ERNW
providing security.

ERNW WHITEPAPER 65

APFS INTERNALS FOR FORENSIC ANALYSIS

Version:	1.0
Date:	4/16/2018
Classification:	Public
Author(s):	Andreas Dewald, Jonas Plum (Siemens CERT)

Matthias Luft

Digitally signed by Matthias Luft
Datum: 2018.04.16 16:44:07 +02'00'

TABLE OF CONTENTS

ABSTRACT

In forensic computing, especially in the field of post-mortem file system forensics, the reconstruction of lost or deleted files plays a major role. The techniques that can be applied to this end strongly depend on the specifics of the file system in question. Various file systems are already well-investigated, such as FAT16/32, NTFS for Microsoft Windows systems and Ext2/3/4 as the most common Linux file system and HFS/HFS+ for macOS. There also exist tools, such as the famous Sleuthkit by Brian Carrier that provide file recovery features for those file systems by interpreting the file system's internal data structures. APFS is the new file system for Apple devices that is applied by default on all current iOS mobile devices, as well as macOS since High Sierra. For APFS that is currently being rolled out on a large number of devices, no forensic file recovery methodologies have been developed so far. To allow for manual analysis or development of forensic file recovery methods, a deeper understanding of the internal structures of the file system is necessary. In this paper, we analyse APFS and describe its internal structures to provide forensic/incident analysts with the necessary knowledge to this end.

1 Introduction

Persistent storage devices are still one of the most important sources of digital evidence in digital investigations. In particular, the forensically sound reconstruction of deleted files alongside with their metadata is an important step in the forensic process. To this end, usually two distinct kinds of methods are applied: file carving or file system parsing. File carving methods try to identify files content of different file formats by common patterns (magic bytes). The advantage of those methods is that the underlying file system that was used to store the files is not relevant and does not need to be understood. Therefore, carving still works for newly introduced and unknown file systems. However, this comes with several disadvantages: First, only file types for which well-known patterns exist can be reconstructed. Further, besides special approaches for specific file types, file carving is mostly not able to handle file fragmentation. And last, but in the forensic context especially important, file carving is not able to obtain metadata, such as timestamps or even file names and directory structures, as this information is only held in the internal structures of the file system. On the other hand, parsing of file system data provides all this information, although file recovery possibilities can be limited, depending on the particular file system (Carrier, File system forensic analysis, 2005). The major problem of file system parsing, however, is that for each upcoming file system, old tools do not provide any results at all, unless the internals of the new file system are studied, specific methods for file recovery have been developed and implemented.

1.1 Motivation

There exist forensic approaches and tools, such as the famous Sleuthkit (Carrier, The sleuth kit (TSK), 2010), for almost all common file systems, such as FAT 16/32, NTFS, Ext2/3/4, HFS, and HFS+. Apple recently introduced their new file system APFS (Apple File System), which is rolled out on all current iOS and macOS devices. So, there is a need for the analysis of APFS. Up to now, there already have been some publications on APFS internals, which we revisit in Section 2. This previous work leaves some gaps that we try to close in this paper.

1.2 Related Work

Apple (Apple Inc., Technical note tn1150: Hfs plus volume format, 2004) released documentation for HFS+, where they describe the specifications of the file system. While APFS works quite different, some concepts, such as the use of B-trees and records are used similar in APFS. Further, Apple (Apple Inc., Apple file system guide, 2017) released information about some features of APFS. However, this information contains not much information about internal structures of the file system. Some of the described features on a high level are clones, snapshots, encryption, copy-on-write, and sparse files. Regarding internal structures, Apple states that they use 64-bit file IDs, nanosecond timestamps, 263 allocation blocks and 263 bytes of maximum file size and

that file names are encoded in UTF-8. Potentially, there might be more official documentation in the future, as Apple states ‘‘An open source implementation is not available at this time. Apple plans to document and publish the APFS volume format specification’’ (Apple Inc., Technical note tn1150: Hfs plus volume format, 2004). Hansen and Toolan (Hansen & Toolan, 2017) describe some artefacts and structures of APFS, of which we were able to verify, update, or complete, as described in detail in Section 2, where we highlight the differences between their work and ours explicitly.

2 APFS data structures

In their article, Hansen and Toolan [2017] have described many APFS structures. The following sections list structures which are present in APFS including contributions by Hansen and Toolan [2017]. The reverse engineering process of the file system was made independently from Hansen and Toolan [2017] and multiple new findings have been added. The APFS version used for this publication is version 748.31.8, while Hansen and Toolan [2017] use version 249.20.2 and 249.30.8. The names used in the following sections are taken from the fsck_apfs command, which is included in macOS. All strings from this command were extracted to keep close to the official Apple naming schema, while Hansen and Toolan [2017] create their own naming schema. In some structures, all fields contain a prefix (e.g. o_oid, o_type in the object header). These prefixes are omitted. Moreover, many additions and changes have been made in the structures.

APFS is structured in a single container that can contain multiple APFS volumes. A container needs to be >512 MB to contain more than one volume, >1024MB to contain more than two volumes and so on. Figure 1 shows an overview of the APFS structure. The file system uses little-endian values for storing information. Strings are stored in UTF-8 encoding and timestamps are 64bit nanoseconds starting from 1.1.1970 UTC (Unix epoch). Standard block size is 4096 bytes per block. APFS is a copy-on-write file system so each block is copied before changes are applied so a history of all files which were not overwritten and file system structures exists.

2.1 Object

Except for the Bitmap all file system data is stored as objects. All objects have a 32-byte header. Table 1 displays the structure of this header.

pos	size	type	id
0	8	u8le	cksum
8	8	u8le	oid
16	8	u8le	xid
24	2	u2le	type
26	2	u2le	flags
28	2	u2le	subtype
30	2	u2le	padding

Table 1: Object Header Structure

The first eight bytes of the header are a variant of the Fletcher's checksum (cksum) as described by Kodis (Kodis, 1992). The original algorithm was adapted to 64bit input. The checksum is calculated using the data of the block of the object without the first 8 byte. The algorithm for calculating the checksum is shown in Listing 1.

```
func createChecksum(data []byte) uint64 {
	var sum1, sum2 uint64
	mod := uint64(2<<31 - 1)
	for i := 0; i < len(data)/4; i++ {
		chunk := data [i*4 : (i+1)*4]
		d := binary.LittleEndian.Uint32(chunk)
		sum1 = (sum1 + uint64(d)) % mod
		sum2 = (sum2 + sum1) % mod
	}
	check1 := mod - ((sum1 + sum2) % mod)
	check2 := mod - ((sum1 + check1) % mod)
	return (check2 << 32) | check1
}
```

Listing 1: Checksum calculation

The checksum is followed by the oid of the object which is used for referencing objects. The xid describes the version of an object. It is incremented when the object is changed. The next two bytes describe the type of the object. Currently nine object types are known:

- **0x1** Container Superblock
- **0x2** Rootnode
- **0x3** Node
- **0x5** Space Manager
- **0x7** Space M. Internal Pool
- **0xB** B-Tree
- **0xC** Checkpoint
- **0xD** Volume Superblock
- **0x11** Reaper

The Bitmap object does not have an object type. The object type is followed by flags. The subtype is used to further distinguish rootnodes and nodes. Six different subtypes are known:

- **0x0** No Subtype
- **0x9** History
- **0xB** Location
- **0xE** Files
- **0xF** Extents
- **0x10** Unknown

The object header is closed by a two-byte padding.

2.2 Container Superblock

The container superblock is the entry point to the file system and is located in the first block of the file system. This does not apply for improperly unmounted file systems, handling of these states it not in scope of this paper. The fields of the container superblock are listed in Table 2. Every container superblock has a unique uuid. Because of the

structure of containers and flexible volumes, allocation needs to be handled at a container level. The container superblock contains information on the block_size and the number of blocks (block_count) for

this task. Feature compatibility of the file system is managed in the fields features, read_only_compatible_features and incompatible_features. The next available oid and xid for the object headers are saved in next_oid and next_xid.

The next ten fields are used to point to two different areas for checkpoints as displayed in 1. Checkpoints are used to store the current and older states of the container. The first area for checkpoints is the checkpoint descriptor area. This area stores the checkpoint object and more versions of the container superblock. The checkpoint descriptor area has an allocated space of xp_desc_blocks. This allocated area starts at the block at offset xp_desc_base and the xp_desc_len field contains the currently used blocks in the area. The xp_desc_index field points to the currently used checkpoint descriptor, which starts with a checkpoint object. This position is relative to the xp_desc_base offset. The xp_desc_index_len field contains the object count of the current checkpoint descriptor.

The values for the second area, the checkpoint data area, are used equally. The current checkpoint data starts with a Space Manager object and can contain (root-) node and reaper objects.

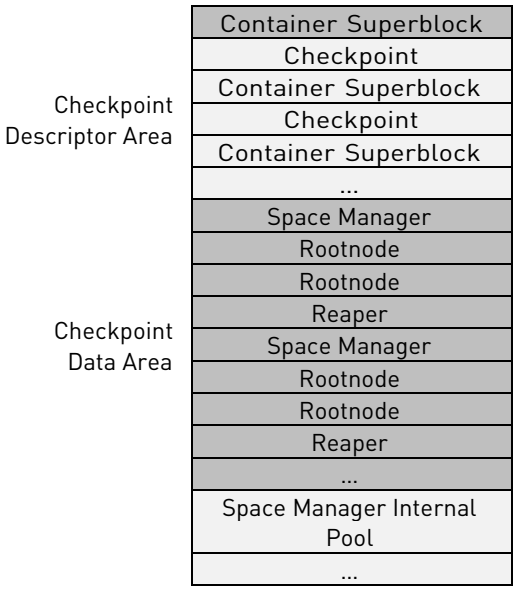


Figure 1: Layout of the checkpoint structures

Following these checkpoint area fields references to multiple other objects are listed: spaceman_oid, omap_oid and reaper_oid. Additionally, the block IDs of all volumes are stored in the superblock in fs_oid with the count of this list stored in max_file_systems.

pos	size	type	id
0	4	str	magic 'NXSB'
4	4	u4le	block_size
8	8	u8le	block_count
16	8	u8le	features
24	8	u8le	read_only_compatible_features
32	8	u8le	incompatible_features
40	16		uuid
56	8	u8le	next_oid
64	8	u8le	next_xid
72	4	u4le	xp_desc_blocks
76	4	u4le	xp_data_blocks
80	8	u4le	xp_desc_base
88	8	u4le	xp_data_base
96	4	u4le	xp_desc_len
100	4	u4le	xp_data_len
104	4	u4le	xp_desc_index
108	4	u4le	xp_desc_index_len
112	4	u4le	xp_data_index
116	4	u4le	xp_data_index_len
120	8	u8le	spaceman_oid
128	8	u8le	omap_oid
136	8	u8le	reaper_oid
152	4	u4le	max_file_systems
160	8	u8le	fs_oid

Table 2: Container Superblock

2.3 Rootnode and Node

Nodes are flexible containers that are used for storing different kind of entries. They can be part of a B-tree or exist on their own. A node starts with a node header as described in Table 3. The node header starts with the node_type which defines some characteristics of the node. The level which describes the depth of the node in the B-tree. The level is zero for leaf nodes and > 0 for index nodes. The following attributes define the entry_count, the position of key section (keys_offset and keys_length) and data section data_offset. The purpose of the meta_entry is currently unknown. This node header information is followed by a list of entry headers that act as pointers to the entry keys and entry values. This way for each entry the node contains an entry header at the beginning of the node, an entry key in the middle of the node and an entry value at the end of the node as displayed in Figure 2.

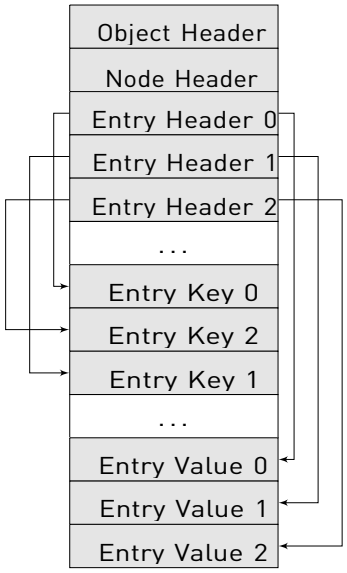


Figure 2: Interal Structure of Nodes

pos	size	type	id
0	2	u2le	node_type
2	2	u2le	level
4	4	u4le	entry_count
10	2	u2le	keys_offset
12	2	u2le	keys_length
14	2	u2le	data_offset
24	...	EntryHead	entry_heads
...	...	EntryKey	entry_keys
...	...	EntryValue	entry_values

Table 3: Node Structure

Nodes contain different kinds of entries. The following entries have been observed in APFS images:

- 0x0 omap
- 0x2 lookup
- 0x3 inode
- 0x4 xattr
- 0x5 sibling
- 0x6 extent_status
- 0x8 extent
- 0x9 drec
- 0xc unknown

All entry types have the same entry header structure, but different key and value structure. Those structures are described in the following sections. The structure with the type 0xc is currently unknown.

2.3.1 Entries

The entry header structure is the same for all node entries and is defined in Table 4. It consists of the position for the entry key (key_offset and key_size) as well as the position of the related values (val_offset and val_size). While the node header contains the offset to all keys the key_offset in this header is especially for one key. The offset with regard to the object start is calculated with 32 (object header size) + 24 (node header size) + keys_offset (from the node header) + key_offset (from the entry header).

pos	size	type	id
0	2	s2le	key_offset
2	2	u2le	key_size
4	2	s2le	val_offset
6	2	u2le	val_size

Table 4: Entry Header Structure

2.3.2 Entry Keys

All entry keys start with a 8 byte field. The highest byte determines the kind as described in the list above, while the remaining bytes contain the obj_id. The obj_id is the first part of the key and contains the id of the referenced element. This can be a file_id or an objects oid. The obj_id can be followed by more fields depending on the entry type. This content is described in the sections of the relating entry. The entries are sorted in a node by their complete key which contains the key_value as well as the individual key content.

2.3.3 Pointer Value

Pointer values do have a special role. For every entry the value is a pointer value, if the node_type of the node is 2. This is independent of the entry value_type in the key. Pointer values are used for the structure of B-tree and direct to other nodes using the pointer attribute. Table 5 displays the structure of pointer values.

pos	size	type	id
0	8	u8le	pointer

Table 5: Pointer Value Structure

2.3.4 omap Entry

omap entries are used for mapping object oids to block offsets. The keys obj_id contains the oid and the key content consists the xid as shown in Table 6. The omap entry value (Table 7) contains the position data for the object (physikal address (paddr), size and obj_id).

pos	size	type	id
0	8	u8le	kind & obj_id
8	8	u8le	xid

Table 6: omap Entry Key Structure

pos	size	type	id
0	4	u4le	paddr
4	4	u4le	size
8	8	u8le	obj_id

Table 7: omap Entry Value Structure

2.3.5 Lookup Entry

Lookup entries provide the possibility for a reverse lookup from blocks to file_ids. The keys (Table 8) contain an objects (offset), while the value (Table 9) contains block_count and block_size which span the allocated section of the file as well as the file_id to find the related entries.

pos	size	type	id
0	8	u8le	kind & obj_id
8	16	u8le	offset

Table 8: Lookup entry key structure

pos	size	type	id
0	4	u4le	block_count
6	2	u2le	block_size
8	8	u8le	file_id

Table 9: Lookup Entry Value Structure

2.3.6 Inode Entry

The inode value contains metadata about files and folders. It does not have any extra key content besides the kind and obj_id. The keys obj_id contains the file_id of the file for that the metadata is stored.

pos	size	type	id
0	8	u8le	parent_id
8	8	u8le	file_id
16	8	u8le	creation_timestamp
24	8	u8le	modified_timestamp
32	8	u8le	changed_timestamp
40	8	u8le	accessed_timestamp
48	8	u8le	flags
56	4	u4le	nchildren_or_nlink
68	4	u4le	bsd_flags
72	4	u4le	owner_id
76	4	u4le	group_id
80	2	u2le	mode
92	2	u2le	xf_num_ext
94	2	u2le	xf_used_data
96	...	xf_header	xf_header
...	...	xf_field	xfields

Table 10: Inode Entry Value Structure

pos	size	type	id
0	2	u2le	type
2	2	u2le	length

Table 11: xfield header

The inode entry value (Table 10) starts with a reference parent_id to the parent folders file_id. The following file_id might contain an id that is different from the normal file_id if the file was cloned. It is used for matching inode entries with extent entries. The next four attributes are timestamps (creation_timestamp, modified_timestamp, changed_timestamp and accessed_timestamp). The next eight bytes contain flags that need further investigation. The nchildren_or_nlink field contains information on the number of contained files for a folder or the number of links to the file. The fields bsdflags, owner_id, group_id and mode contain access information. The xf_num_ext field contains the count of the following extended fields. Extended fields contain additional information about files and folders, most importantly name and filesize are stored here. xf_used_data is the number of bytes for all extended fields. xf_header is a list of type and length values of the following extended fields, as shown in Table 11.

The extended fields contain information based in their type, as follows:

- **Type 0x204** name (string)
- **Type 0x2008** size (u8le)

Extended fields are aligned at multiples of 8 byte, so padding between extended fields might exist.

2.3.7 xattr Entry

The xattr entry has no additional key content. The keys obj_id contains the file_id of the file for that the attribute is stored. The xattr entry value (Table 12) starts with the type of the attribute in the xattr_obj_id field. Currently the type **0x2** for generic attributes and **0x6** for symlinks are known. The next two bytes xdata_length define the size of the stored data. The dstream field varies by type and contain the attribute information.

pos	size	type	id
0	2	u2le	xattr_obj_id
2	2	u2le	xdata_len
4	...		dstream

Table 12: xattr Entry Value Structure

2.3.8 Sibling Entry

Sibling entries are used for hardlinks. The structure of sibling entries is displayed in Table 14, they contain length and name fields. In addition, they contain a file_id reference to the file the hardlink links to. Their key (Table 13) contains a second 8-byte object oid (id2) with unknown purpose.

pos	size	type	id
0	8	u8le	kind & obj_id
8	8	u8le	object

Table 13: Sibling Entry Key Structure

pos	Size	type	id
0	8	u8le	file_id
8	2	u2le	length
10	length	str	name

Table 14: Sibling Entry Value Structure

2.3.9 Extent Status Entry

The extent status entry precedes one or more extent entries and contains the number of extent values in the 4-byte extent_count field as listed in Table 15.

pos	size	type	id
0	4	u4le	extent_count

Table 15: Extent Status Entry Value Structure

2.3.10 File extent Entry

File extent (fext) entries contain information about the position and size of file content. One file may have an arbitrary number of extents. The key (Table 16) contains, besides the obj_id the 8-byte offset into the file data, so that multiple extent values are sorted in order. The extent entry value (Table 17) first contains the len of the extent. This is usually a multiple of the file systems block_size. The second field phys_block_num contains the offset of the extent in blocks. The remaining bytes contain unknown flags.

pos	size	type	id
0	8	u8le	kind & obj_id
8	8	u8le	offset

Table 16: Extent Entry Key Structure

pos	size	type	id
0	8	u8le	len
8	8	u8le	phys_block_num
8	8	u8le	flags

Table 17: Extent Entry Value Structure

2.3.11 Directory Record (dcrec) Entry

A dcrec entry contains the parents file_id as a obj_id preceded by a length and name in the drec entry key (Table 18). The drec entry value (Table 19) contains the file_id, a creation timestamp as well as a type (file or folder). It is used to provide fast directory listings.

pos	size	type	id
0	8	u8le	kind & obj_id
8	1	byte	length
12	...	str	name

Table 18: Directory Record Entry Key Structure

pos	size	type	id
0	8	u8le	file_id
8	8	u8le	timestamp
16	2	u2le	type

Table 19: Directory Record Entry Value Structure

2.4 Space Manager

The space manager (sometimes called spaceman) is used to manage allocated blocks in the APFS container. Its structure is shown in Table 20. It contains numerous sizes for different elements of the file system: block_size, blocks_per_chunk, chunks_per_cib, cibs_per_cab, block_count, chunk_count, cib_count and cab_count. The number of free blocks is stored in free_count. Additionally, it contains references to the space manager internal pool in list of spaceman_internal_pool_blocks with entry_count elements at entries_offset. A reference to the previous pool is saved in the field prev_spaceman_internal_pool_block.

pos	size	type	id
0	4	u4le	block_size
4	4	u4le	blocks_per_chunk
8	4	u4le	chunks_per_cib
12	4	u4le	cibs_per_cab
16	4	u4le	block_count
20	4	u4le	chunk_count
24	4	u4le	cib_count
28	4	u4le	cab_count
32	4	u4le	entry_count
40	8	u8le	free_count
48	4	u4le	entries_offset
144	8	u8le	prev_spaceman_internal_pool_block
...	...	u8le	spaceman_internal_pool_blocks

Table 20: Space Manager Structure

2.5 Space Manager Internal Pool

The space manager internal pool works as a missing header for the bitmap files. It contains a list of entries as the structure in Table 21 shows. Those entries store the bitmap files bitmap_block_xid, total bm_block_count, bitmap_free_blocks and the position (bitmap_block) as listed in Table 22.

pos	size	type	id
4	4	u4le	entry_count
8	...	SpacemanInternalPoolEntry	entries

Table 21: Space manager Internal Pool structure

pos	size	type	id
0	8	u8le	bitmap_block_xid
16	4	u4le	bm_block_count
20	4	u4le	bitmap_free_blocks
24	8	u8le	bitmap_block

Table 22: SpacemanInternalPoolEntry

2.5.1 Bitmap File

Bitmap files are used to state the allocation status of blocks. They do not have an object header and therefore no type id.

2.6 B-Tree

B-trees manage multiple nodes. The structure of the B-tree is quite simple, as shown in Table 23. There are currently two known btree_types: 0 for object map B-trees and 1 for file system B-trees, which are used to resolve the fs_oids from the container superblock. Both types contain the offset of the root node.

pos	size	type	id
0	8	u8le	btree_type
16	8	u8le	root

Table 23: B-Tree Structure

2.7 Checkpoint

A checkpoint contains a list of checkpoint entries (Table 24) which reference objects (type, flags, subtype, size, oid, object) as listed in Table 25.

pos	size	type	id
4	4	u4le	entry_count
8	...	CheckpointEntry	entries

Table 24: Checkpoint Structure

pos	size	type	Id
0	2	u2le	type
2	2	u2le	flags
4	4	u4le	subtype
8	4	u4le	size
24	8	u8le	oid
32	...	u8le	object

Table 25: CheckpointEntry structure

2.8 Volume Superblock

A volume superblock exists for each volume in the file system. Its structure is displayed in Table 26.

pos	size	type	id
0	4	str	magic 'APSB'
4	4	u4le	fs_index
24	4	u4le	features
40	8	u4le	fs_reserve_block_count
48	8	u4le	fs_quota_block_count
56	8	u4le	fs_alloc_count
96	8	u8le	omap_oid
104	8	u8le	root_tree_oid
112	8	u8le	extentref_tree_oid
120	8	u8le	snap_meta_tree_oid
144	8	u8le	next_doc_id
152	8	u8le	num_files
160	8	u8le	num_directories
168	8	u8le	num_symlinks
176	8	u8le	num_other_fsobjects
184	8	u8le	num_snapshots
208	16		vol_uuid
224	8	u8le	last_mod_time
232	8	u8le	formatted_by.last_xid
264	32	str	formatted_by.id
272	8	u8le	formatted_by.timestamp
280	8	u8le	modified_by.last_xid
288	32	str	modified_by.id
320	8	u8le	modified_by.timestamp
672	...	str	volname

Table 26: Volume Superblock Structure

The Volume Superblock starts with the magic bytes 'APSB'. The second file contains the fs_index of the current volume. Feature compatibility of the volume is managed in the features field, similarly to the container superblock.

While allocation management is done at container level, volume might contain quotas. Those are managed with the fields fs_reserve_block_count, fs_quota_block_count and fs_alloc_count. References to the omap_oid, the root_tree_oid, and the extentref_tree_oid and the snap_meta_tree_oid are the next fields. Similarly, to the container superblock the omap_oid contains a pointer to a block map which maps block IDs to block offsets. The number of elements currently stored in the volume is saved in the fields num_files,num_directories,num_symlinks,num_other_fsobjects and num_snapshots. A uuid for every volume is stored in vol_uuid. The time of the last update (last_mod_time) is stored with no further information, while the formatted_by.timestamp is linked with an xid (formatted_by.last_xid) and a string containing the volume creator (formatted_by.id, e.g. *newfs_apfs (748.31.8)*). A second similar entry with modified_by.last_xid, modified_by.id and modified_by.timestamp exists referencing the apfs kernel extension. The last information stored in the volume superblock is the volume name in volname.

2.9 Reaper

The reaper structure contains only very sparse information. Its purpose is yet unknown.

3 APFS composition

Figure 3 shows and illustrates the overall design of APFS and the interplay of the previously explained structures. The first central structure in APFS when it comes to parsing the file system, is the Container Superblock. Amongst other information (see details given before), it contains pointers to the Checkpoint Descriptor and the Checkpoint Data. The Checkpoint Descriptor contains the Checkpoint itself and copies of (older) Container Superblocks. Those Container Superblocks follow the Checkpoint blocks immediately. The Checkpoint data contains the Spacemanager, History Blocks and Reaper that present a view on the entire container at a given point in time. The Spacemanager points to the Spacemanager Internal Pool, which in turn contains a pointer to the Bitmap.

Further, the first Container Superblock stores a reference to an Object Map (OMAP) B-tree that points to its OMAP Root Node. This B-tree contains OMAP Entries which link object IDs to block offsets.

Most importantly, the Container Superblock stores pointers to the Volume Superblocks of all Volumes in the file system (which are similar to old-fashioned partitions, but do not have hard boundaries and instead all share the entire storage space).

Each Volume Superblock points to the Extentref Tree, its own OMAP, and a Root Directory Node for the particular volume.

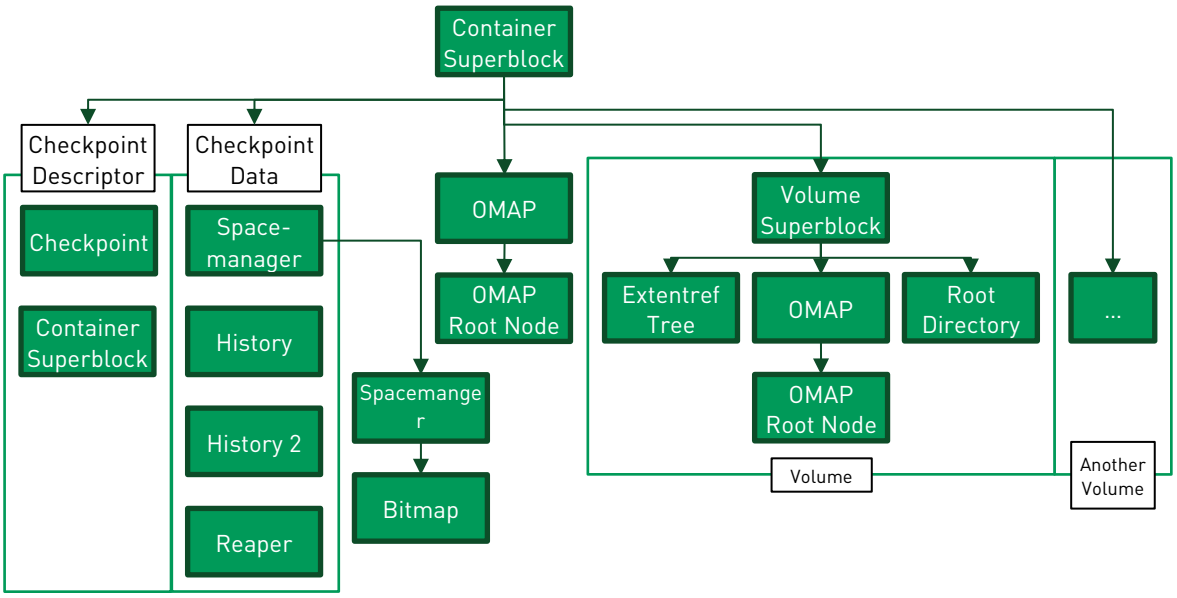


Figure 3: APFS composition

In the appendix of this whitepaper, we attach a reference sheet with the most important structures.

4 Summary and Conclusion

In this paper, we summarized the current state of knowledge about the internals of the APFS file system and try to close some of the prior existing gaps by providing additional insights from our analysis.

4.1 Limitations

While the listed structures in this paper are sufficient to parse generated disk images, assumptions about the file systems where made. Firstly, other versions of APFS where not addressed in this paper and might require adaption in the specification. Furthermore, there are various specific features of APFS, for which further research has to be done and that have not been in the focus of this work:

- Encryption and compression
- Extra structures for Fusion Drives
- Snapshots
- Sparse files
- Hardlinks and softlinks
- APFS file systems that stem from conversion of HFS+
- Corrupted images (e.g. when not unmounted properly)

4.2 Outlook Future Work

The limitations listed in Section 4.1 need improvements. Especially encryption, which is enabled by default on system partitions, should be analysed in depth. Further, we are currently developing a tool for forensic file recovery, which we are going to publish soon.

4.3 Conclusion

In this work, we were able to close some of the gaps regarding the understanding of the internal structures and workings of the APFS file system, and also updated existing information according to the most recent version of APFS.

5 Literaturverzeichnis

Apple Inc. (2004). Technical note tn1150: Hfs plus volume format. *https://developer.apple.com/legacy/library/technotes/tn/tn1150.html (last visited: 2018-01-10).*

Apple Inc. (2017). Apple file system guide. *https://developer.apple.com/library/content/documentation/FileManagement/Conceptual/APFS_Guide/Introduction/Introduction.html (last visited: 2018-01-10).*

Apple Inc. (2017). Apple file system guide: Frequently asked questions. *https://developer.apple.com/library/content/documentation/FileManagement/Conceptual/APFS_Guide/FAQ/ FAQ.html (last visited: 2018-01-10).*

Carrier, B. (2005). *File system forensic analysis* (3 ed.). Addison-Wesley Reading.

Carrier, B. (2010). *The sleuth kit (TSK)*. Retrieved 10 14, 2017, from http://www.sleuthkit.org/sleuthkit/

Craiger, P. (2005). *Advances in Digital Forensics - Recovering digital evidence from Linux systems.* Springer.

Hansen, K., & Toolan, F. (2017). Decoding the apfs file system. *Digital Investigation 22*, (pp. 107–132).

Kodis, J. (1992). Fletcher's checksum. *Dr. Dobb's J. 17 (5)*, (pp. 32–38).

APFS Reference Sheet

By Jonas Plum and Andreas Dewald

Object

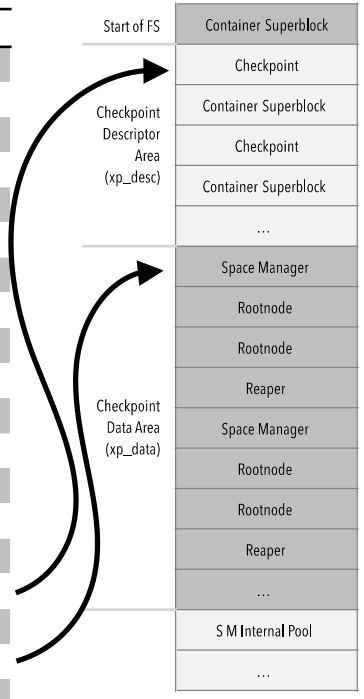
pos	size	type	id
0	8	u8le	cksum
8	8	u8le	oid
16	8	u8le	xid
24	2	u2le	type
26	2	u2le	flags
28	2	u2le	subtype
30	2	u2le	padding

Types:
- **0x1** Container S.block
- **0x2** Rootnode
- **0x3** Node
- **0x5** Space Manager
- **0x7** S. M. Internal Pool
- **0xB** B-Tree
- **0xC** Checkpoint
- **0xD** Volume S.block
- **0x11** Reaper

Subtypes:
- **0x0** No Subtype
- **0x9** History
- **0xB** Location
- **0xE** Files
- **0xF** Extents
- **0x10** Unknown

Container Superblock

pos	size	typ	id
0	4	str	magic 'NXSB'
4	4	u4le	block_size
8	8	u8le	block_count
16	8	u8le	features
24	8	u8le	read_only_compatible_featur
32	8	u8le	incompatible_features
40	16		uuid
56	8	u8le	next_oid
64	8	u8le	next_xid
72	4	u4le	xp_desc_blocks
76	4	u4le	xp_data_blocks
80	8	u4le	xp_desc_base
88	8	u4le	xp_data_base
96	4	u4le	xp_desc_len
100	4	u4le	xp_data_len
104	4	u4le	xp_desc_index
108	4	u4le	xp_desc_index_len
112	4	u4le	xp_data_index
116	4	u4le	xp_data_index_len
120	8	u8le	spaceman_oid
128	8	u8le	omap_oid
136	8	u8le	reaper_oid
152	4	u4le	max_file_systems
160	8	u8le	fs_oid



Ressources:
- Enhanced APFS analysis: Whitepaper by J. Plum and A. Dewald
- Decoding the APFS file system: Paper by K. H.Hansen and F. Toolan in Digital Investigation
- Apple File System Guide: Official documentation on APFS

Last updated: 2018-03-27

Volume Superblock

pos	size	typ	id
0	4	str	magic 'APSB'
4	4	u4le	fs_index
24	4	u4le	features
40	8	u4le	fs_reserve_block_count
48	8	u4le	fs_quota_block_count
56	8	u4le	fs_alloc_count
96	8	u8le	omap_oid
104	8	u8le	root_tree_oid
112	8	u8le	extentref_tree_oid
120	8	u8le	snap_meta_tree_oid
144	8	u8le	next_doc_id
152	8	u8le	num_files
160	8	u8le	num_directories
168	8	u8le	num_symlinks
176	8	u8le	num_other_fsobjects
184	8	u8le	num_snapshots
208	16		vol_uuid
224	8	u8le	last_mod_time
232	8	u8le	formatted_by.last_xid
264	32	str	formatted_by.id
272	8	u8le	formatted_by.timestamp
280	8	u8le	modified_by.last_xid
288	32	str	modified_by.id
320	8	u8le	modified_by.timestamp
672	…	str	volname

B-Tree

pos	size	typ	id
0	8	u8le	btree_type
16	8	u8le	root

Space Manager

pos	size	typ	id
0	4	u4le	block_size
4	4	u4le	blocks_per_chunk
8	4	u4le	chunks_per_cib
12	4	u4le	cibs_per_cab
16	4	u4le	block_count
20	4	u4le	chunk_count
24	4	u4le	cib_count
28	4	u4le	cab_count
32	4	u4le	entry_count
40	8	u8le	free_count
48	4	u4le	entries_offset
144	8	u8le	prev_sm_internal_pool_block
…	…	u8le	spaceman_internal_pool_blocks

Space Man. Internal Pool

pos	size	type	id
4	4	u4le	entry_count
8	…	SMInternalPoolEntry	entries

Space Manager Internal Pool Entry

pos	size	typ	id
0	8	u8le	bitmap_block_xid
16	4	u4le	bm_block_count
20	4	u4le	bitmap_free_blocks
24	8	u8le	bitmap_block

APFS is Little Endian
Timestamps are nanoseconds starting 1970-01-01

Rootnode & Node

pos	size	type	id
0	2	u2le	node_type
2	2	u2le	level
4	4	u4le	entry_count
10	2	u2le	keys_offset
12	2	u2le	keys_length
14	2	u2le	data_offset
16	8	u8le	meta_entry
24	…	EntryHead	entry_heads
…	…	EntryKey	entry_keys
…	…	EntryValue	entry_values

Entry Head

pos	size	typ	id
0	2	s2le	key_offset
2	2	u2le	key_size
4	2	s2le	val_offset
6	2	u2le	val_size

Entry Keys

pos	size	typ	id
0	8	u8le	kind & obj_id
8	8	u8le	xid
16	…	…	depends on kind

Entry Values

Kinds:
- **0x0** omap
- **0x2** lookup
- **0x3** inode
- **0x4** xattr
- **0x5** sibling
- **0x6** extent_status
- **0x8** extent
- **0x9** drec

Pointer Value (when node_type is 2)

pos	size	typ	id
0	8	u8le	pointer

xattr value

pos	size	typ	id
0	2	u2le	xattr_obj_id
2	2	u2le	xdata_len
4	…		dstream

omap value

pos	size	typ	id
0	4	u4le	paddr
4	4	u4le	size
8	8	u8le	obj_id

Extent Value

pos	size	typ	id
0	4	u4le	paddr
4	4	u4le	size
8	8	u8le	obj_id

Inode Value

pos	size	type	id
0	8	u8le	parent_id
8	8	u8le	file_id
16	8	u8le	creation_timestamp
24	8	u8le	modified_timestamp
32	8	u8le	changed_timestamp
40	8	u8le	accessed_timestamp
48	8	u8le	flags
56	4	u4le	nchildren_or_nlink
68	4	u4le	bsd_flags
72	4	u4le	owner_id
76	4	u4le	group_id
80	2	u2le	mode
92	2	u2le	xf_num_ext
94	2	u2le	xf_used_data
96	…	xf_hea	xf_header
…	…	xf_field	xfields

Extended Field Header (xf_header)

pos	size	type	id
0	2	u2le	type
2	2	u2le	length

Extended Field Types
- **0x204** name (string)
- **0x2008** size (u8le)