Revisiting Data Hiding Techniques for Apple File System

Thomas Göbel, Jan Türr and Harald Baier

da/sec - Biometrics and Internet Security Research Group,
Hochschule Darmstadt, Germany
{thomas.goebel,harald.baier}@h-da.de
jan.tuerr@stud.h-da.de

ABSTRACT

Data hiding is an important part of anti-forensic research since the continuous development of operating systems, file systems and other software may close some previously known vulnerabilities but will often inadvertently create new ones. Many of the currently used file systems such as FAT, NTFS or ext4 have been thoroughly analysed. There are quite a few theoretical approaches and also some practical tools that help us to hide data in the existing file systems in different ways. For the Apple File System (APFS), the new standard file system for all Apple devices, only part of the previous work is transferable. There are only a few published forensic analyses of APFS so far and some forensic tools like the Sleuthkit have at least partially adapted APFS functionality. However, anti-forensic techniques specific to APFS have not yet been explored.

This paper aims to introduce APFS and some of its noncritical areas which can be exploited to hide data. A recently published modular anti-forensics framework called *fishy* allows the implementation of modules containing a file system interface and corresponding data hiding techniques. After a short theoretical introduction to the framework, we present, as a practical part of this work, specific data hiding techniques for APFS which are implemented in a separate module for *fishy*. Finally, the newly found techniques are evaluated, e.g., on the basis of their detectability, stability and capacity.

CCS CONCEPTS

• **Applied computing → Computer forensics**; **Evidence collection, storage and analysis**.

KEYWORDS

File system forensics, Anti-Forensics, Data hiding, Apple File system, APFS, fishy

ACM Reference Format:
Thomas Göbel, Jan Türr and Harald Baier. 2019. Revisiting Data Hiding Techniques for Apple File System. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES 2019) (ARES '19), August 26–29, 2019, Canterbury, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3339252.3340524

1 INTRODUCTION

The rise of connectivity and digitisation of personal and work spaces brings with it an increased threat of cyber attacks. An important part of understanding the growing threat is the early discovery of vulnerabilities, attack methods as well as potential anti-forensic techniques used to stop or slow down the forensic process. Of the four elementary types of anti-forensics described by Ryan Harris [9], data hiding is of special interest as it does not destroy or create any additional data [9] and simply exploits already existing weaknesses. There are many forms of data hiding like abusing file formats, encryption[1] or hiding data within other files, like splitting audio and video tracks to hide coded messages [14]. According to the anti-forensic taxonomy of Conlan et al. [2], our paper focuses on *file system manipulation*. This particular part of data hiding uses specific data structures and potential vulnerabilities of a file system to hide data. While there are some hiding places that are present in almost every file system, such as the use of file slack which is described by Stephen P. Larson [11] as well as Knut Eckstein and Marko Jahnke [4], most of them are specific to a particular file system.

With the help of the recently released modular open-source anti-forensic framework *fishy*[2], multiple data hiding techniques can be applied, whether they are unique to one file system or not. The file systems implemented so far (FAT12, FAT16, FAT32, ext4 and NTFS) consist of several data hiding techniques and an interface to interpret the respective file system. A new module developed for *fishy* adds both compatibility with some existing techniques and new techniques specific to the pooled-storage file system *APFS*.

1.1 Motivation

The motivation for this paper and the newly developed *fishy* module is based on the following two points:

- Although they play an important role in forensic research, many tools that allow in-depth investigation of data hiding methods are either undocumented, meaning that both the source code and the original author is unknown, or they focus on a single file system, making adjustments and modifications almost impossible.
- Apple products play an important role in private households today and are also gaining in importance for work environments. Because Apple systems are not as open and accessible to third parties as most alternatives, the specific risk of a

[1]Some researchers do not consider encryption as a form of data hiding as it does not hide the existence of the data and simply tries to limit access to the data [12].
[2]https://github.com/dasec/fishy, last visited 24.04.2019

targeted attack increases [1]. With the introduction of Apple's new file system in mid-2017, which is used by both MacOS and iOS operating systems, it is now possible to find vulnerabilities before they are abused in malicious ways.

This paper therefore gives an overview of all methods found so far to hide data within APFS. Additionally, it presents an open and well documented module adding APFS capabilities including the previously mentioned hiding techniques to the open source framework *fishy*.

1.2 Contribution

This paper first gives a brief introduction to both the APFS file system and the anti-forensic framework *fishy*. In order to understand the hiding technique approaches it is important to first understand the new APFS-specific data structures, which is why our description covers the most important data structures and functions of APFS. Furthermore, we give a brief explanation of the modular structure of *fishy* and the structure of the newly developed APFS module itself. An in-depth explanation of the framework can be found on the GitHub address mentioned above or in the corresponding paper by Göbel and Baier [6].

Since this paper focuses on the discovery and presentation of potential APFS-specific hiding techniques, the second part of the paper describes in detail the vulnerabilities of APFS that these techniques exploit. We thereby present the exact functionality of the five hiding techniques already implemented. Finally, we evaluate the functionality of the new hiding techniques by taking a closer look at the following three evaluation metrics: (i) Detectability, (ii) Stability, and (iii) Capacity.

1.3 Outline

The rest of the paper is organised as follows: Section 2 outlines known forensic analyses of APFS as well as forensic tools that support the new file system. Section 3 gives a brief overview of APFS as well as an introduction to its relevant data structures and functions. Section 4 describes in detail the newly found hiding techniques. This Section also gives a short description of *fishy*, the framework used to implement these techniques. In Section 5, we then evaluate the implemented hiding techniques and rate them based on the evaluation metrics *Detectability*, *Capacity* and *Stability*. Section 6 and Section 7 conclude the paper and discuss potential future work.

2 RELATED WORK

The general idea of how this paper is approaching anti-forensic research is based on knowledge gained by other researchers such as Grugq [7] or Ryan Harris [9] who have already started to collect and evaluate anti-forensic methods and principles for different systems in 2002 and 2006 respectively.

To the best of our knowledge, no current research on APFS-specific data hiding techniques has yet been published. However, there are a few in-depth forensic analyses as well as a forensic tool used to recover data from an APFS partition. The first of them comes from Kurt Hansen and Fergus Toolan [8], who published their analysis of APFS in September 2017. Their analysis is based on the sparse information Apple published about APFS at this time[3] as well as a manual examination of the file system.

Building on the aforementioned paper, Jonas Plum and Andreas Dewald published a further analysis of the Apple file system [3]. Unlike their predecessors Hansen and Toolan, they generally use the nomenclature defined by Apple.

An additional, independent forensic analysis was done in 2017 by Laura Pfeiffer [15], achieving results similar to Hansen and Toolan.

Plum and Dewald added another paper to their previous work describing different ways to recover data from an APFS image [17]. This paper focuses on the process of parsing the Apple file system, which is needed to recover as many current and deleted files as possible, and compared it to more universal approaches like file carving. They also developed an open source tool called *afro* (*APFS File Recovery Options*)[4].

In September 2018 Apple released the first version[5] of their official APFS reference [10], confirming many of the previously mentioned papers' findings and also adding previously unknown information.

The support and creation of forensic tools besides the already mentioned *afro* tool has also grown regarding drivers like *fuse*[6] and *Paragon*[7], making APFS disk images mountable in Linux and Windows environments, respectively. There are also some forensic tools freely available yet, like the *iBored*[8] hex editor that have at least partial APFS support. The popular Sleuthkit framework falls into the same category, having recently published a development branch [18], which provides updated APFS-compatible versions of some of its core features, such as the *fsstat* command.

In addition to the forensic tools, the already mentioned modular anti-forensic framework called *fishy* exists, that enables implementation and testing of data hiding techniques on multiple file systems [6]. For the purpose of this paper, the framework is expanded by a module containing an APFS interface and the hiding techniques mentioned in Section 4.

3 INTRODUCTION TO APPLE FILE SYSTEM

The structure of APFS is a mix of known ideas and functionalities. Unlike its predecessor HFS+ it is not a journaling file system. To ensure secure file system transactions, APFS employs both atomic operations and a checkpoint system [8]. Additionally, APFS is a double layered file system. The external layer, the *Container*, is a managing entity, responsible for the entire file system. *Volumes* are the internal layer and manage files and directories with the help of B-trees [10]. While an implementation of APFS only has one container, the amount of possible volumes depends on the size of the container[9]. An interesting note is the role of the volumes. While they act similar to traditional partitions in some ways, they are completely different in others. The most glaring difference being

[3]https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/APFS_Guide/Introduction/Introduction.html, last visited 29.04.2019.
[4]https://github.com/cugu/afro. *Last visited 2019-04-05.*
[5]There have been multiple updates as of April of 2019.
[6]https://github.com/sgan81/apfs-fuse. *Last visited 16.04.2019.*
[7]https://www.paragon-software.com/home/apfs-windows/. *Last visited 16.04.2019.*
[8]http://files.tempel.org/iBored/ by Thomas Tempelmann. *Last visited 16.04.2019.*
[9]The upper limit is 100 volumes.

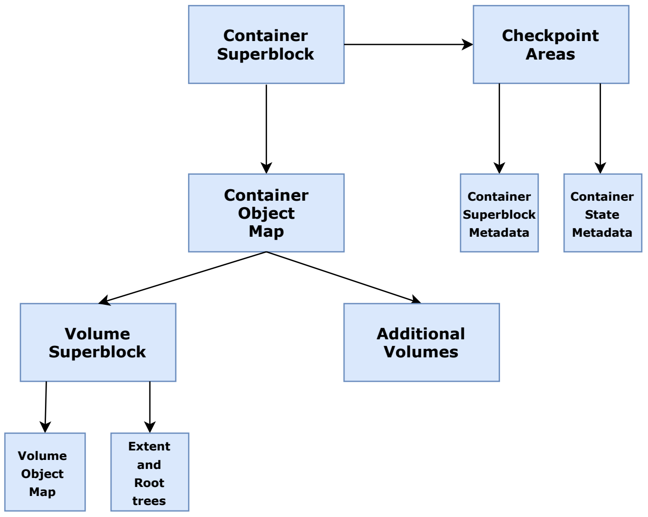


Figure 1: Simplified structure of an APFS container, adapted from Plum [17].

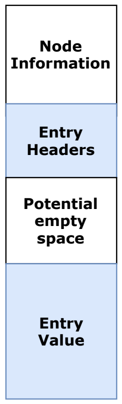


Figure 2: Simplified structure of an APFS node.

that, barring potential reserved blocks or quotas, they have a dynamic size they share with all other volumes as blocks are allocated on a container level [3]. This feature is called *Space Sharing*.

An interesting challenge to overcome for data hiding purposes is the way APFS references different structures. APFS distinguishes between three different object types: (i) *Ephemeral*, (ii) *Virtual*, and (iii) *Physical*. While virtual objects are always stored on disk and ephemeral objects are generally stored in memory and only written to disk as part of a checkpoint [10], they are referenced in the same way. Their object ID, which is stored in a universally used object header, is usually determined by the container superblock field *nx_next_oid*. B-trees are used to map these object IDs to block addresses. Physical objects are always written to disk and their object IDs are their block addresses.

Figure 1 is a simple representation of the structure of an APFS container. The data structures relevant to the hiding techniques shown in Section 4 are the container and volume superblocks as well as their respective object maps and the nodes contained in the volume B-trees. Furthermore, the checkpoint areas, specifically the *Container Superblock Metadata*, are of interest as well.

Another important fact of APFS is that all file system structures, with the exception of the allocation bitmap, are stored as objects and given object headers. These 32 byte sized headers contain a checksum among other information. The checksum is a 64-bit version of *Fletcher's checksum* with the input being the entire block without the checksum field [3].

There are two different versions of superblocks, one for each layer. A copy of the current version of the container superblock is usually found at block 0 of the container. It contains much of the overarching information about the container, such as references to its object map, a list of volumes and elementary information like the system's block size. The volume superblock acts as a similar managing instance for its respective volume, containing information about the size, name and contents of the volume as well as references to extent trees and the volume object map.

Nodes are some of the most important data structures in APFS, as they contain most of the information needed to interpret the file system. The node structure can be seen in Figure 2. In addition to general information about the node, such as its position in a B-tree, it contains a table of contents that specifies the location of the nodes' entries [10]. A node's entries vary drastically depending on the node's application within the file system. The entries are split in key and value pairs, with the key determining the entry type. The key and value areas are usually separated by free space, so new entry keys and values can be added. Among the most important entry types for this paper is type 3, which represents the inode and whose structure can be seen in Table 1.

Checkpoints are a core feature of APFS. They represent past states of the container by saving important structures and data. The container superblock, memory states at the time of the checkpoint and allocation structures, such as the space manager, are stored in the *Checkpoint Areas* depicted in Figure 1. Object maps and volume superblocks are stored in different places in the container and referenced by the corresponding structures. This function has positive impacts upon data hiding, like potentially bigger capacities for techniques that use these structures. There also is a negative impact since some crucial information about the processes the checkpoint system uses are still unknown. It is not known how many checkpoints a container can have[10] or when exactly a new checkpoint is created. More importantly, it is not known how old structures are overwritten. If entire blocks are erased before a new checkpoint is created, techniques, that use the slack space of these structures, suffer from significant stability deficits.

4 HIDING TECHNIQUES

All presented hiding techniques, unless otherwise mentioned, are developed and integrated into the anti-forensic data hiding framework *fishy*, which therefore gets an additional APFS interface.

fishy is a python-based modular framework developed to be an open solution for data hiding research. So far, four major file system modules (NTFS, FAT, ext4, APFS) are available in the framework.

[10]Initial research using images of different sizes indicate that it is tied to the size of the container.

The modules contain specific hiding techniques as well as file system interfaces. These interfaces use self-programmed parsers or other open source tools such as Construct[11] to interpret file system images. The framework already provides multiple wrappers for its hiding techniques as well as a command line interface for interaction with the user. In particular, all techniques are not only able to write data, they can also read and erase[12] the data that was previously written into the file system. In order to find the hidden data later on, every write method provides metadata relevant to the technique, such as the size or offset of the hidden data, which is then turned into an encrypted JSON file. If needed, the techniques also calculate the structure's checksum after data is written or erased. To calculate the checksum after modifying an APFS structure the algorithm developed by Jonas Plum [16] is used.

Most of the hiding techniques presented in the following Section use free, reserved or otherwise unimportant space found in the file systems' inodes. These areas are highlighted by the bold text in Table 1. The only exception is the *Superblock Slack* technique which uses both types of superblocks and their respective object maps.

Table 1: Contents of an inode. Bold fields indicate the use of a particular field for one of the hiding techniques.

Position	Size	Name
0	8	parent_id
8	8	private_id
16	8	**create_time**
24	8	**mod_time**
32	8	**change_time**
40	8	**access_time**
48	8	internal_flags
56	4	nchildren_or_nlink
60	4	def_protection_class
64	4	**write_gen_counter**
68	4	bsd_flags
72	4	owner
76	4	group
80	2	mode
82	2	pad1
84	8	**pad2**
92	variable	**extended_fields**

4.1 Superblock Slack

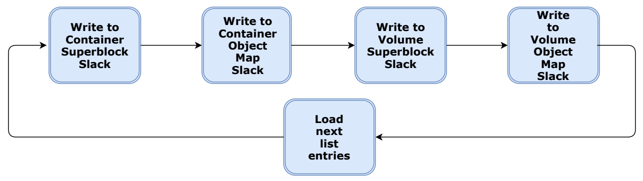

Write to Container Superblock Slack
Write to Container Object Map Slack
Write to Volume Superblock Slack
Write to Volume Object Map Slack
Load next list entries

Figure 3: Visual representation of superblock slack target choice.

Although this technique is not only usable in APFS[13], the implementation for APFS is somewhat unique due to its usage of two different superblock types and the adjacent object maps as well as the effects of the checkpoint feature discussed in Section 3.

The block addresses of all relevant structures need to be collected before any data can be hidden. For that purpose an external class called *Checkpoints* gathers all four used structures in separate lists. This happens in four steps. First, the container superblock is added to the respective list. Second, the address of the attached object map is extracted and stored in a separate list. Third, using the object map, all current volume superblocks are found and added to a another list. Finally, the object maps attached to each volume superblock are stored in a fourth list. To expand the capacity of this technique, older versions of these structures are used as well by following the previously mentioned steps after locating all previous container superblocks in the *Checkpoint Areas* seen in Figure 1. The information to locate the *Checkpoint Areas* is found in a dedicated section in the container superblock. All lists are sorted by the objects' transaction IDs, with the newest object located at the beginning of each list.

The hiding technique itself requests all four lists before starting the hiding process. The write method itself calls four submethods, one for each used structure type. The main method transmits the corresponding address as well as the stream containing the data it is supposed to hide to each submethod which in turn writes to the respective structure's slack space and returns the size of the written data. After successful completion of a submethod, the checksum for the used structure is recalculated. This process is visualised in Figure 3 and Figure 4.

To recover or erase the hidden data, the method creates a metadata file containing four lists, one for each structure type filled with the offsets of all used structures of that type as well as a field containing the full size of the hidden data.

Due to the unknown nature of the checkpoint write system[14] during development, a method that only uses older versions of the structures when a higher capacity is needed was chosen. As seen in Figure 3, the technique writes into the first entry of each list before choosing an older and potentially less stable hiding space.

4.2 Write-Gen-Counter

Besides the nodes' version numbers in the universal object headers, every inode entry in these nodes has its own versioning method.

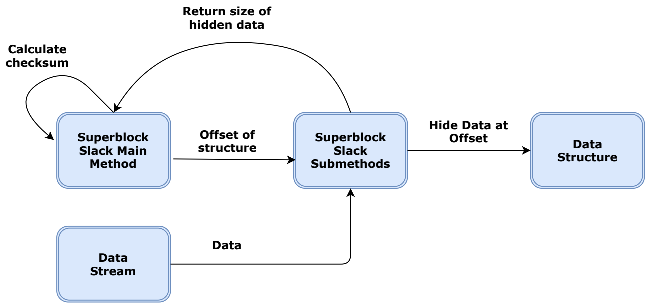


Figure 4: Visual representation of superblock slack write algorithm.

[11] https://construct.readthedocs.io/en/latest/, *Last visited 24.04.2019.*
[12] The hidden data is overwritten with zeroes.
[13] For example, the ext4 module implemented in fishy already has a technique exploiting the slack created by superblock data structures [6].
[14] It is unknown when exactly new checkpoints are created and how old checkpoints are overwritten.

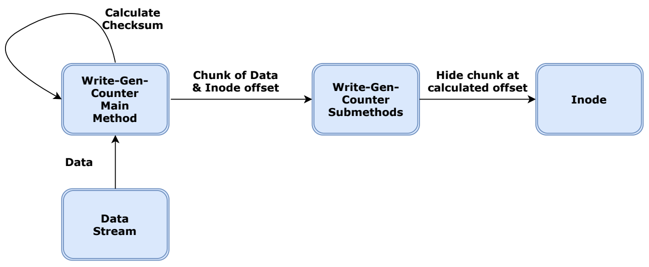

Calculate Checksum
Write-Gen-Counter Main Method
Chunk of Data & Inode offset
Write-Gen-Counter Submethods
Hide chunk at calculated offset
Inode
Data
Data Stream

Figure 5: Visual representation of Write-Gen-Counter write algorithm.

The inode versioning is being managed by *write_generation_counter*, a 4 byte counter that is increased whenever the inode or its data is modified. When reaching its limit the counter restarts from 0 [10], making it a possibly undetectable hiding space since there are no disallowed values within the fields 4 byte range.

Similar to the *Superblock Slack* technique described before, the first step this and all subsequent techniques must perform is parsing the target image and extracting the addresses of all nodes containing inode entries as well as the exact offset of each inode entry. This is done by an auxiliary class called *InodeTable*. This class gathers all volume object maps and parses the referenced B-tree and its nodes for inode entries. Before returning a tuple list containing the node and inode addresses, the root nodes are removed since they only contain references to inodes.

The process of hiding the data itself is quite simple and can be seen in Figure 5. First, the stream of data that is supposed to be hidden is split into chunks of 4 byte. An inode address is extracted from the list received from *InodeTable* and given to a submethod alongside one of the 4 byte chunks. The submethod then calculates the exact offset of the hiding space and writes the contents of the chunk to this address. After every written chunk, the respective node's checksum is newly calculated.

The metadata saved to access the hidden data at a later time consists of lists of the used inode addresses and node addresses.

4.3 Timestamp Hiding

Similar to other modern file systems (e.g., btrfs, ext4, NTFS), APFS offers 64-bit nanosecond timestamps. Exemplary techniques exploiting the nanoseconds part of timestamp fields already exist for ext4 [5] and NTFS [13], which are used as a basis for the APFS implementation. Although the currently implemented version of this technique only uses the highlighted timestamps in Table 1, it should be mentioned that APFS also has other structures, such as volume superblocks, which contain usable timestamps.

Again, the address is extracted from the list returned by the *InodeTable* class. This address, the data stream and a number ranging from 0 to 3 is given to a submethod that calculates the exact offset and writes data to the first 4 bytes of the timestamp. The number transmitted is an indicator of what timestamp the data is written to. If the number exceeds 3, it is set to 0 and a new inode is targeted. This means that the current implementation of this technique writes to all four timestamps present in an inode. By default, our implementation not only ensures that all timestamps are written to, but

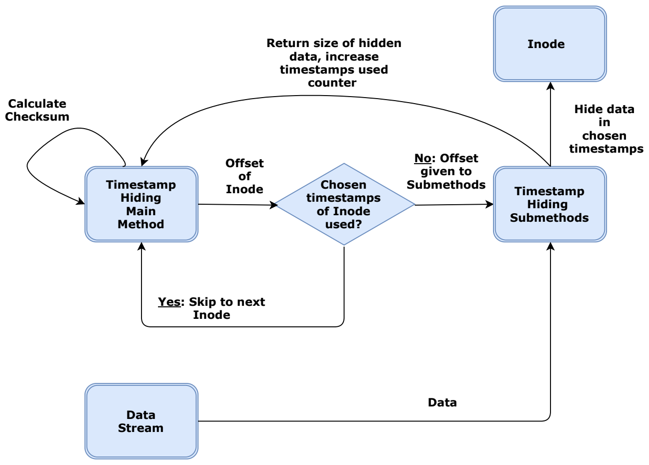

Return size of hidden data, increase timestamps used counter
Inode
Calculate Checksum
Hide data in chosen timestamps
Offset of Inode
No: Offset given to Submethods
Timestamp Hiding Main Method
Chosen timestamps of Inode used?
Timestamp Hiding Submethods
Yes: Skip to next Inode
Data Stream
Data

Figure 6: Visual representation of timestamp hiding write algorithm.

also makes it possible to write to a specific timestamp with minimal change. The complete hiding process is visualised in Figure 6.

In order to recover hidden data, the metadata file of this technique contains used node and inode addresses as well as the size of the hidden data.

4.4 Extended Field Padding

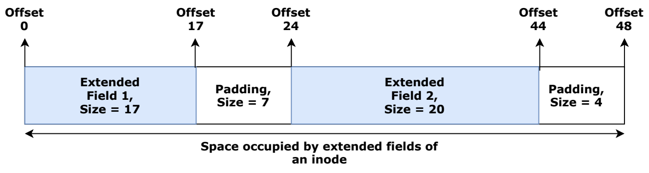

Offset 0
Offset 17
Offset 24
Offset 44
Offset 48
Extended Field 1, Size = 17
Padding, Size = 7
Extended Field 2, Size = 20
Padding, Size = 4
Space occupied by extended fields of an inode

Figure 7: Visual representation of extended field padding generation.

The inode and directory record entry types have a unique feature called *Extended Fields* that follow the regular entry contents. Usually, these fields' sizes are a multiple of 8 bytes. However, some of these fields have variable sizes based on their content. If a field is not the right size, it is enlarged to the next possible multiple of 8 bytes by adding a padding field as seen in Figure 7. The programmed version of this technique only writes to extended fields found in inodes since extended fields seem to be more consistent there. Directory records rarely use this feature in all tested images.

Like the previous hiding techniques, this method also extracts the inode addresses from *InodeTable*. The difference here is that the addresses of the hiding spaces are not always at a fixed offset inside the inode. Instead, there are three additional steps. First, the table of contents for the extended field of an inode, as seen in Table 2, has to be interpreted and the number of extended fields has to be extracted. Second, the sizes of the extended fields have to be extracted from the list of extended field headers as seen in Table 3 following *xf_blob*[15]. Third, the sizes have to be used to determine if there is any padding among this set of extended fields and if so, the size of the padding has to be calculated[16]. Once all paddings are calculated, the data stream and a tuple list containing the padding addresses, the padding sizes as well as the inode addresses are given to a submethod. The submethod writes to the different padding locations and returns a similar list with used addresses, the size of the padding at each used address and the corresponding node addresses. In a final step, the correct checksums are calculated.

In this case, the created metadata contains lists of padding offsets, padding sizes and node addresses.

Table 2: The table of contents for extended fields.

Position	Size	Name
0	2	number_of_xfields
2	2	size_of_xfields
4	number_of_xfields*4	xfield headers

Table 3: The structure of extended field headers.

Position	Size	Name
0	1	xfield_type
1	1	xfield_flags
2	2	xfield_size

4.5 Inode Padding

Each inode entry of a node has two padding fields of 2 and 8 bytes each, which separate the entry value contents from the dynamic extended fields[17]. This technique aims to exploit the *pad2* padding field[18].

Just like other techniques, which hide data in inodes, this technique refers to the *InodeTable* class to get a list of all inode addresses. The data stream is then split into chunks of 8 byte each, which in turn is given to a submethod alongside an inode address. This submethod calculates the exact offset of the padding field and writes the content of the chunk to it. After the submethod is finished, the inode flags are adjusted[19] and the corresponding node's checksum is recalculated. This technique works almost exactly like the *Write-Gen-Counter* technique described above, with the only differences in the calculated offsets and the size of the chunks and the adjusting of the inode flags.

The metadata generated by this technique contains two lists, one with the used inode addresses and the other with the corresponding node addresses.

[15] *xf_blob* is the official name of the table of contents managing the extended fields as seen in Table 2

[16] *(xfield_size + (8 - xfield_size % 8))-xfield_size* is the formula used to calculate the padding space.

[17] *pad1* and *pad2* in Table 1.

[18] Reverse engineering the file system check has revealed that *pad1* can not be any value other than zero.

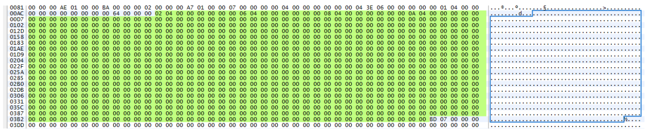
[19] To write in *pad2*, the *has_uncompressed_size flag* has to be set. The value of this flag is *0x40000*. This flag is not yet referenced in the official APFS reference and has been determined by reverse engineering the *fsck_apfs* binary.

4.6 Further Potential Hiding Techniques

In addition to the techniques already mentioned, some additional potential hiding places were found during this research. Some of them require further information about structures not discussed in this paper.

Two more general hiding techniques, that can be applied to many file systems and which could potentially be applied here as well, are the use of file slack and the allocation of additional blocks to an existing file. A file slack technique would require parsing the extent trees in volumes and a way to calculate the free space in a block. To implement a block allocation technique, the module would first have to be expanded by the Space Manager and the rest of the Bitmap structures. Both of these techniques would potentially have a high capacity but low stability due to the changing nature of a file's content and its size.

Furthermore, there are two fields within the *Container Superblock*[20] (4 and 8 bytes in size, respectively) that could be used to hide data. These fields are reserved for testing and debugging purposes and are set to 0 by default. Although this does not provide much capacity by itself, combining this method with another technique to potentially hide more sensible data may prove effective. In addition to the low capacity, the stability of this technique would also be low since the container superblocks are regularly overwritten on a running Apple system.



```
0081 00 00 00 AE 01 00 00 BA 00 00 00 02 00 00 00 A7 01 00 00 07 00 00 00 00 04 00 00 00 00 00 00 04 3E 06 00 00 00 00 00 01 04 00 00  ...®...º.......§................>..........
00AC 00 00 00 00 00 00 00 00 64 00 00 00 02 04 00 00 00 00 00 00 06 04 00 00 00 00 00 00 08 04 00 00 00 00 00 00 0A 04 00 00 00 00 00  ........d..................................
00D7 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
0102 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
012D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
0158 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
0183 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
01AE 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
01D9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
0204 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
022F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
025A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
0285 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
02B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
02DB 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
0306 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
0331 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
035C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
0387 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
03B2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 BD 07 00 00 00  ......................................½....
03DD 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...........................................
```

Figure 8: Hex view of the empty spaces in the list of volume object IDs.

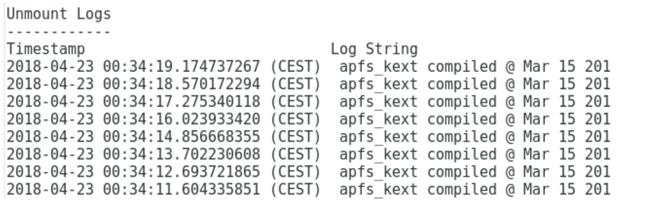
Another potential *Container Superblock*-based hiding technique could be the exploitation of the list of volume superblock object IDs. This list is present in every *Container Superblock*. Its size is not determined by how many volumes the container has, but how many it can potentially have[21]. Since most commercially available macOS-based systems would easily reach the upper limit of 100 volumes and there usually being 4 volumes already set up, this technique would allow for up to 768 bytes of hiding space per container superblock[22]. A visual representation of that space can be seen in Figure 8. However, the stability of this technique would theoretically be low since users can create new volumes at will, which would overwrite hidden data.

Figure 9 shows the unmount logs that are present in every volume superblock. These logs can have up to 8 entries logging the modification of the volume with a fixed ninth entry displaying information about the creation of the volume. The timestamps could be manipulated similar to the technique described in Subsection 4.3 and the strings chronicling the entity responsible for creating or modifying the volume could be overwritten completely, offering up to 36 bytes of space from the timestamps and up to 288 bytes

[20] *nx_test_type* and *nx_test_oid*.

[21] 512 Megabyte of size are needed for one volume.

[22] 96 empty spaces multiplied with the fields size of 8 bytes.



```
Unmount Logs
------------
Timestamp                             Log String
2018-04-23 00:34:19.174737267 (CEST)  apfs_kext compiled @ Mar 15 201
2018-04-23 00:34:18.570172294 (CEST)  apfs_kext compiled @ Mar 15 201
2018-04-23 00:34:17.275340118 (CEST)  apfs_kext compiled @ Mar 15 201
2018-04-23 00:34:16.023933420 (CEST)  apfs_kext compiled @ Mar 15 201
2018-04-23 00:34:14.856668355 (CEST)  apfs_kext compiled @ Mar 15 201
2018-04-23 00:34:13.702230608 (CEST)  apfs_kext compiled @ Mar 15 201
2018-04-23 00:34:12.693721865 (CEST)  apfs_kext compiled @ Mar 15 201
2018-04-23 00:34:11.604335851 (CEST)  apfs_kext compiled @ Mar 15 201
```

Figure 9: Partial output of the Sleuthkit command *fsstat* concerning unmount logs.

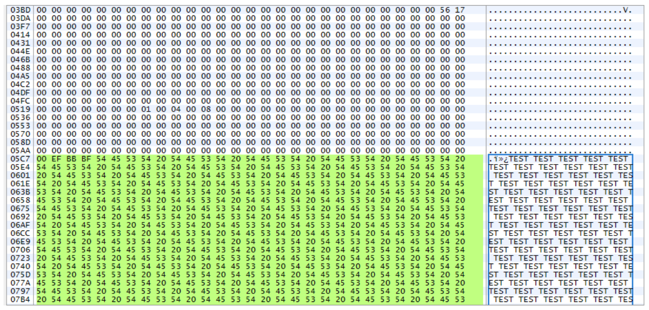
from the string[23] for each volume superblock. The stability of this technique would be rather low since most of the entries can be overwritten in a running system. An additional stability concern would be that the volume superblocks might be completely overwritten due to the checkpoint system.

5 EVALUATION OF PRESENTED HIDING TECHNIQUES

In this Section we evaluate the implemented hiding techniques based on the following three evaluation metrics. As a first metric, we consider the **Detectability** of hidden data. Besides the use of *fsck_apfs*, the grade of detectability is being determined by comparing them to the detectability of similar techniques used on other file systems as well as on the possibility of a manual detection. Our second metric, **Stability**, is influenced by how likely the hidden data can be completely or partially overwritten on a running system. Our third metric, **Capacity**, is determined by the size of the potential hiding space in relation to the size of the used file system image.

For the evaluation process various APFS images of different sizes were used. The specific images can be found and downloaded via a link provided by Plum and Dewald within their paper [17] mentioned above. These images have been modified using *mmcat* so that any non-APFS specific data is no longer included.

5.1 Evaluation of Superblock Slack

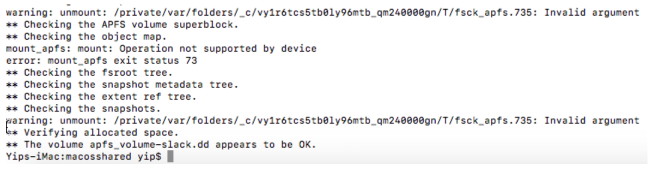


```
03BD 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 56 17 ...........................V.
03DA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
03F7 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
0414 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
0431 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
044E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
046B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
0488 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
04A5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
04C2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
04DF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
04FC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
0519 00 00 00 00 00 00 00 01 00 04 00 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
0536 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
0553 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
0570 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
058D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
05AA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............................
05C7 00 EF BB BF 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 .ï»¿TEST TEST TEST TEST TEST 
05E4 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 TEST TEST TEST TEST TEST TEST
0601 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53  TEST TEST TEST TEST TEST TES
061E 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 T TEST TEST TEST TEST TEST TE
063B 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 ST TEST TEST TEST TEST TEST T
0658 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 EST TEST TEST TEST TEST TEST 
0675 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 TEST TEST TEST TEST TEST TEST
0692 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53  TEST TEST TEST TEST TEST TES
06AF 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 T TEST TEST TEST TEST TEST TE
06CC 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 ST TEST TEST TEST TEST TEST T
06E9 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 EST TEST TEST TEST TEST TEST 
0706 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 TEST TEST TEST TEST TEST TEST
0723 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53  TEST TEST TEST TEST TEST TES
0740 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 T TEST TEST TEST TEST TEST TE
075D 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 ST TEST TEST TEST TEST TEST T
077A 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 EST TEST TEST TEST TEST TEST 
0797 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 TEST TEST TEST TEST TEST TEST
07B4 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53 54 20 54 45 53  TEST TEST TEST TEST TEST TES
```

Figure 10: Hex editor view of an occupied container superblock slack.

While this technique works as intended, what can be seen in Figure 10, a potential improvement would be a different, more refined way for the write method to choose targets, specifically how

[23]9*4 for the timestamps, 9*32 for the strings.



```
warning: unmount: /private/var/folders/_c/vy1r6tcs5tb0ly96mtb_qm240000gn/T/fsck_apfs.735: Invalid argument
** Checking the APFS volume superblock.
** Checking the object map.
mount_apfs: mount: Operation not supported by device
error: mount_apfs exit status 73
** Checking the fsroot tree.
** Checking the snapshot metadata tree.
** Checking the extent ref tree.
** Checking the snapshots.
warning: unmount: /private/var/folders/_c/vy1r6tcs5tb0ly96mtb_qm240000gn/T/fsck_apfs.735: Invalid argument
** Verifying allocated space.
** The volume apfs_volume-slack.dd appears to be OK.
Yips-iMac:macosshared yip$
```

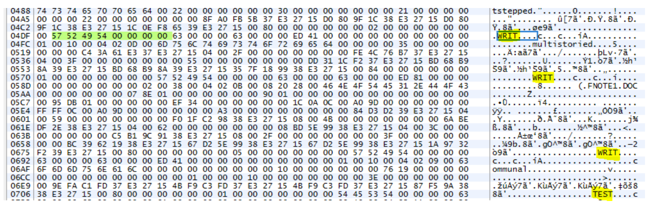
Figure 11: *fsck_apfs* output when checking an image with data hidden using the *Superblock Slack* technique.

it chooses the volume superblocks. The current implementation sorts the volume superblock list simply by version, which means multiple versions of the same volume superblock could be written to before a different volume superblock is used. To avoid this, the technique could remember the object IDs of all used volume superblocks and skip list entries with known object IDs until every unique volume superblock has been accessed once.

(1) **Detectability** - **Medium**: While Apple’s file system check *fsck_apfs* does not find the hidden data as we can see in Figure 11, a manual investigation using the hexdump should be quite easy. The usage of different types of structures may not help in hiding the data itself, but could help to obfuscate the content of the actual hidden data.
(2) **Stability** - **Low-High**: The stability of this technique depends mostly on the unknown impact of the checkpoint write and overwrite systems mentioned in Subsection 4.1. If the system only overwrites the structure without touching the rest of the block, this technique’s stability is **high**. However if the entire block is cleared before writing a new checkpoint, the stability is **low**.
(3) **Capacity** - **High**: The capacity of this technique is proportional to the size of the container. Larger containers allow for more checkpoints, with a standard checkpoint allowing up to 34776 bytes[24] of hiding space. Since there is no clear indicator on how the size of the container affects the size of the checkpoint areas, no definite statement on maximum capacity can be made[25]. An optimal checkpoint would encompass one container superblock, one container object map, four volume superblocks and four volume object maps. However, not every checkpoint is structured in this way. Some checkpoints reference older volume superblocks if their content has not changed since the last checkpoint. This can be demonstrated by comparing the theoretical and actual hiding space in an image. In one of the test images with a size of 5 GB, the theoretical hiding space is 904176 bytes because the image has 26 checkpoints. When removing all duplicates from the four lists introduced in Subsection 4.1, only 506016 bytes remain. In an image with a larger size, there are more checkpoints available[26] resulting in enough space to hide pictures and perhaps even small video or audio files.

[24]*(5*3984)+(4*3060)+2616* being the formula.
[25]Determining the correlation between the size of a container and the size of the checkpoint area could be achieved by using multiple containers of various sizes and comparing the size of their checkpoint areas after performing the same large amount of data operations on all of them.
[26]A tested 100 GB image had 139 checkpoints in total.

5.2 Evaluation of Write-Gen-Counter



```
0488 74 73 74 65 70 70 65 64 00 22 00 00 00 00 00 00 30 00 00 00 00 00 00 00 21 00 00 00 00  tstepped."......0.......!....
04A5 00 00 00 22 00 00 00 00 00 00 00 8F A0 FB 5B 37 E3 27 15 D0 80 9F 1C 38 E3 27 15 D0 80  ..."........ û[7ã'.Ð.Ÿ.8ã'.Ð.
04C2 9F 1C 38 E3 27 15 1C 0E F8 65 39 E3 27 15 00 80 00 00 00 00 00 00 02 00 00 00 00 00 00  Ÿ.8ã'...øe9ã'................
04DF 00 57 52 49 54 00 00 00 00 63 00 00 00 63 00 00 00 ED 41 00 00 00 00 00 00 00 00 00 00  .WRIT....c...c...íA..........
04FC 01 00 10 00 04 02 0D 00 6D 75 6C 74 69 73 74 6F 72 69 65 64 00 00 00 00 35 00 00 00 00  ........multistoried....5....
0519 00 00 00 C4 3A 61 E3 37 E3 27 15 04 00 2F 00 00 00 00 00 00 00 FE 4C 76 B7 37 E3 27 15  ...Ä:aã7ã'.../.......þLv·7ã'.
0536 04 00 3F 00 00 00 00 00 00 00 55 00 00 00 00 00 00 00 DD 31 1C F2 37 E3 27 15 BD 68 B9  ..?.......U.......Ý1.ò7ã'.½h¹
0553 8A 39 E3 27 15 BD 68 B9 8A 39 E3 27 15 35 7F 18 99 38 E3 27 15 00 84 00 00 00 00 00 00  Š9ã'.½h¹Š9ã'.5..™8ã'..„......
0570 01 00 00 00 00 00 00 00 57 52 49 54 00 00 00 00 63 00 00 00 63 00 00 00 ED 81 00 00 00  ........WRIT....c...c...í....
058D 00 00 00 00 00 00 00 02 00 38 00 04 02 0B 00 08 20 28 00 46 4E 4F 54 45 31 2E 44 4F 43  .........8...... (.FNOTE1.DOC
05AA 00 00 00 00 00 00 07 8E 01 00 00 00 00 00 00 90 01 00 00 00 00 00 00 00 00 00 00 00 00  .......Ž.....................
05C7 00 95 DB 01 00 00 00 00 00 EF 34 00 00 00 00 00 00 1C 0A 0C 00 A0 9D 00 00 00 00 00 00  .•Û......ï4.......... .......
05E4 FF FF 0C 00 A0 9D 00 00 00 00 00 00 A3 00 00 00 00 00 00 00 00 84 D3 D2 39 E3 27 15 04  ÿÿ.. .......£........„ÓÒ9ã'..
0601 00 59 00 00 00 00 00 00 00 F0 1F C2 98 38 E3 27 15 08 00 4B 00 00 00 00 00 00 00 6A BE  .Y.......ð.Â˜8ã'...K.......j¾
061E DF 2E 38 E3 27 15 04 00 62 00 00 00 00 00 00 00 08 BD 5E 99 38 E3 27 15 04 00 3C 00 00  ß.8ã'...b........½^™8ã'...<..
063B 00 00 00 00 00 C5 B1 9C 91 38 E3 27 15 08 00 2F 00 00 00 00 00 00 00 3F 00 00 00 00 00  .....Å±œ'8ã'.../.......?.....
0658 00 00 BC 39 62 19 38 E3 27 15 67 D2 5E 99 38 E3 27 15 67 D2 5E 99 38 E3 27 15 1A 97 32  ..¼9b.8ã'.gÒ^™8ã'.gÒ^™8ã'..—2
0675 F2 39 E3 27 15 00 80 00 00 00 00 00 00 05 00 00 00 00 00 00 00 57 52 49 54 00 00 00 00  ò9ã'.................WRIT....
0692 63 00 00 00 63 00 00 00 ED 41 00 00 00 00 00 00 00 00 00 00 01 00 10 00 04 02 09 00 63  c...c...íA..................c
06AF 6F 6D 6D 75 6E 61 6C 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 76 19 00 00 00 00  ommunal................v.....
06CC 00 00 00 00 00 00 00 00 00 00 01 00 00 00 10 00 00 00 00 00 00 00 00 3E 00 00 00 00 00  .......................>.....
06E9 00 9E FA C1 FD 37 E3 27 15 4B F9 C3 FD 37 E3 27 15 4B F9 C3 FD 37 E3 27 15 87 F5 9A 38  .žúÁý7ã'.KùÃý7ã'.KùÃý7ã'.‡õš8
0706 38 E3 27 15 00 80 00 00 00 00 00 00 01 00 00 00 00 00 00 00 54 45 53 54 00 00 00 00 63  8ã'.................TEST....c
```

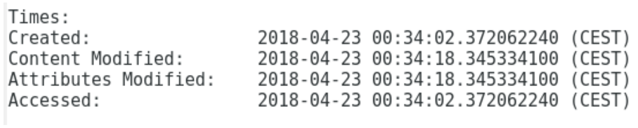
Figure 12: Hex editor view of an inode after hiding data with the *Write-Gen-Counter* technique.

(1) **Detectability** - **Low**: The file system check does not detect any foul play and there seems to be no disallowed value within the 4 byte parameter. This makes a manual investigation incredibly hard, if not impossible. Due to inodes being stored out of order, the reconstruction of hidden data is challenging, as can be seen in Figure 12. Here, the technique was used to hide the string 'WRITEGENTEST' several times. The inode excerpt shows that the data is split and hidden out of sequence.

(2) **Stability** - **Medium**: Since the field is a counter that is increased every time the inode or its corresponding data are modified, it is possible that hidden data can get corrupted. This risk increases proportionally to the size of the hidden data, as more inodes will be needed.

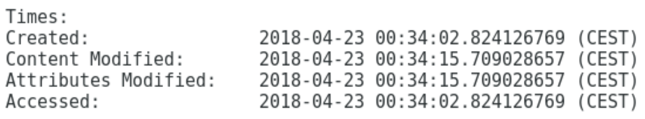
(3) **Capacity** - **Low**: Each inode offers only exactly 4 bytes of hiding space[27] which limits the capacity of this technique severely. The capacity however scales with the amount of files and directories in a container and therefore also depends on the size of a container.

5.3 Evaluation of Timestamp Hiding



```
Times:
Created:                  2018-04-23 00:34:02.372062240 (CEST)
Content Modified:         2018-04-23 00:34:18.345334100 (CEST)
Attributes Modified:      2018-04-23 00:34:18.345334100 (CEST)
Accessed:                 2018-04-23 00:34:02.372062240 (CEST)
```

Figure 13: Output of the Sleuthkit command *istat* before hiding data in the timestamps.



```
Times:
Created:                  2018-04-23 00:34:02.824126769 (CEST)
Content Modified:         2018-04-23 00:34:15.709028657 (CEST)
Attributes Modified:      2018-04-23 00:34:15.709028657 (CEST)
Accessed:                 2018-04-23 00:34:02.824126769 (CEST)
```

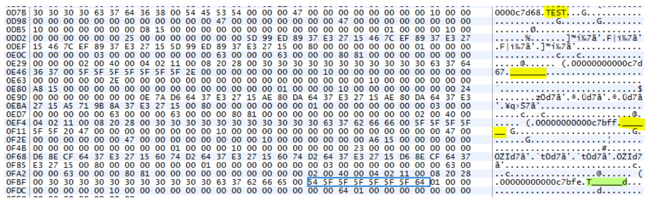
Figure 14: Output of the Sleuthkit command *istat* after hiding data in the timestamps.

(1) **Detectability** - **Low**: The file system check does not detect modified timestamps at all. However, as can be seen in Figure 13 and Figure 14, a minimal change to the seconds part of the timestamp is possible, depending on the last bits of the chunk of the data hidden in the respective timestamp. While this may not have much impact when used against average users, it could attract the attention of a professional investigator. It is also possible to avoid manipulating the seconds by writing slightly smaller chunks[28].

(2) **Stability** - **Medium**: The current version of this technique uses all four inode timestamps, which increases the capacity but decreases the stability, since three of the timestamps (*mod_time, change_time* and *access_time*) have the possibility to change while the system is in use. As explained in Subsection 4.3, this technique can easily be changed to write to a specific set of timestamps.

(3) **Capacity** - **Low**: Although this technique writes to all four timestamps, the capacity remains low, as there are still only 16 bytes of space[29] available in each inode. Just like the previous technique, the capacity of this technique rises proportionally to the number of files in the container. Another way to increase the capacity of this technique would be to add timestamps of other structures, such as the directory records (one timestamp per record) or the volume superblocks (between 1 and 9 timestamps per superblock).

5.4 Evaluation of Extended Field Padding



```
0D7B 30 30 30 30 63 37 64 36 38 00 54 45 53 54 00 00 00 47 00 00 00 00 00 00 00 00 10 00 00  0000c7d68.TEST...G...........
0D98 00 00 00 00 00 00 00 00 00 00 00 00 47 00 00 00 00 00 00 00 47 00 00 00 00 00 00 00 00  ............G.......G........
0DB5 10 00 00 00 00 00 00 D8 15 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 10 00  .......Ø.....................
0DD2 00 00 00 00 00 00 25 00 00 00 00 00 00 00 5D 99 ED 89 37 E3 27 15 46 7C EF 89 37 E3 27  ......%.......]™í‰7ã'.F|ï‰7ã'
0DEF 15 46 7C EF 89 37 E3 27 15 5D 99 ED 89 37 E3 27 15 00 80 00 00 00 00 00 00 01 00 00 00  .F|ï‰7ã'.]™í‰7ã'.............
0E0C 00 00 00 00 03 00 00 00 00 00 00 00 63 00 00 00 63 00 00 00 80 81 00 00 00 00 00 00 00  ............c...c............
0E29 00 00 00 02 00 40 00 04 02 11 00 08 20 28 00 30 30 30 30 30 30 30 30 30 30 30 63 37 64  .....@...... (.00000000000c7d
0E46 36 37 00 5F 5F 5F 5F 5F 5F 5F 2E 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 00 00 00  67._______...................
0E63 00 00 00 00 00 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00  .............................
0E80 A8 15 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 10 00 00 00 00 00 00 00 24  ¨............................$
0E9D 00 00 00 00 00 00 00 0E 7A D6 64 37 E3 27 15 AE 80 DA 64 37 E3 27 15 AE 80 DA 64 37 E3  ........zÖd7ã'.®.Úd7ã'.®.Úd7ã
0EBA 27 15 A5 71 9B 8A 37 E3 27 15 00 80 00 00 00 00 00 00 01 00 00 00 00 00 00 00 03 00 00  '.¥q›Š7ã'....................
0ED7 00 00 00 00 00 63 00 00 00 63 00 00 00 80 81 00 00 00 00 00 00 00 00 00 00 02 00 40 00  .....c...c.................@.
0EF4 04 02 11 00 08 20 28 00 30 30 30 30 30 30 30 30 30 30 30 63 37 62 66 66 00 5F 5F 5F 5F  ..... (.00000000000c7bff.____
0F11 5F 5F 20 47 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 47 00  __ G........................G.
0F2E 00 00 00 00 00 00 47 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 A6 15 00 00 00 00 00  ......G...............¦......
0F4B 00 00 00 00 00 00 00 00 00 01 00 00 00 10 00 00 00 00 00 00 00 23 00 00 00 00 00 00 00  .....................#.......
0F68 D6 8E CF 64 37 E3 27 15 60 74 D2 64 37 E3 27 15 60 74 D2 64 37 E3 27 15 D6 8E CF 64 37  ÖŽÏd7ã'.`tÒd7ã'.`tÒd7ã'.ÖŽÏd7
0F85 E3 27 15 00 80 00 00 00 00 00 00 01 00 00 00 00 00 00 00 03 00 00 00 00 00 00 00 63 00  ã'..........................c.
0FA2 00 00 63 00 00 00 80 81 00 00 00 00 00 00 00 00 00 00 02 00 40 00 04 02 11 00 08 20 28  ..c.................@...... (
0FBF 00 30 30 30 30 30 30 30 30 30 30 30 63 37 62 66 65 00 54 5F 5F 5F 5F 5F 5F 64 01 00 00  .00000000000c7bfe.T______d...
0FDC 00 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 64 01 00 00 00 00 00 00 00  ....................d........
```

Figure 15: Hex editor view of occupied extended field padding.

(1) **Detectability** - **Low**: The file system check does not find any inconsistencies and a manual investigation would be difficult due to the dynamic and irregular nature of the extended fields and their padding. An example of that can be seen in Figure 15. The reconstruction of already found hidden data would be somewhat easier since the size of each extended field is known through its header (see Table 3).

(2) **Stability** - **Medium**: While the dynamic and irregular nature of the extended fields is beneficial to the detectability of this technique, it is detrimental to its stability. Not all dynamic extended fields are known, but one of them is the file name. If the file name is changed, the size of the field may also change and could overwrite the hidden data in this

[27]This means there is *4 * number of inodes* bytes of space present. The tested 5 GB image has 397 inode entries in the list generated by *InodeTable*, meaning there is 1588 bytes of potential space.

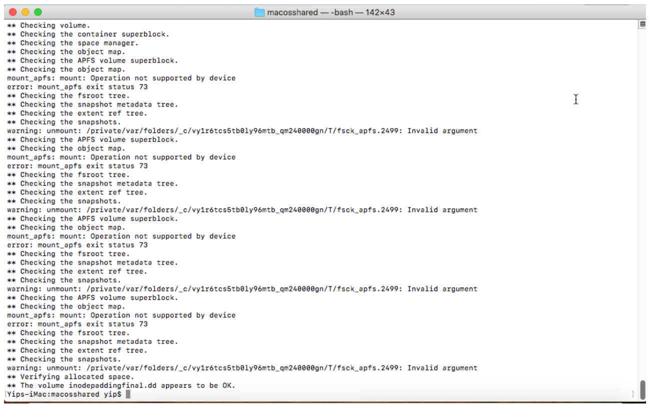
[28]It seems like the nanoseconds part is only 30-bits large with the last 2 bits affecting the seconds. There is a similar behaviour in ext4.

[29]*(4 * used timestamps) * number of inodes* is the formula to calculate the bytes of available space. For example, in the 5 GB test image, there would be 6352 bytes available.

extended field padding, possibly corrupting the entire set of hidden data.

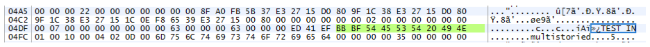
(3) **Capacity** - **Low**: Although the capacity of this technique is rather low, it is not possible to make a general statement about how capable this technique will be. Besides depending on the amount of created files and the total size of a file system, there is also a reliance on the size of some extended fields' content, such as file names. However, it can be assumed that the capacity is in the same range as for the two previous techniques *Write-Gen-Counter* and *Timestamp Hiding*.

5.5 Evaluation of Inode Padding



```
** Checking volume.
** Checking the container superblock.
** Checking the space manager.
** Checking the object map.
** Checking the APFS volume superblock.
** Checking the object map.
mount_apfs: mount: Operation not supported by device
error: mount_apfs exit status 73
** Checking the fsroot tree.
** Checking the snapshot metadata tree.
** Checking the extent ref tree.
** Checking the snapshots.
warning: unmount: /private/var/folders/_c/vy1r6tcs5tb0ly96mtb_qm240000gn/T/fsck_apfs.2499: Invalid argument
** Checking the APFS volume superblock.
** Checking the object map.
mount_apfs: mount: Operation not supported by device
error: mount_apfs exit status 73
** Checking the fsroot tree.
** Checking the snapshot metadata tree.
** Checking the extent ref tree.
** Checking the snapshots.
warning: unmount: /private/var/folders/_c/vy1r6tcs5tb0ly96mtb_qm240000gn/T/fsck_apfs.2499: Invalid argument
** Checking the APFS volume superblock.
** Checking the object map.
mount_apfs: mount: Operation not supported by device
error: mount_apfs exit status 73
** Checking the fsroot tree.
** Checking the snapshot metadata tree.
** Checking the extent ref tree.
** Checking the snapshots.
warning: unmount: /private/var/folders/_c/vy1r6tcs5tb0ly96mtb_qm240000gn/T/fsck_apfs.2499: Invalid argument
** Checking the APFS volume superblock.
** Checking the object map.
mount_apfs: mount: Operation not supported by device
error: mount_apfs exit status 73
** Checking the fsroot tree.
** Checking the snapshot metadata tree.
** Checking the extent ref tree.
** Checking the snapshots.
warning: unmount: /private/var/folders/_c/vy1r6tcs5tb0ly96mtb_qm240000gn/T/fsck_apfs.2499: Invalid argument
** Verifying allocated space.
** The volume inodepaddingfinal.dd appears to be OK.
Yips-iMac:macosshared yip$
```

Figure 16: *fsck_apfs* output when checking an image with data hidden using the *Inode Padding* technique.



```
04A5 00 00 00 22 00 00 00 00 00 00 00 8F A0 FB 5B 37 E3 27 15 D0 80 9F 1C 38 E3 27 15 D0 80   ..."........ û[7ã'.Ð.Ÿ.8ã'.Ð.
04C2 9F 1C 38 E3 27 15 1C 0E F8 65 39 E3 27 15 00 80 00 00 00 00 00 00 02 00 00 00 00 00 00   Ÿ.8ã'...øe9ã'................
04DF 00 07 00 00 00 00 00 00 00 63 00 00 00 63 00 00 00 ED 41 EF BB BF 54 45 53 54 20 49 4E   .........c...c...íAï»¿TEST IN
04FC 01 00 10 00 04 02 0D 00 6D 75 6C 74 69 73 74 6F 72 69 65 64 00 00 00 00 35 00 00 00 00   ........multistoried....5....
```

Figure 17: Hex view of the inode padding after data is hidden.

(1) **Detectability** - **Medium**: As seen in Figure 16, *fsck_apfs* recognises no errors related to this hiding technique[30]. Since this padding field does not contain data in most cases and the corresponding flag mentioned in subsection 4.5 is not usually set, a manual investigation or an APFS-specific forensic tool could reveal this data.

(2) **Stability** - **High**: Since this field is supposed to be empty and is not used by the file system, it will most likely not be overwritten and will only be affected by deleting the inode or the respective file.

(3) **Capacity** - **Low**: While the capacity of this technique grows with the amount of files in a system, it remains comparatively low with only 8 bytes[31] available per inode. Similar to the *Write-Gen-Counter* technique, this technique would only allow small amounts of text, as can be seen in Figure 17.

[30] *The mount_apfs* related errors seen in this screenshot and Figure 11 are not related to the hiding of data but rather to the nature of the used (raw) test images.

[31] *8*number of inodes* is the formula to calculate the space. In the previously used example with 397 inodes, this would amount to 3970 available bytes.

6 CONCLUSION

This paper described several data hiding techniques for APFS. Some of them have been applied in practice, others were treated theoretically. As a proof of concept and as an expansion of the existing anti-forensic framework *fishy*, an APFS module was developed including the APFS interface as well as five different hiding techniques. Furthermore, we gave insight into the functionality of each hiding technique and highlighted the used data structures and vulnerabilities. The evaluation showed that all of the found techniques are able to successfully hide a varying amount of data. With the exception of the *Superblock Slack*, all of these techniques abuse non-critical areas that are primarily present within the inode structure. This means that the capacity of these techniques is solely dependent on the amount of files and directories existing in the container. While there are still some open questions concerning the stability of some of these techniques, this paper proves exploitable areas in Apple's new file system APFS.

7 OUTLOOK AND FUTURE WORK

Besides general improvements to the existing techniques mentioned in Section 5, further additions to the *fishy* APFS module can be made. With the additional vulnerabilities mentioned in Section 4.6, further hiding techniques can be implemented. On the one hand, there are more universal exploits, like the use of File Slack or the potential allocation of additional blocks. On the other hand, there are a few additional APFS-specific exploits. Among those are techniques which (i) use empty spaces in the list of volumes present in the container superblock, (ii) use test fields in the container superblock, or (iii) modify the volume mount logs found in every volume superblock. There are also more general enhancements, such as implementing a method to compress or encrypt data before using a hiding technique. Besides, a more dynamic interface that allows the combination of multiple hiding techniques to enhance the overall capacity would be useful.

In addition to improving the framework's ability to hide data, there would be real added value in identifying potentially hidden data. Using the knowledge gained from implementing and testing the hiding techniques, a new set of tools for data recognition, extraction and interpretation could be added. Many of the existing structures and functions could be modified to facilitate this extension.

ACKNOWLEDGEMENT

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the funding program 'Forschung an Fachhochschulen' (contract number: 13FH019IB6) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP (www.crisp-da.de).

REFERENCES

[1] Avecto.com. 2016. State of Mac Security 2016 Enterprise Mac anagement.

[2] Kevin Conlan, Ibrahim Baggili, and Frank Breitinger. 2016. Anti-forensics: Furthering digital forensic science through a new extended, granular taxonomy. *Digital Investigation* 18 (2016), 66–75.

[3] Andreas Dewald and Jonas Plum. 2018. ERNW WHITEPAPER 65 - APFS internals for forensic analysis. https://www.ernw.de/en/whitepapers/issue-65.html, last visited 05.04.2019.

[4] Knut Eckstein and Marko Jahnke. 2005. Data Hiding in Journaling File Systems.. In *DFRWS 2005 USA*.

[5] Thomas Göbel and Harald Baier. 2018. Anti-forensics in ext4: On secrecy and usability of timestamp-based data hiding. *Digital Investigation* 24 (2018), 111–120. https://doi.org/10.1016/j.diin.2018.01.014

[6] Thomas Göbel and Harald Baier. 2018. fishy - A Framework for Implementing Filesystem-based Data Hiding Techniques. In *Proceedings of the 10th EAI International Conference on Digital Forensics and Cyber Crime (ICDF2C), New Orleans (United States), September 2018.*

[7] Grugq. 2002. Defeating Forensic Analysis on Unix. https://grugq.github.io/docs/phrack-59-06.txt, last visited 21.06.2019.

[8] Kurt H. Hansen and Fergus Toolan. 2017. Decoding the APFS file system. *Digital Investigation* 22 (2017), 107–132. https://doi.org/10.1016/j.diin.2017.07.003

[9] Ryan Harris. 2006. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. *Digital Investigation* 3 (2006), 44–49.

[10] Apple Inc. 2019. Apple File System Reference. https://developer.apple.com/support/apple-file-system/Apple-File-System-Reference.pdf, last visited 29.01.2019.

[11] Stephen P Larson. 2009. Concerning File slack. (2009). http://commons.erau.edu/cgi/viewcontent.cgi?article=1090&context=adfsl, last visited 24.04.2019.

[12] XIAODONG LIN. 2019. *INTRODUCTORY COMPUTER FORENSICS: A hands-on practical approach.* SPRINGER, [S.l.].

[13] Sebastian Neuner, Artemios G Voyiatzis, Martin Schmiedecker, Stefan Brunthaler, Stefan Katzenbeisser, and Edgar R Weippl. 2016. Time is on my side: Steganography in filesystem metadata. *Digital Investigation* 18 (2016), 76–86.

[14] Bhushan Patole, Ashok Shinde, Mohit Bhatt, and Pallavi Shimpi. 2018. Data Hiding in Audio-Video using Anti Forensics Technique for Secret Message and Data in MP4 Container.

[15] Laura Pfeiffer. 2017. Forensische Analyse des Apple File System (APFS).

[16] Jonas Plum. 2018. Checksum calculation for APFS, taken from afro implementation. https://github.com/cugu/afro/blob/master/afro/checksum.py, last visited 07.01.2019.

[17] Jonas Plum and Andreas Dewald. 2018. Forensic APFS File Recovery. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*. ACM, 47.

[18] BlackBag Technologies. 2019. Providing APFS Support to the Sleuth Kit Framework. https://www.blackbagtech.com/blog/2018/12/19/present-santa-apfs-providing-apfs-support-sleuth-kit-framework, last visited 16.04.2019.