that it is possible to create a file format validator for the PST file format. Furthermore, we came up with a design and implementation for a PST file validator. Our findings were that this validator is able to recognize and validate PST file fragments. In related work there are currently no other validators that are able to recognize and validate file fragments of the PST file format. Further integration of this validator with a file carver is recommended in order to investigate whether it is possible to recover fragmented PST files. This could be a novelty, since current available tooling does not seem to be able to recover fragmented PST files. The impact of recovering fragmented PST files would be beneficial for digital forensics, since from a digital forensics point of view you want to gather as much information as possible.

9.4. Limitations and future work

This thesis introduced a method to determine file format validation feasibility. This method is based on currently known validation techniques and file format concepts that were identified as a result of a file format investigation we performed in this thesis. The introduced method to determine file format validation feasibility provides a starting point, but further refinement is still possible. The current selection of file formats analyzed in the file format investigation performed in this study does not guarantee completeness of the identified file format concepts. In case more file formats are investigated there could be more concepts discovered or this provides more evidence that the current list is complete. The same applies to validation techniques that are currently used. If there are more file format concepts and validation techniques identified, these must be added in the mapping that was created in Table 7.1.

Another step would be to investigate whether it is possible to automate the process of file format specification analysis and application of the method to determine file format validation feasibility, because the current method involves a lot a of manual steps. The proposed method also provides guidance on how to implement a file validator, thus the added value of automation could be that file format validators are automatically generated based on the provided file format specification.

The integration between a file carver and a file format validator has not been investigated in this study. In order for a file carver to fully benefit of the added value of file format validation, the file carver needs to use information retrieved from the file format validator. The file format validator could have more information available regarding the fragments that limits the search space during the reconstruction of file fragments. For instance, the validator could have information regarding the unique identifiers that are present or missing in a file fragment. In this approach a validator provides more information to the file carver, instead of only returning whether a file is valid or not.

The application of the proposed method to determine file format validation feasibility of the PST file format showed that the PST file format provides a lot of structure and file format concepts. The proof-of-concept PST validator already shows promising results in stand-alone tests, however further testing of the validator with open world PST files is recommended. Currently the validator is able to recognize and validate fragments of PST files. The next step would be to integrate the PST validator with a file carver to create a file carver that is able to recover fragmented PST files. In related work we did not find implementations to recover fragmented PST files. If academic publications are a good representation of the current knowledge available in digital forensics, than the proof-of-concept PST validator definitely has an added value for digital forensics for the recovery of fragmented PST files.

Another aspect that is not covered in this study is sparse file support in validators. The administration of sparse files is located in the administration of NTFS, thus this information is not always available during file carving. However, the PST file format contains metadata about the allocations in a file format. This information could be used to allow the presence of sparse file data in a file in non allocated sections. Sparse data is not physically stored on a storage medium, the validator could take this property into account during validation. The presence of sparse data cannot be recognized by a validator, because a section containing sparse data is not physically stored. The validator should take this into account during the validation of references, since offsets are affected in case sparse data is present. The validator should restore the sparse data, by placing zeroes in the area of the sparse data, this addresses the incorrect offset problem. Sparse data that occurs in an allocated section is more difficult to recognize. However, the PST file format has a CRC value and the length of an allocated section available. This might provide enough information to implement sparse data support for the PST validator, further research is required to verify this.

BIBLIOGRAPHY

[Coh07] Michael I Cohen. Advanced carving techniques. *Digital Investigation*, 4(3-4):119–128, 2007. 8, 9, 10, 15, 16, 39, 70

[CZX$^+$08] Mo Chen, Ning Zheng, Ming Xu, Yongjian Lou, and Xia Wang. Validation algorithms based on content characters and internal structure: The pdf file carving method. In *2008 International Symposium on Information Science and Engineering*, volume 1, pages 168–172. IEEE, 2008. 9, 16

[Gar07] Simson L Garfinkel. Carving contiguous and fragmented files with fast object validation. *digital investigation*, 4:2–12, 2007. 7, 8, 9, 10, 13, 14, 15, 16, 28

[Lin18] Xiaodong Lin. Deleted file recovery in ntfs. In *Introductory Computer Forensics*. Springer, 2018. 4, 5, 6, 14

[Mic20] Microsoft. *[MS-PST]: Outlook Personal Folders (.pst) File Format*, 2020. 46, 47, 49, 50, xiii

[PM09] Anandabrata Pal and Nasir Memon. The evolution of file carving. *IEEE signal processing magazine*, 26(2):59–71, 2009. 7, 8, 9, 10, 28

[PT13] R. Poisel and S. Tjoa. A comprehensive literature review of file carving. In *2013 International Conference on Availability, Reliability and Security*, pages 475–484, 2013. 9, 15

[RF04] Richard Russon and Yuval Fledel. NTFS documentation, 2004. 4

[RG09] Vassil Roussev and Simson L Garfinkel. File fragment classification-the case for specialized approaches. In *2009 Fourth international IEEE workshop on systematic approaches to digital forensic engineering*, pages 3–14. IEEE, 2009. 10, 70

[RIR05] Golden G Richard III and Vassil Roussev. Scalpel: A frugal, high performance file carver. In *DFRWS*. Citeseer, 2005. 8

[vdB$^+$14] Jeroen van den Bos et al. *Gathering evidence: Model-driven software engineering in automated digital forensics*. PhD thesis, Universiteit van Amsterdam [Host], 2014. 7, 14

[vdMJvdB20] Vincent van der Meer, Hugo Jonker, and Jeroen van den Bos. A contemporary investigation of NTFS file fragmentation. *Digital Investigations*, 2020. 4, 5, 7, 10, 18, 46, 53, 70

[Vee07] Cor J Veenman. Statistical disk cluster classification for file carving. In *Third international symposium on information assurance and security*, pages 393–398. IEEE, 2007. 9

[WZX10] Yingjie Wei, Ning Zheng, and Ming Xu. An automatic carving method for rar file based on content and structure. In *2010 Second International Conference on Information Technology and Computer Science*, pages 68–72. IEEE, 2010. 9, 70

[YXLS17] Yitao Yang, Zheng Xu, Liying Liu, and Guozi Sun. A security carving approach for avi video based on frame size and index. *Multimedia Tools and Applications*, 76(3):3293–3312, 2017. 9, 70, vi, vii

[YYP^{+}12] Byeongyeong Yoo, Byeongyeong Yoo, Jungheum Park, Jungheum Park, Sungsu Lim, Sungsu Lim, Jewan Bang, Jewan Bang, Sangjin Lee, and Sangjin Lee. A study on multimedia file carving method. *Multimedia Tools and Applications*, 61(1):243–261, 2012. v

A

FILE FORMAT INVESTIGATION

This appendix contains the remaining investigated file formats that are mentioned in Chapter 6. This appendix uses the same conventions for the schematic diagrams as introduced in Section 6.1.

A.1. IMAGE

A.1.1. JPG

The JPG file format[1] specifies that a JPG file consists of different parts. Parts within a JPG file are identified by the use of markers. Markers are specific byte sequences of two bytes. Each JPG file starts with a start-of-image (SOI) marker and ends with a end-of-image (EOI) marker at the end of a file. The file format consists of different parts, which are segments in the file data. Segments are identified by a marker at the start of a segment. The structure of each type of segment is defined in the file format.

Figure A.1 provides a schematic overview of the file format. The file format dictates a specific structure and a strict ordering of segments. For instance, the file starts with a SOI marker and ends with a EOI marker with a frame segment in between. The frame segment itself starts with a recognizable frame header and contains scan segments that start with a recognizable scan header. The frame header contains information about the data stored in the frame section, like the dimensions of the image and the used coding scheme. Each frame header also contains a unique component identifier. Compression/encoding is applied on specific segments that contain the image data of the file. JPG supports two types of encoding: Huffman coding and arithmetic coding. The size of the segments depend on how the file is stored, the file format does not specify a fixed segment size. Furthermore, the file format does not make use of CRC checksums.

The sections containing the encoded image data can be difficult to recognize, because these sections can be larger than the cluster size of a file system. The only property this section has, is that it is Huffman or arithmetic encoded. It can be a challenge to recognize from which point in an encoded section the data is no longer valid.

The JPG file format contains more markers than the ones mentioned in Figure A.1. Also the frame and scan sections are simplifications of the JPG specification. These items have

[1] https://www.w3.org/Graphics/JPEG/itu-t81.pdf

been omitted because they are not required to understand the properties and used concepts of the JPG file format. Furthermore, the majority of a JPG file will consist out of the encoded image data section.

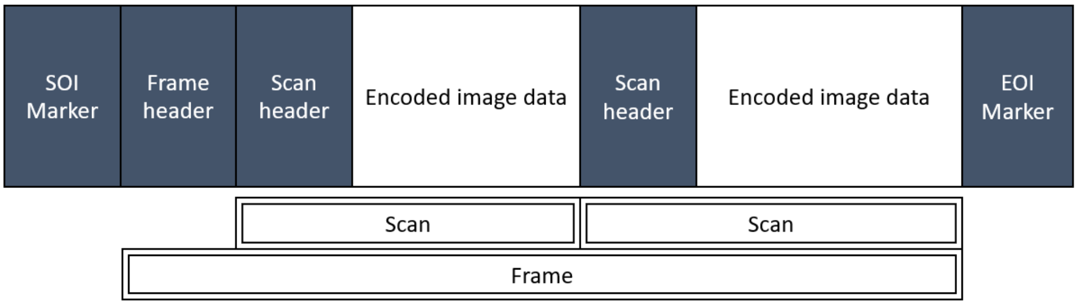

Figure A.1: JPG file format diagram

A.1.2. PNG

The PNG file format[2] consists of a recognizable PNG signature followed by a sequence of chunks. The recognizable PNG file signature is used to identify the bytes following as a PNG image. The data after the PNG file signature, consists of a series of chunks belonging to the PNG file. A chunk is a sequence of bytes that belong together. Furthermore, there are different types of chunks specified by the file format. Please note that the PNG signature is not a chunk, the PNG signature is a fixed value of 8 bytes to identify the start of a PNG image.

Figure A.2 provides a schematic overview of the PNG file format. As mentioned before, a PNG file always starts with a recognizable PNG signature followed by a sequence of chunks. Chunks are identifiable based on the specific byte sequences used for each different chunk type. A chunk contains fields, these fields contain at least the following information: the length of data field of the chunk (maximum length around 2 gigabytes), the type of the chunk, followed by the data contained in the chunk. A chunk contains a CRC value of the data contained in the chunk and the chunk type, the CRC value is stored at the end of each chunk. The layout and content of each type of chunk is specified by the file format. Some chunks are optional, but the file format dictates that the following chunks are required: the image header (IHDR) chunk, the image data (IDAT) chunk and the image trailer (IEND) chunk.

Some chunks have restrictions regarding the ordering and location, however there are also chunks that do not have restrictions regarding the ordering. Data in the chunks can be compressed/encoded using the deflate method and stored in the zlib format depending on the chunk type. For instance, the image data (IDAT) chunk contains compressed image data.

The IDAT chunk containing the image data can be larger than the cluster size of a file system. The beginning of the IDAT chunk is recognizable, due to a recognizable byte sequence to identify each chunk. However, when the IDAT chunk is larger than the cluster size of the file system, it can be a challenge to determine whether data still belongs to the same chunk. The CRC value can be used to verify the contents of the IDAT chunk, but in case this CRC check fails it is difficult to pinpoint from which point within the IDAT chunk the data is no longer valid.

[2] https://www.w3.org/TR/PNG/

The PNG file format contains more chunk types than the IHDR, IDAT and IEND chunk. Not every chunk type within the PNG file format is described. However, to understand how the PNG file format works the introduction of other chunks is not required. Furthermore, Figure A.2 describes a valid PNG file, since the IHDR, IDAT and IEND chunks are the minimum requirements of a valid PNG file. The majority of a PNG file will consist out of the image data, the IDAT chunk. The chance that corruption occurs within the IDAT chunk is therefore larger and probably is the biggest challenge when validating PNG files, this problem can be understood by using Figure A.2.

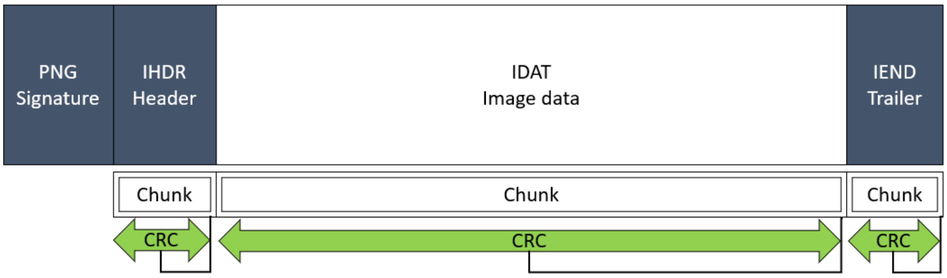

Figure A.2: PNG file format diagram

A.2. Audio

A.2.1. MP3

[YYP+12] The MP3 file format[3] uses a frame data structure to store the contents of the file. Each frame contains a recognizable byte sequence to identify the start of a frame. A MP3 file consists of a series of frames, as a result a MP3 file consists of a repeating sequence of frame data structures.

Figure A.3 provides a schematic overview of the layout of a MP3 file. The file format consists of a sequence of frames. Each frame starts with a recognizable frame header followed by frame data. A frame header contains the bitrate and sampling frequency of the audio stored in the frame data, the size of the frame data can be calculated based on the used bit rate and sampling frequency. Furthermore, the frame header indicates whether a CRC checksum of the frame data is available, the CRC checksum can be used to verify the frame data. The CRC value is stored at the end of each frame header and before the frame data. Audio data stored in the frame is encoded using Huffman encoding.

The frame data containing the huffman encoded audio data is unrecognizable. The maximum size of a frame can be calculated by using the following formula [YYP+12]: (144*bit rate/ (Sampling frequency + Padding)). The maximum bit rate of a MP3 file is 320000 bytes and the lowest sampling frequency is 32000 Hertz. This means that the maximum frame size is: (144 * 320000 / (32000 + 0)) = 1440 bytes. As long as the 1440 bytes stays within the cluster size of a file system, this does not have to be a problem. Since the CRC value can be used to validate the contents of a frame. This means that a validator can detect when the next cluster of a file system no longer forms a valid MP3 file. However, it can be a challenge to detect the difference between two MP3 files stored after each other, since the file format does not contain unique identifiers to identify the frames within the MP3 file.

The MP3 file format uses several steps before the audio data is encoded and is written

[3] https://www.iso.org/standard/25371.html

to a MP3 file. This process is not described, since we are only interested in how the data is organized in a MP3 file. Furthermore, not every detail of the frame header is described.

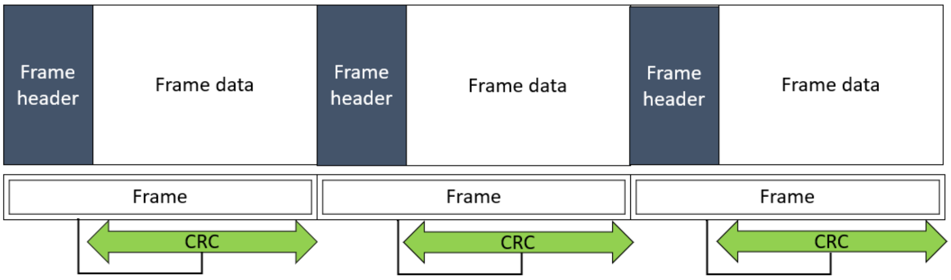

Figure A.3: MP3 file format diagram

A.3. Video

A.3.1. AVI

The AVI file format[4] is based on the resource interchange file format (RIFF). AVI files start with a RIFF header. This RIFF file header contains a file signature, this file signature is a four-character code (FOURCC). FOURCCs are used to identify sections in the file format. The file header also contains the file size of the file. File data is located after the file header. File data consist out of two types of data structures: a chunk data structure, which contains data (audio, video or text) and a list data structure, which contains other lists or chunks. Data structures can be recognized due to the use of FOURCCs.

Figure A.4 provides a schematic overview of the AVI file format. As mentioned before, the file starts with a RIFF header. Furthermore, the file format specifies the presence of two mandatory list chunks for AVI files: stream headers list (hdrl chunk) and a stream data list (movi chunk). These lists have to occur in the sequence as specified by the file format. The data structure of the hrdl and movi list is specified by the file format.

The stream headers list starts with a AVI main header structure, this structure contains metadata of the AVI file, such as the file size, the total amount of frames stored in the file and the video dimensions in pixels. Stream headers contain information about stream data in the file.

Stream data contain the actual video frames and audio samples. The stream headers that are stored in the hrdl list, describe the stream data that is contained in the stream data (movi) list. The movi list contains a list of the actual video frames. A chunk containing a video frame or an audio sample starts with a recognizable and specific byte sequence (FOURCC), for instance "00dc" for a compressed video frame. A frame header contains the size in bytes of a frame [YXLS17]. Optionally, an AVI file contains an index (idx1) structure, which contains a list of the frame offset relative to the start of the file (absolute file offset) and the frame size for each frame stored in the AVI file. The AVI file format does not contain CRC values that can be used to verify the contents of the file.

The data sections in Figure A.4 are marked as unrecognizable. Although the data chunks start with a recognizable byte sequence, for instance "00dc" for a compressed video frame, the length of the data chunks is arbitrary and can be larger than the cluster size of the file system. The length of a data chunk is stored in the trunk itself and can be used to

[4] https://docs.microsoft.com/en-us/windows/win32/directshow/avi-riff-file-reference

check whether another chunk is present at the end of the current chunk, by checking for a FOURCC of the next chunk. However, it is not possible to verify the contents of the data, since there is no error checking present in the file format. Also, if the data becomes corrupt from a certain point within the data section, it is not possible for a validator to pinpoint from which position the data is no longer valid within the data chunk. This property combined with the possibility that the size of a chunk is larger than the cluster size of a file system, resulted in that the data chunk is marked as unrecognizable in Figure A.4.

Figure A.4 is a simplified overview of an AVI file. Not all the details and fields stored within each type of chunk are discussed. Also the used encoding is not mentioned, because this depends on the used codec during the creation of a AVI file. The stream header contains a FOURCC that specifies which codec needs to be used to decode the data. The AVI file format can contain data that is encoded by a selection of encoders. The used documentation from Microsoft and Yang et al. [YXLS17] does not mention which selection of encoders is supported.

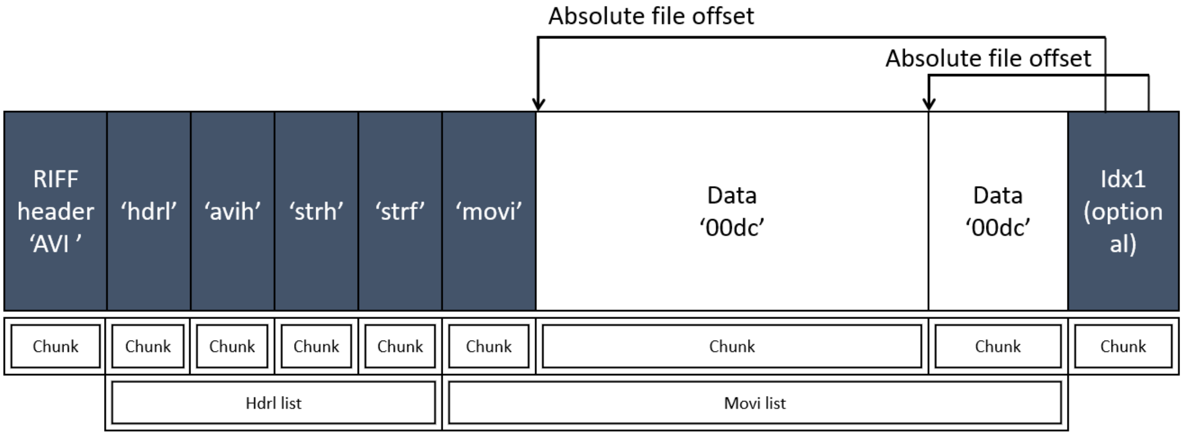

Figure A.4: AVI file format diagram

A.3.2. MKV

The Matroska (MKV) file format[5] is based on the Extensible Binary Meta Language (EBML). EMBL can be seen as the binary equivalent of an XML structure and as a result of this the sections within an MKV file are recognizable.

Figure A.5 contains a schematic overview of the MKV file format. Each MKV file starts with a recognizable EBML file header. The file format uses different sections, with a hierarchical structure. At the top level, the file consists of an EBML header and one (or more) segment. A segment consists of multiple other sections, including a collection of cluster sections. Please note, that not all the other sections that are present in a segment are explicitly mentioned.

Clusters can be identified by a specific ID. Clusters contain the audio and video data. These clusters contain the following information: timestamp information and optionally the position, which is a file offset. The size of the previous cluster is also stored in a cluster data structure. Furthermore, the file format can optionally make use of CRC checksums. A recommendation from the MKV specification is to split the audio and video data across

[5] https://datatracker.ietf.org/doc/draft-ietf-cellar-matroska/

multiple clusters and to store no more than 5 seconds or 5 megabytes of data in each cluster segment.

An MKV file thus consists of a sequence of cluster elements (repeating structures). Each cluster contains a SimpleBlock or BlockGroup element, this element contains for instance the frame data of a video. The encoding of audio and video data depends on the used codecs. Matroska is a container file format and does not state anything about which encoding should be used in the MKV file format. Instead, the MKV file format can be used to store audio and video data from different codecs. The file format specifies fields that contain metadata about which codecs were used to the store audio and video data. Furthermore, a selection of fields has a specific order defined by the file format.

The frame data element within a SimpleBlock element is indicated as an unrecognizable element. This has the following reasons: the frame data element itself can be of a arbitrary size, although the file format specification recommends to store no more than 5 seconds or 5 megabytes of data in each cluster segment. Thus the length of the section containing the frame data can be longer than the cluster size of a file system. Furthermore, the data within the frame data depends on the used codec. As a result it can be a challenge from a file format validation perspective, to recognize which data still belongs to the same encoded data section within the element containing the frame data.

When analyzing an MKV file, the biggest part of the file will consist out of audio and video data. Therefore, the focus of explaining how the MKV file is organized is on explaining how audio and video data is stored in an MKV file. MKV supports more features, such as subtitles and chapters. These features are implemented using similar concepts as how the audio and video data is stored in the MKV file format. By only explaining how the audio and video data is stored within the file format, it is clear which concepts are used and how the file format works. Figure A.5 as a result does not cover all the features that are present in the MKV file format and is a simplified view of the actual file format.

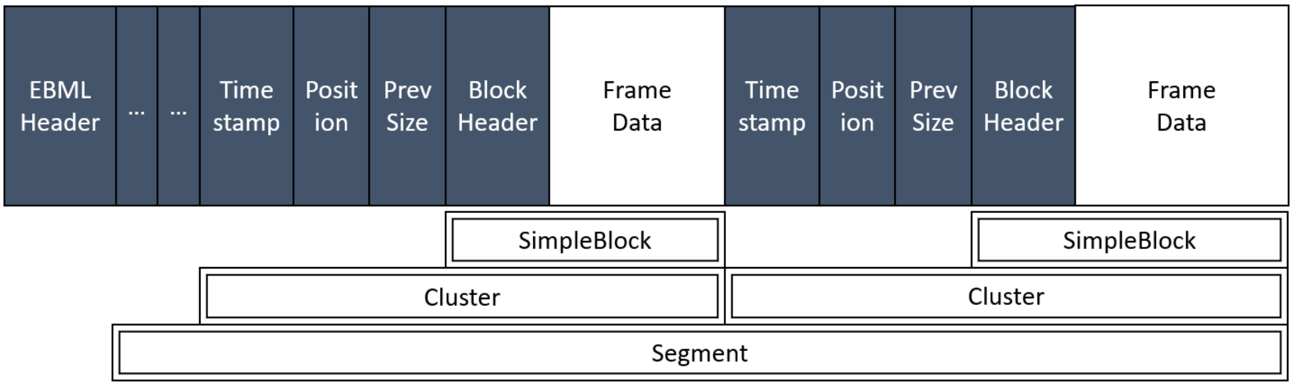

Figure A.5: MKV file format diagram

A.4. DOCUMENTS

A.4.1. OFFICE OPEN XML FILE FORMAT (DOCX, PPTX, XLSX)

The structure of the file format is described in the Open Packaging Conventions, which is described in part 2 of the ECMA-376 standard[6]. The Office Open XML file format (OOXML) is layered on top of the zip file format. Thus, an OOXML file is basically a zip file containing a collection of files.

OOXML files can contain multiple objects that are called parts. In case of a Word document, it can for instance contain a XML document containing the markup and a picture that is embedded in the Word document. Parts are stored as zip items in the zip file. Naming conventions are specified by the OOXML specification to identify the parts and to specify the hierarchy and relation with other parts. Parts can also be split into multiple pieces, in this case the pieces are stored as multiple separate zip items (files) in the zip container. The file format mandates that a content types stream is present in the form of a specific zip item in the root of the zip file. This content types stream is a XML file that contains the mapping between parts (zip items) and the content type of each part. The content types stream also indicates which type of file the OOXML is, for instance a Word document or Powerpoint presentation.

Since the OOXML file format uses the zip file format as a container, please refer to the ZIP file format paragraph and Figure 6.4 to learn more about the zip file format.

The description above provides a simplified overview of how the OOXML file format works. For instance, not every detail and rule about the contents of the zip file is introduced. However, given the information described above, it is clear that on top of the zip file format the OOMXL file format has an additional set of requirements and rules. These rules described above, give insight on how resources of an OOMXL file are organized.

From a file validation point of view, the additional requirements and rules of the OOMXL file format could be used to verify whether the data of the OOMXL file is valid. This should be done after checking whether the data forms a valid zip file in the first place, since the OOMXL file format is contained within a zip file, because if the OOMXL file does not form a valid zip file it also not a valid OOMXL file.

A.4.2. EPUB

EPUB files are stored in an EPUB container. An EPUB container is a ZIP based file format that is defined in the OCF ZIP Container specification[7]. The EPUB container is basically a ZIP file with specific rules regarding the contents of the ZIP file.

Since the EPUB file format uses the zip file format as a container, please refer to the ZIP file format paragraph and Figure 6.4 to learn more about the zip file format. In order to recognize a zip file as an epub file format, the first file entry in the zip file is always the "mimetype" file, the contents of the mimetype file contain a recognizable string that is used to identify the epub file format. Furthermore, the root directory of the zip file must contain a "META-INF" directory containing a container.xml file. This container.xml file is used to identify EPUB packages within the zip container. An EPUB package is a set of resources that represent a publication, for instance a book. The set of resources of an EPUB package are defined in a package document. The package document is an XML that contains the

[6] https://www.ecma-international.org/publications/standards/Ecma-376.htm

[7] https://www.w3.org/publishing/epub/epub-ocf.html

details of the resources of an EPUB package.

The description above provides a simplified overview of how the EPUB file format works. For instance, not every xml file that is stored in the META-INF" directory is introduced. However, given the information described above, it is clear that on top of the zip file format the EPUB file format has an additional set of requirements and rules. These rules described above, give insight on how resources of an EPUB file are organized.

From a file validation point of view, the additional requirements and rules of the EPUB file format could be used to verify whether the data of the epub file is valid. This should be done after checking whether the data forms a valid zip file in the first place, since the epub file format is contained within a zip file, because if the epub file does not form a valid zip file it also not a valid epub file.

A.5. Archive

A.5.1. Tar

The tar file format[8] is used as an archive/container to store files in. A tar file consists of a series of file entries that are preceded by corresponding header structure. The tar file format does not alter the data of the files stored within a tar archive.

Figure A.6 provides a schematic overview of the tar file format. Each file using the tar file format starts with a recognizable header structure (Ustar) that contains a magic string. This magic string can be used to recognize the header structure. Each file stored in the archive is proceeded by this header structure. A header structure contains metadata of the file stored in the tar archive. The metadata of the file contains the following information: file size, the name of the file (including directory, if any). After this header structure, the raw data of the file contained in the tar archive is stored. At the end of a tar file, the end of archive indicator (2 blocks of binary zeroes) should be placed. The tar file format uses no compression or encoding, the data of the files contained in a tar file are not altered. The file format contains error checking facilities. Each header has a field that contains the sum of all bytes contained in the header block, this allows basic error checking on the contents of the header. There are no error detecting facilities present for the file data section of each file entry.

As mentioned before, there are no error checking facilities for the file data for each file entry. Furthermore, there is no encoding or compression used on the file data for each file entry, the file data is stored unaltered within a tar file. As a result, it can be a challenge to recognize corruption within the file data of each entry. Especially when files contained in the tar archive are larger than the cluster size of a file system. It depends on the type of file that is stored in the tar file, if a validator exists for that specific file type, this validator could be used to check the consistency of the specific file entry within the tar file. The tar file format does not add a lot of recognizable data structures on top of each entry.

The tar file format is pretty straightforward, almost every aspect is described in this section. Some fields were omitted, because they are not relevant in the context of file validation or not required to understand how the tar file format works.

[8] https://www.gnu.org/software/tar/manual/html_node/Standard.html

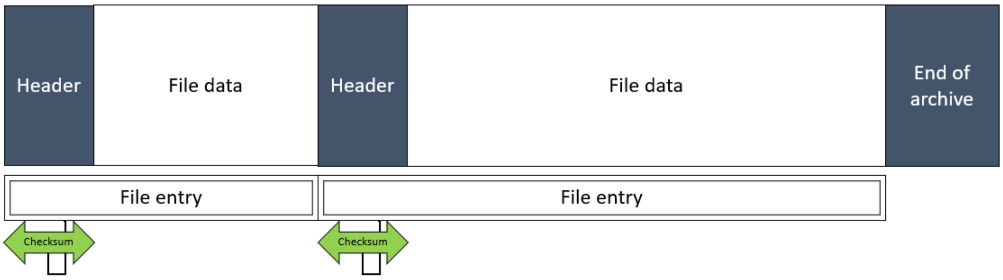

Figure A.6: TAR file format diagram

A.6. DATABASE

A.6.1. SQLITE

The SQLite file format[9] is used to store a SQLite database in a single database file. The database file consists of a series of data structures called pages. The size of a page is fixed, thus all pages within a database file have the same size and each page is of a specific type.

Figure A.7 provides a schematic overview of the SQLite file format. Each database file starts with a recognizable database header data structure that contains the following metadata: the database page size and the amount of pages stored in the database file. The complete file size of the database can be derived from the page size and the amount of pages used in the database file. Each page is of a specific type, there are the following page types: Lock-byte page, freelist page, b-tree page, payload overflow page and pointer map page. The page size is always a power of two between 512 and 65536 bytes. Each page in the database file is numbered, references to pages are implemented using the page number of the corresponding page.

The lock-byte page is only present at a specific location, if the file is larger than a specific size and there is only one lock-byte page (if applicable). The specification does not mention anything about a recognizable byte sequence for the lock-byte page type.

The free list page contains a list of pages that are free (possibly because of deleted information). The amount of free list pages is stored in the header and also the reference to the first free list page is stored in the header. The free list pages refer to each other by using a linked list. The specification does not mention anything about a recognizable byte sequence for the free list page type.

B-Tree pages start with a b-tree page header. The first byte of the b-tree page header describe the type of b-tree page. Only a limited set of values is valid for this field, thus these values can be used as recognizable byte sequences. B-tree pages contain the data that is stored in the database.

Payload overflow pages contain the data does not fit in a B-tree page. Payload overflow pages are linked lists and the first 4 bytes of the payload overflow page contain the page number of the next payload overflow page. The remaining space in the page is available for storing overflow data of the B-tree page. Payload overflow pages do not contain a recognizable byte sequence.

Pointer map pages are pages that contain references (page numbers) to other pages. Each entry in this list contains a page type (1-byte) followed by a page number (4-bytes). Pointer page maps do not have a header or recognizable byte sequence.

[9] https://www.sqlite.org/fileformat.html

Furthermore, the file format does not use encoding or compression or contain CRC values or other error checking facilities. The file format does not specify a specific sequence regarding the ordering of pages, the only requirement is that the file has to start with a database header structure.

Some of the page types can be recognized, such as b-tree pages. However, not all of the page types can be recognized or validated. For instance, the free list page type. Therefore, Figure A.7 contains page sections that are recognizable and unrecognizable. It is also possible to have free pages (grey shaded section) in a file, these pages are unused. As mentioned before, the data of the database is contained within b-tree pages. Thus, it is possible that the majority of the pages in a database file are b-tree pages, which are recognizable. If the page size is larger than the used cluster size of the file system, this can be a challenge for detecting fragmentation by a validator. Since it can be difficult to detect fragmentation or corruption within a page, because there are no error checking facilities present in the file format.

Only the lower level of the SQLite file format is described in this section, this lower level contains the b-tree layer. The b-tree layer is used to implement the capabilities of SQL, how this is accomplished is not described. In order to understand how the data is organized at file level, it is not required to know how the b-tree layer is used to implement SQL capabilities. Therefore, this information is omitted. Furthermore, not every detail or field of each page type is described, only the information relevant in the context of file validation or information that is required to understand the file format is mentioned.

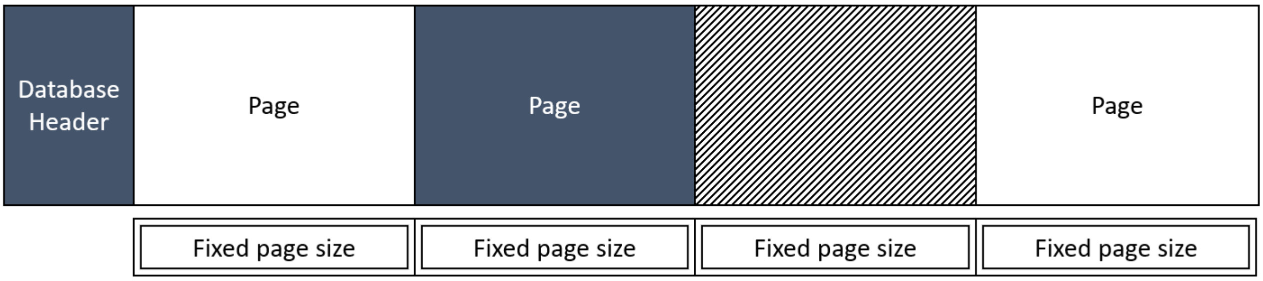

Figure A.7: SQLite file format diagram

B

PST File format

This chapter describes the internals of the PST file format. Information in this chapter is based on the documentation of the PST file format provided by Microsoft [Mic20].

B.1. Grammar

- <PSTfile> ::= <Minimal> | <Minimal> <AdditionalStorage>
- <Minimal> ::= <Header> <Reserved> <DList> <AMap> <PMap> <Data>
- <Header> ::= **<dwMagic> <dwCRCPartial> <wMagicClient> <wVer> <wVerclient> <bPlatformCreate> <bPlatformAccess> <dwReserved1> <dwReserved2> <bidUnused> <bidNextP> <dwUnique>** <rgnid> **<qwUnused>** <root> **<dwAlign> <rgbFM> <rgbFP> <bSentinel> <bCryptMethod> <rgbReserved> <bidNextB> <dwCRCFull> <rgbReserved2> <bReserved> <rgbReserved3>**
- <rgnid> ::= <nid> | <rgnid> <nid>
- <nid> ::= **<nidType> <nidIndex>**
- <root> ::= **<dwReserved> <ibFileEof> <ibAMapLast> <cbAMapFree> <cbPMapFree>** <BREFNBT> <BREFBBT> **<fAMapValid> <bReserved> <wReserved>**
- <BREFNBT> ::= <BREF>
- <BREFBBT> ::= <BREF>
- <AdditionalStorage>::= AdditionalData | <AdditionalData> <AdditionalData> <AdditionalData> <AdditionalData> <AdditionalData> <AdditionalData> <AdditionalData> <AdditionalData> <PMap> | < 128MB AdditionalStorage> <FMap> | < 8GB AdditionalStorage> <FPMap>
- <AdditionalData> ::= <AMap> <Data>
- <Data> ::= <Datablock> | <Page> | <Data> <Datablock> | <Data> <Page>
- <Page> ::= <Page Content> <Page Trailer>
- <Page Content> ::= <AMapPage> | <PMapPage> | <DListPage> | <FMapPage> | <FPMapPage> | <BTPage>
- <AMap> ::= <AMapPage> <Page Trailer>
- <AMapPage> ::= **<rgbAMapBits>**
- <PMap> ::= <PMapPage> <Page Trailer>
- <PMapPage> ::= **<rgbPMapBits>**
- <DList> ::= <DListPage> <Page Trailer>

- <DListPage> ::= **<bFlags> <cEntDList> <wPadding> <ulCurrentPage>** <rgDListPageEnt> **<rgPadding>**
- <rgDListPageEnt> ::= <DLISTPAGEENT> | <rgDListPageEnt> <DLISTPAGEENT>
- <DLISTPAGEENT> ::= **<dwPageNum> <dwFreeSlots>**
- <FMap> ::= <FMapPage> <Page Trailer>
- <FMapPage> ::= **<rgbFMapBits>**
- <FPMap> ::= <FPMapPage> <Page Trailer>
- <FPMapPage> ::= **<rgbFPMapBits>**
- <BBT> ::= <BTPage> <Page Trailer>
- <NBT> ::= <BTPage> <Page Trailer>
- <BTPage> ::= <rgentries> **<cEnt> <cEntMax> <cbEnt> <cLevel> <dwPadding>**
- <rgentries> ::= <BTENTRY> | <BBTENTRY> | <NBTENTRY> | <rgentries> <BTENTRY> | <rgentries> <BBTENTRY> | <rgentries> <NBTENTRY> * (all entries in the rgentries are of the same type, either BBTENTRY, BTENTRY or NBTENTRY)
- <BTENTRY> ::= **<btkey>** <BREF>
- <BBTENTRY> ::= <BREF> **<cb> <cRef> <dwPadding>**
- <NBTENTRY> ::= <nid> **<bidData> <bidSub> <nidParent> <dwPadding>**
- <BREF> ::= <bid> **<ib>**
- <Page Trailer> ::= **<ptype> <ptypeRepeat> <wSig> <dwCRC>** <bid>
- <Datablock> ::= **<Block data> <Padding>** <Block Trailer>
- <Block Trailer> ::= **<cb> <wSig> <dwCRC>** <bid>
- <bid> ::= **<A><B><bidIndex>**

B.2. List of fields

The following fields are used in the PST file format (unicode variant):

dwMagic (4 bytes) fixed value magic string " 0x21, 0x42, 0x44, 0x4E ("!BDN")".

dwCRCPartial (4 bytes) CRC value of the 471 bytes of data starting from wMagicClient field

wMagicClient (2 bytes) fixed value magic string " 0x53, 0x4D ".

wVer (2 bytes) File format version.

wVerclient (2 bytes) Client file format version.

bPlatformCreate (1 byte) fixed value 0x01.

bPlatformAccess (1 byte) fixed value 0x01.

dwReserved1 (4 bytes) Value must be ignored and must be initialized to zero.

dwReserved2 (4 bytes) Value must be ignored and must be initialized to zero.

bidUnused (8 bytes) Unused padding used in the Unicode version of the PST file format.

bidNextP (8 bytes) Next page BID. BidIndex values of pages are allocated using this counter.

dwUnique (4 bytes) Monotonically increasing value that is modified each time the header of the PST file is modified.

qwUnused (8 bytes) Unused space that must be set to zero.

dwAlign (4 bytes) Unused alignment bytes that must be set to zero.

rgbFM (128 bytes) Deprecated FMap, this field is no longer used and must be set to 0xFF, the field should be ignored by readers.

rgbFP (128 bytes) Deprecated FPMap, this field is no longer used and must be set to 0xFF, the field should be ignored.

bSentinel (1 byte) fixed value 0x80.

bCryptMethod (1 byte) Indicates which method is used to encode the data within the PST file.

rgbReserved (2 bytes) Reserved field that must be set to zero.

bidNextB (8 bytes) Next BID, the value of this field is a monotonic counter that indicates which BID is assigned for the next allocated block. BID values increase in increments of 4.

dwCRCFull (4 bytes) CRC value of the 516 bytes of data starting from the wMagicClient field up to the bidNextB field.

rgbReserved2 (3 bytes) Value must be ignored and must be initialized to zero.

bReserved (1 byte) Value must be ignored and must be initialized to zero.

rgbReserved3 (32 bytes) Value must be ignored and must be initialized to zero.

nidType (5 bits) Identifies which type of the node is represented by the NID (Node ID).

nidIndex (27 bits) Identification part of the NID (Node ID).

dwReserved (4 bytes) Value must be ignored and must be initialized to zero.

ibFileEof (8 bytes) Size in bytes of the complete PST file.

ibAMapLast (8 bytes) Contains the absolute file offset to the last AMap page of the PST file.

cbAMapFree (8 bytes) Total amount of free space in all AMaps combined.

cbPMapFree (8 bytes) Total amount of free space in all PMaps combined, PMap is deprecated therefore the value must be set to zero.

fAMapValid (1 byte) Indicates whether all of the AMaps that are stored in the PST file are valid.

wReserved (2 bytes) Value must be ignored and must be initialized to zero.

rgbAMapBits (496 bytes) AMap data.

rgbPMapBits (496 bytes) PMap data.

bFlags (1 byte) Flags related to DList.

cEntDList (1 byte) contains the number of entries in the rgDListPageEnt array.

wPadding (2 bytes) Padding bytes, always zero.

ulCurrentPage (4 bytes) If the backfill complete is active in bFlags, this field contains the AMap page index that is used in the next allocation. If the backfill complete is not active in the bFlags field, this value contains the AMap page index that is attempted for backfilling in the next allocation.

rgPadding (12 bytes) Unused padding, always zero.

dwPageNum (20 bits) AMap page number.

dwFreeSlots (12 bits) Total number of free slots of the AMap.

rgbFMapBits (496 bytes) FMap data.

rgbFPMapBits (496 bytes) FPMap data.

cEnt (1 byte) The number of BTree entries stored in a page.

cEntMax (1 byte) The maximum number of entries allowed in the page data

cbEnt (1 byte) BTree entry size in bytes.

cLevel (1 byte) The binary tree depth level of this page, leaf pages are zero.

dwPadding (4 bytes) Padding, always zero value.

btkey (8 bytes) The value of the key of the BTENTRY record.

cb (2 bytes) The amount of bytes of data the data section of the block contains.

cRef (2 bytes) The amount of references to this block.

bidData (8 bytes) The value of the BID of the data block.

bidSub (8 bytes) The value of the BID of the subnode block.

nidParent (4 bytes) contains the NID of the parent Folder object's node if applicable.

bid (64 bits) Block ID

nid (8 bytes) Node ID

ib (64 bits) Byte index, this value is used to indicate the absolute offset within the PST file relative to the start of the PST file.

ptype (1 byte) Indicates the type of data of the page.

ptypeRepeat (1 byte) the same value as the ptype field.

wSig (2 bytes) Page signature (in case of a page) or block signature (in case of data block)

dwCRC (4 bytes) CRC value of the page data, the page trailer is not included. In case of a data block this field contains the CRC value of the amount of bytes of raw data.

Padding (4 bytes) reserved, padding is used to guarantee the data block is always a multiple of 64 bytes. The content of the padding field is undetermined and not guaranteed to be zero.

A (1 bit) part of the bid value, reserved bit which is always 0.

B (1 bit) part of the bid value.

bidIndex a monotonically increasing value that is part of the bid value, that uniquely identifies the BID with the PST file.