An Introduction to the exFAT File System and How to Hide Data Within

Julian Heeger*, York Yannikos and Martin Steinebach

Fraunhofer SIT, Germany
E-mail: julian.heeger@sit.fraunhofer.de; york.yannikos@sit.fraunhofer.de; martin.steinebach@sit.fraunhofer.de
**Corresponding Author*

Received 15 December 2021; Accepted 18 January 2022;
Publication 16 March 2022

Abstract

In the recent years steganographic techniques for hiding data in file system metadata gained focus. While commonly used file systems received tooling and publications the exFAT file system did not get much attention – probably because its structure provides only few suitable locations to hide data. In this work we present an overview of exFAT's internals and describe the different structures used by the file system to store files. We also introduce two approaches that allow us to embed messages into the exFAT file system using steganographic techniques. The first approach has a lower embedding rate, but has less specific requirements for the embedding location. The other one, called *exHide*, uses error correcting to allow for an more robust approach. Both approaches are specified, evaluated and discussed in terms of their strengths and weaknesses.

Keywords: Data hiding, file systems, anti forensic.

Journal of Cyber Security and Mobility, Vol. 11_2, 239–264.
doi: 10.13052/jcsm2245-1439.1125

1 Introduction

Steganography is a part of cryptology enabling hidden communication in contrast to obvious by scrambled communication provided by cryptography [12]. It is used for hiding data within a cover (like an image, a data stream or a file system)and is often technically identical to digital watermarking. The only widely accepted difference between the two techniques is that watermarking embeds data relevant for a cover and stenography embeds a message into a cover with no relation to the message [5]. We will address the concept of robustness later in this work, and usually watermarking requires robustness while steganography does not, but there are also concepts of fragile (non-robust) watermarking. Therefore robustness is not a distinguishing characteristic between watermarking and steganography. In this work we only address digital steganography. There are many known methods in the history ranging from knot patterns to microdots that work in the analogue domain. For further reference we suggest the many books available on the subject.

Steganography is often seen as a technique to enable illegal data transfer, especially from the perspective of steganalysis. Commonly discussed use cases of steganography are the distribution of terrorism-related information [7, 14], espionage, or exfiltration of data gathered using malware. Still, steganography is dual-use technology, similar to anonymity networks like Tor or encryption in general. Many scenarios are possible where steganography is used legally, like deniability or circumvention of censorship [1].

Due to the increase in surveillance worldwide, plausible deniability of information transfer is gaining more and more importance. For example at borders, smartphones or other electronic devices may be searched [4], which can only be countered by hiding relevant information with suitable tools. The use of cryptography alone is not effective, since the mere existence of encrypted data could raise additional suspicion and the screened person could then be forced, e.g. by existing laws, to decrypt the data [9]. Therefore, relevant data should better be hidden in such a way that automated screening tools can not find or recognize them. This can be achieved with steganography: While cryptography hides the content of the message, steganography hides the message itself.

While many steganographic tools exist to hide data by making unperceivable changes to multimedia files like images, video, or audio, there are also other approaches that use changes in metadata or structural information to hide information. One example is hiding data within file system metadata. Recently, steganographic techniques to hide data in file systems like FAT32,

ext4, or NTFS have been proposed. However, not much work has been done on the exFAT file system, the successor of FAT32. Although exFAT is not as widely used as FAT32 yet, its optimizations for flash memory makes it very attractive for use in modern smartphones and on SD cards for digital cameras.

Even if there would be no known acceptable used cases for steganography like those mentioned above, research in this domain would still be of importance: understanding the potential strategies for steganographic embedding lays the foundation of steganalysis. The goal of steganalysis is to develop tools for recognizing the usage of steganography. These tools usually depend on knowing the embedding algorithm. Based on this knowledge, they build detectors for changes caused by the embedding process.

This journal article is an extended version of the paper *exHide: Hiding Data within the exFAT File System* by [11] et al. and gives an deeper introduction to the exFAT File System. Specifically the layout of files and folders on an volume and their interaction with each other was extended.

Contribution

In this work we take a deep dive into the exFAT file system and describe the inner workings of the file system. In a second step, we describe its use for steganography. exFAT was released by Microsoft in 2006 as successor of the widely used FAT32 file system with optimizations for flash memory like SD cards or USB drives. We analyse the internal metadata structure of exFAT and identify suitable locations to embed data. We then propose, evaluate, and discuss two approaches to hide data within exFAT metadata.

Outline

This remainder of this work is structured as follows: In Section 2 we give a deep explanation of the exFAT file system. We discuss related work in Section 3 and propose two approaches for data hiding in exFAT in Section 4. In Section 5 we evaluate and discuss our approaches and conclude in Section 6.

2 exFAT File System Specifics

exFAT is the latest version of the FAT family. The file system introduced by Microsoft in 2006 is intended both to replace the old version FAT32 and the associated limitations and to meet the current requirements for a file system. FAT file systems are used if as much memory as possible is to be used for files or if the memory is to be used for files or the devices which are to use

Table 1 Boot sector

Name	Size (byte)	Name	Size (byte)
JumpBoot	3	VolumeSerialNumber	4
FileSystemName	8	FileSystemRevision	2
VolumeFlags	2	MustBeZero	53
PartitionOffset	8	BytesPerSectorShift	1
VolumeLength	8	SectorsPerClusterShift	1
FatOffset	4	NumberOfFats	1
FatLength	4	DriveSelect	1
ClusterHeapOffset	4	PercentInUse	1
ClusterCount	4	Reserved	7
RootDirectoryCluster	4	BootCode	390
		BootSignature	2

the memory are limited in computing capacity and memory (e.g. embedded systems). Typically exFAT is used by memory cards for digital cameras or for memory expansion for Android phones.

2.1 Volume Layout

The layout of an exFAT partition, shown in detail in Figure 1, can be divided into three different regions. The first region is the *Boot Region*, also called the *superblock*, which occupies the first 512 bytes of each exFAT partition. This is where the basic configuration of the file system is stored, which provides information on how to interpret the partition. The *superblock* is duplicated and written directly after the original to have a backup in case of corruption of the original. The next region is the *FAT Region*, which stores the file allocation table shown in Table 2. The last region is the *Data Region*, which stores, in addition to the contents of files, also the metadata of files and folders.

Figure 1 illustrates the exact boundaries of the individual regions. The fields *BytesPerSectorShift* and *SectorsPerClusterShift* from the superblock are used to calculate the offsets, based on the Equations (2) and (1).

$$\text{Byte} = \text{Sector} \ll 2^{\text{BytesPerSectorShift}} \tag{1}$$

$$\text{Sector} = \text{Cluster} \ll 2^{\text{SectorsPerClusterShift}} \tag{2}$$

2.2 File Allocation Table

The name FAT stands for *File Allocation Table* and describes the way in which files are stored on the system. Figure 2 shows such a table. The first

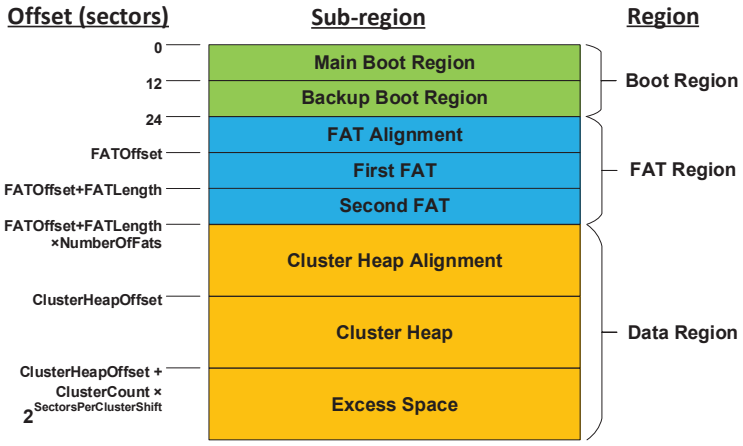


Figure 1 Volume structure.

Table 2 Example: file allocation table

Cluster	Value	Offset (byte)
0	0xFFFFFFF8	+0
1	0xFFFFFFFF	+4
2	...	+8
⋮		⋮
40	41	+160
41	80	+164
42	...	+168
43	...	+172
⋮		⋮
80	101	+320
81	...	+324
⋮		⋮
100	...	+400
101	0xFFFFFF	+404
102	...	+408

two rows are predefined, because the first two clusters of an exFAT file system cannot be allocated by the user. Each row in the table represents one cluster on the partition. The offset specifies the number of bytes between the first cluster and the current one. Each entry is 4 bytes in size and contains a value between 2 and the number of clusters plus one. Two additional values

Field Name	Offset (byte)	Size (byte)
EntryType	0	1
SecondaryCount	1	1
SetChecksum	2	2
FileAttributes	4	2
Reserved1	6	2
CreateTimestamp	8	4
LastModifiedTimestamp	12	4
LastAccessedTimestamp	16	4
Create10msIncrement	20	1
LastModified10msIncrement	21	1
CreateUtcOffset	22	1
LastModifiedUtcOffset	23	1
LastAccessedUtcOffset	24	1
Reserved2	25	7

(a) File Directory Entry

Field Name	Offset (byte)	Size (byte)
EntryType	0	1
GeneralSecondaryFlags	1	1
Reserved1	2	1
NameLength	3	1
NameHash	4	2
Reserved2	6	2
ValidDataLength	8	8
Reserved3	16	4
FirstCluster	20	4
DataLength	24	8

(b) Stream Extension Directory Entry

Field Name	Offset (byte)	Size (byte)
EntryType	0	1
GeneralSecondaryFlags	1	1
FileName	2	30

(c) File Name Directory Entry

Field Name	Offset (byte)	Size (byte)
EntryType	0	1
CustomDefined	1	31

(d) Template for metadata structures

Field Name	Size (bot)
TypeCode	5
TypeImportance	1
TypeCategory	1
InUse	1

(e) EntryType

Figure 2 Metadata entries.

are `0xFFFFFFF7`, which identifies a defective cluster, and `0xFFFFFF`, which indicates the end of a file, called the *end-of-file* marker.

Figure 2 shows that the file in cluster 40 continues in cluster 41. Thus, the values in the clusters can be followed further until the end of the file has been reached in cluster 101. The information that a file starts in cluster 40 comes from is explained in Section 2.4 about the metadata.

2.3 Root Directory

To store files in an exFAT file system, metadata about the file must be written to the partition in addition to its content. This metadata contains information about the file, e.g. the size or time of creation. The superblock contains a field called *RootDirectoryCluster*, which specifies a cluster, in which the root folder of a file system is located (e.g. the folder/on Linux). This cluster contains, in addition to entries of data and folders, special entries that are

Table 3 Directory entry structures

(a) Allocation Bitmap Directory Entry

Field	Size (byte)
EntryType	1
BitmapFlags	1
Reserved	18
FirstCluster	4
DataLength	8

(b) Volume Label Directory Entry

Field	Size (byte)
EntryType	1
CharacterCount	1
VolumeLabel	22
Reserved	8

(c) Up-case Table Directory Entry

Field	Size (byte)
EntryType	1
Reserved1	3
TableChecksum	4
Reserved2	12
FirstCluster	4
DataLength	8

important for exFAT, like the name of the partition, which is stored in the **Volume Label** (see Table 3(b)). Each of those entries are based on the template of Table 2(d).

To find a free cluster on the partition where content of a file can be stored, the **Allocation Bitmap** (see Table 3(a)) is needed. This bitmap stores which clusters are occupied on the partition, by setting the bit using the cluster number as its index. A free cluster in an exFAT file system does not have to mean that the memory is zeroed. To prevent unnecessary write operations, the cluster is simply marked as free. This increases the life of flash memories, which are used e.g. in memory cards or USB sticks.

The **Upcase Table** structure defines a list of character conversions; the exact fields are listed in Table (see Table 3(c)). The table converts lowercase letters to the respective uppercase letters based on their corresponding Unicode representations. exFAT as a file system ignores case sensitivity for file names; thus, the Upcase Table as a lookup table increases the speed with which a name can be searched for.

2.4 Files and Folders

As described earlier files are stored in the *Data Region* of an exFAT partition. In this part of the file system two different types of clusters exist:

data cluster, which contain the content of files, and metadata cluster, which contain information about the files. Therefore the cluster specified by the *RootDirectoryCluster* is a metadata cluster. Each file and folder, regardless of whether it is empty or not, occupies at least a cluster.

The metadata for a file or folder is composed of three different entries, which are based on a generic template, shown in Figure 2(d). The first byte of each structure starts with the *EntryType* field, which groups information about the entry into different flags. Only the first 5 bits are responsible for the actual identification of the structure. *TypeImportance* 2(e) describes whether this entry is critical for exFAT, so that without it the file system cannot function. The next field namely, *TypeCategory* 2(e), distinguishes between additional information (length of the file name, size of the file) and the main information (creation date, modification date). The last bit indicates whether the structure and thus the file is used. If the bit is not set, the file is considered to be deleted.

exFAT as a file system is, as described in the introduction, primarily intended to be operated on memory cards or USB sticks. Therefore, the write accesses are to be minimized in order to relief the hardware so that it can remain in operation longer. Hence, if a folder is deleted, only the metadata of this folder is changed. The FAT table and the allocation bitmap are adjusted accordingly, freeing up the memory again. The metadata of the files in the deleted folder are not touched; since for exFAT the folder does not exist any more, there is also no further linking to the data in this folder. Thus there may exist files that have been deleted but still have their *InUse* bit set.

All metadata entries belonging to a file are grouped together in a cluster without any interruptions, shown in Figure 3. The first entry for each file is the *File Directory Entry* (see Figure 2(a)) which contains the timestamps and

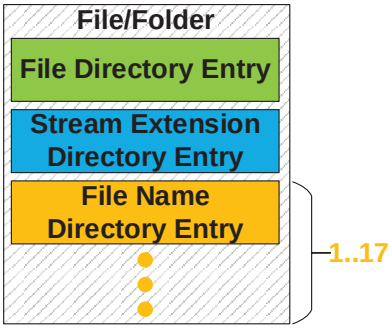


Figure 3 Directory Entries for a file or folder.

a basic checksum over all metadata entries. The *FileAttributes* field specifies via the *Directory* flag if the element described in these entries is a file or directory (this flag is the only distinction between a file and a directory within exFAT). The timestamps allow for a two-second resolution with an additionally ten-millisecond field for the last modified and create timestamp. Each ten-millisecond field has a valid range from 0 to 199.

The *Stream Extension Directory Entry* 2(b) contains metadata about the name and size of the file/folder. The fields *ValidDataLength* and *DataLength* both describe the size of a file/directory. While the first describes how much of the file content has been written, the latter describes the total file size in bytes. If the complete file was written successfully, both fields are identical. Therefore, when we talk about file size, we refer to both fields. If the structure describes an folder, the size of the metadata written in the cluster is used as value. The *FirstCluster* field points to the first cluster containing the content of the file. If the metadata describe a folder, the referenced cluster contains metadata of all the files and folders residing in it. Based on this field all folders on the exFAT file system can be found, starting at the *RootDirectoryCluster*.

For example, if the content of a file `005_31.jpg` with the path `/DCIM/2021/01/` should be found, the search starts at the root folder (*RootDirectoryCluster*), here `/`. Figure 4 shows this search graphically. Based on the *FirstCluster* value of the root folder's metadata, the cluster is selected to find the metadata for the next folder, here *DCIM*. The metadata in the cluster is searched in order until the end of the cluster is reached. If the searched DCIM folder has not been found by then, the *File Allocation Table* is looked up to see if the contents of the folder span multiple clusters. If this is the case, the *File Allocation table* indicates which cluster must be searched next. If the folder is found, the procedure is repeated, and folder after folder is searched until the until the file `005_31.jpg` has been found. In order to now also obtain the contents of the file, the *FirstCluster* field is read; this specifies the first cluster in which the contents of the file were stored. If the file is

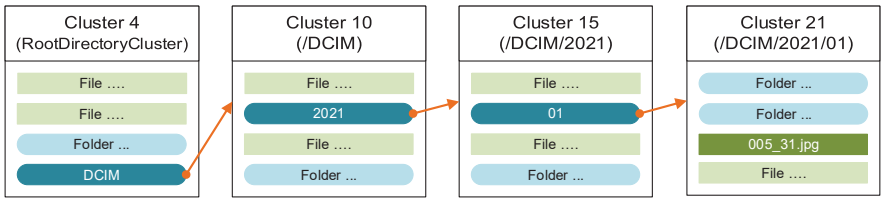


Figure 4 Lookup of a file.

larger than one cluster, the *File Allocation Table* is queried as before to read the complete file.

While the previously defined entries exist only once per file, the *File Name Directory Entry* (see Figure 2(c)) may be present up to seventeen times. Each of the entries contain up to 15 Unicode characters of the filename in the *FileName* field. Therefore the maximum length of file and folder names in exFAT is 255 Unicode characters.

3 Related Work

Steganography hides a message in an object, called a carrier or cover, so that the carrier containing the message looks unchanged. Multimedia data, such as images or videos, are well suited as carriers for messages. Most common are Least Significant Bit methods (LSB) encoding the message into the least significant bit of a pixel or sample.

This work addresses steganography in file systems, specifically the hiding of messages in the metadata of the file system. In [10] Göbel and Baier show that within the ext4 file system it is possible to embed information into the nanosecond part of timestamps. Four of the five timestamps in the ext4 file system provide an extra 30 bytes for a nanosecond part compared to previous file system versions. With their proposed method the authors can embed 6.5 bytes per file.

Neuner et al. provide an overview of how timestamps in different file systems can be used for hiding messages [13]. Among other file systems, they examined FAT32, the predecessor of exFAT. FAT32 has only an accuracy of 2 seconds for the creation and modification date. Also, it uses only a day-precision for the last access date. The authors eventually use the NTFS file system to develop and evaluate an implementation of their proposed approach.

Eckstein and Jahnke show how messages can be hidden in a file system that uses journaling [6]. They propose a method of manually creating inconsistencies by allocating storage without assigning inodes to it. This allows a method of hiding data which is robust against sudden system crashes.

Göbel et al. address the Apple File System in [8] for which they show methods to hide data. They develop a module for *fishy* – a framework to test out common data hiding techniques on file systems – to demonstrate the practical application of their method. The authors implement the techniques separately from the corresponding file systems to allow an easy extension of the fishy framework.

Vandermeer et al. applied reverse engineering to examine artifacts of the exFAT file system [18]. They conducted their research before Microsoft released the exFAT specification to the public.

The authors also have addressed other aspects of steganography in their work previously, especially in steganalysis. To prevent confusion, that work is about graphical images, not file system images. It includes strategies for blind image steganalysis like [16] [15] [17] and non-blind steganalysis like [2] [3].

4 Data Hiding in exFAT

Several approaches were investigated and, based on different scenarios, two are presented. The stego-only approach, uses the methods of steganography and adopts to their assumptions. Therefore the solution does not need to be robust against changes of the carrier medium and active attacks against the carrier medium are not considered as part of the attacker model. The second approach, referred to as *exHide* in the following, is a method that is more closely associated with data hiding.

The basic principle for both approaches is, that the data must be divided into several parts, so-called blocks. Each of these blocks is embedded into the metadata of a file, whereby the block size depends on the respective solution. Potential embedding locations are fields in the metadata structures, which can be changed without creating inconsistencies in the exFAT file system. Additional care must be taken to ensure, that the embedding of data and the associated manipulation of metadata does not generate any anomalies, which could be easy to detect. Since exFAT does not have complex structures or additional functionalities such as journaling, data can only be embedded in the metadata structures described in Section 2.

Both approaches use the same cryptographic methods, illustrated in Figure 5, to ensure that the embedded data cannot be read without a password. The user supplied password, is hashed with *SHA-512* to map it to a fixed

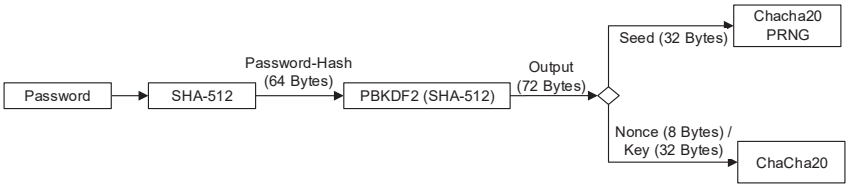


Figure 5 Encryption of the input data.

length of 64 bytes. This output is then taken to initialize a password-based key derivation function (*PBKDF2-SHA512*), by splitting it into two 32 byte blocks and to be used as key and salt. Based on the generated key material of 72 bytes a pseudorandom number generator (PRNG) and a stream cypher (*Chacha20*) is initialized. A stream cipher is used, because it does not change the length of the resulting cipher text. *ChaCha20* is used as a PRNG, which is seeded with the first 32 bytes of the generated key material. The remaining key material is used as a 8 bytes nonce and 32 bytes key for the stream cipher.

4.1 Stego-only Approach

The stego-only approach has a low embedding rate, but it is more difficult to detect than the second approach.

4.1.1 Locations to embed

In the stego-only approach, the two fields *Create10msIncrement* and *LastModified10msIncrement* are used to embed data in both existing and deleted data. Both provide additional time resolution in ten-millisecond multiples for the creation and modified date and are each 1 byte (8 bits) in size. The maximum value that the fields can assume is 199 according to the standard. On an unchanged file system, an equal distribution of the values is assumed.

To ensure that the maximum value of 199 is not exceeded during embedding, the most significant bit is not used. However, with 7 bits only values between 0 and 127 are possible to embed, which means that values between 128 and 199 are not embedded. This would be noticeable in a statistical analysis, because the values 0 to 127 would occur more frequently than values between 128 and 199. To achieve an equal distribution of embedded values, only 6 bits are used. Thus only values between 0 and 63 ($2^6 - 1$) are embedded, in addition however one of the two most significant bits is set randomly during embedding. For example, if the value 50 (`0011 0010`) should be embedded, there are three different options to do this. Either the most significant bit is set, so that the value 178 (`1011 0010`) is written into the field, or the second most significant bit set, and instead of 50, the value 114 (`01011 0010`)) is written. As a last option the value is not changed, and 50 is embedded. The least significant 6 bits are not changed and during extracting in any of the above options 50 is read.

When embedding, an uniform distributed PRNG is used to decide, which of the above mentioned cases is used for embedding. By doing so, an uniform distribution of the values between 0 and 191 is achieved. This approach has

one problem, values between 192 and 199 are not reachable, therefore they are not embedded. To solve this, the values 192 to 199 are further written into the carrier medium on average as often as the values 0 to 191 were during embedding. In doing so, an uniform distribution between 0 to 199 is achieved.

4.1.2 Selection of metadata for embedding

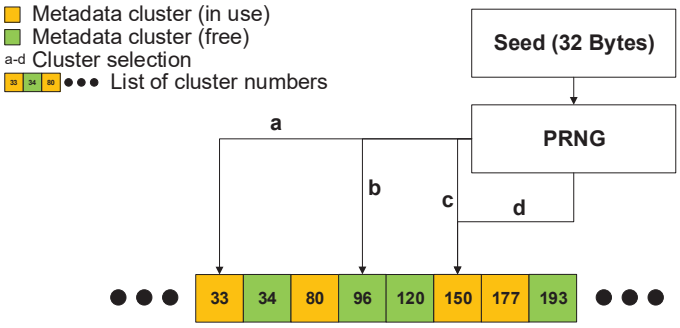
In a first step, a list of all metadata clusters, the metadata cluster list, is created. This includes metadata clusters that are still used by the exFAT file system, as well as metadata clusters that are free.

Clusters that are marked as in use can be found by starting from the *RootDirectoryCluster* and traversing each subdirectory. For this, all directory entries in the *RootDirectoryCluster* are listed; their *FirstCluster* fields indicate the new clusters in which further metadata of files and directory can be found. Accordingly, these clusters are searched again for directory entries. Based on these new directories, the search is repeated until no new directory entries can be found.

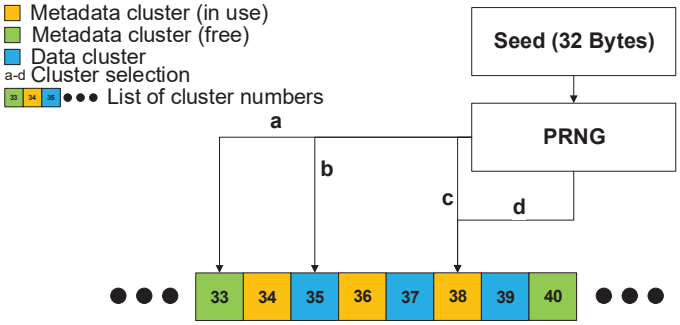
For metadata clusters that are no longer used by exFAT, every free cluster has to be searched. To decide, if a cluster contains metadata, it can be exploited that each metadata structure has a size of 32 bytes. Additionally, as shown in Table 2(c), each metadata structure has an *EntryType* as it first field, and thus its first byte. Therefore, the first byte of each 32-byte block of a cluster can be used to decide whether it may or may not be a metadata structure. Based on these information in a potential metadata cluster every 32nd byte is read and is checked whether it can be an *EntryType*. Each metadata entry has two different *EntryType* values, the original value and the value without the *InUse* bit set. For example `0xC0` and `0x40` are valid *EntryType* values for the *Stream Extension Directory Entry*. If the read bytes are valid according to the above procedure, the cluster is added to the metadata cluster list as a metadata cluster.

The selection in which metadata the next block of data is embedded, is selected via the PRNG. After its initialisation with a seed, described in Section 4, the then generated random numbers are deterministic. The PRNG selects a cluster from the metadata cluster list and tracks the number of times this cluster was selected.

With the selected cluster and the number of times this cluster was already selected, a metadata entry can be selected. Figure 6(a) illustrates a selection using the arrows $a - d$. If a metadata cluster is selected for the first time (Figure 6(a) arrow a), it is embedded into the first possible metadata. If a cluster is selected a second time by the PRNG (Figure 6(a) arrows c and d)



(a) Stego-only



(b) exHide

Figure 6 Specifying the metadata entries to use.

it will be embedded in the second possible metadata entry. If there are not enough metadata entries in a cluster, it will be skipped and the PRNG selects a new cluster. Since the file system does not changed after the embedding, this cluster is also skipped when extracting the message.

4.1.3 Embedding

Alice wants to send a secret message to Bob. Both must have set a password in advance, which Alice can use for embedding. Alice takes a usb stick or memory card that contains an exFAT file system and embeds her message. The embedding works as follows.

The input data is encrypted using *Salsa20* and each is split into 12-bit blocks. The first block is special, as it contains the size of the data to be embedded in bytes. The metadata cluster list is initialized so that, together with random number generator, the 12-bit blocks can be embedded. Based

on Section 4.1.2 metadata entries are selected and checked if the metadata is corresponding to a file. If this is the case, the two fields *Create10msIncrement* and *LastModified10msIncrement* are overwritten with the data from the 12-bit block. In doing so, the block is divided into two 6-bit blocks and those are embedded as described in Section 4.1.1. If the amount of data to be embedded is not a multiple of 12 bits, the *Create10msIncrement* field is written to first. Since the two fields have been modified, the *SetChecksum* field of the *File Directory Entry* must be recalculated and updated. This process is repeated until all 12-bit blocks are embedded. After all data has been written, it is calculated how often the values between 192 and 199 must be written additionally. This data is then embedded using the same procedure as described previously. To do this, the current state of the random number generator is used and not reinitialized, so no embeddings are overwritten. Alice can now pass the exFAT file system to Bob.

4.1.4 Extracting

Bob receives an exFAT file system from Alice, with an embedded message. To extract Alice's message, Bob needs to initialise the PRNG, metadata cluster list and the encryption function, as described in Section 4. Again, metadata entries are selected as described in Section 4.1.2 and the message parts are read from the two *Create10msIncrement* and *LastModified10msIncrement* fields until the size, read from the first block, is reached. In the end, the message can be decrypted using the password.

4.2 exHide Approach

The second approach offers a higher embedding rate than the stego-only method, but it also has more specific requirements for the exFAT file system and is easier to detect, although measures were taken for making the embedding as difficult to detect as possible. For this method, embedding data is written only in metadata entries of deleted files, since exFAT allows few possibilities for metadata manipulation.

4.2.1 Locations to embed

Since *exHide* can only be used to embed in metadata of deleted files, the carrier medium must also contain deleted files. This enables a wider range of modifications to the metadata, as there is now a disconnect between the file content and the information stored in the metadata. Section 2 described, that metadata of deleted files are still on the file system, therefore the clusters,

in which the content of these files were located could be overwritten, hence the disconnect between file content and metadata. This approach tries to be as plausible as possible with all modifications done to the metadata for a more difficult detection. To achieve as little change as possible, embedding is done in the bytes of a field that have the least influence on the fields value. Assuming a big-endian byte representation, these are the last bytes of a value. Based on this byte order, *Byte1* is considered the least significant 8 bits and *Byte2* is defined as the 8 bits before that.

The *Create10msIncrement* field from the stego-only approach is used. Since only 6 bits can be embedded in the field, 2 more bits are needed to be able to embed a byte. To achieve this, one bit is embedded in each of the double-second part of the timestamp fields *CreateTimestamp* and *LastModifiedTimestamp* 2(a). Additional to the stego-only field, the *FirstCluster* and the file size fields *ValidDataLength* and *DataLength* are used, but because both of them need to be identical, they only count as one additional field. For a low detection and to achieve plausible values in the modified fields only *Byte1* and *Byte2* are used to embed data. Therefore an additional amount of 4 bytes can be embedded per metadata. For the *FirstCluster* field this has the additional benefit, that the field points to cluster numbers, which are higher than the amount of existing clusters on the file system. Similar as the file size fields describe the file size in bytes, embedding large numbers in these fields, could create files, which are larger than the carrier medium. To make the steganalysis harder, *exHide* requires that all files need to be larger than 65 KiB and the first 65537 ($2^{16} + 1$) clusters must be occupied before an embedding can take place. With this restrictions in place, it can be assumed that *Byte1* and *Byte2*, in both fields, are uniform distributed.

4.2.2 Selection of metadata for embedding

When embedding in the exFAT file system, a distinction has to be made between the three cluster types: data cluster, used metadata cluster, and free metadata cluster. Free metadata clusters contain metadata of a deleted folder, while used metadata clusters can contain deleted and used files. In a free metadata cluster it is possible, to embed data in both metadata entries with the *InUse* bit set and in those without. While in a used metadata cluster data can only be embedded in metadata entries without the *InUse* bit set. The metadata cluster list contains all clusters of the file system to ensure, that the length of the list is identical during embedding and extracting. This is important, because if files are written after embedding, the cluster types can

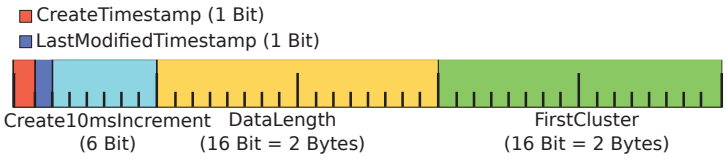


Figure 7 exHide: Embedding block.

change and a different cluster is selected during extracting. Figure 6(b) shows a similar procedure, as described in 4.1.2, with the addition of data clusters. If a data cluster is selected it is skipped and a new cluster is selected. The other cluster types are treated the same as in the stego-only approach.

4.2.3 Embedding

The same scenario as for the stego-only approach unfolds. Alice wants to send a message to Bob encrypted with a password they exchanged some time before. Based on this password, the random number generator and the encryption function are initialized. Alice's message is encoded using an error correction method, in this case Reed-Solomon. This allows for robustness against loss of embedding blocks, when the data is extracted. The encoded message is encrypted with *Salsa20* and split into 5-byte blocks, displayed in Figure 7.

The first block contains the length of the data and since the length does not need to be a multiple of 5, an order must be specified in which fields are embedded first. The first byte is embedded in *Create10msIncrement* together with *CreateTimestamp* and *LastModifiedTimestamp*, then 2 bytes into *DataLength* and finally 2 bytes into *FirstCluster*. The block is written to the file system as described in Section 4.2.2. After the message is embedded, the same method as in the stego-only approach is used to correct the uniform distribution of the *Create10msIncrement* field.

4.2.4 Extracting

Bob receives an exFAT file system from Alice, with an embedded message. To extract the message Bob needs to initialise the PRNG, metadata cluster list and the encryption function, as described in Section 4. He begins extracting message parts, as described in Section 4.2.2, until the size, read from the first block, is reached. After all data has been extracted, it is decrypted. The message is then decoded with Reed-Solomon to correct errors during extraction.

5 Evaluation

To evaluate both of our approaches we created exFAT file systems with several existing and deleted files and directories. Using the corresponding file system metadata of the files, we applied the stego-only approach and exHide and compared both regarding embedding rate and detectability.

For our stego-only approach the files on the exFAT file system do not have to meet any special requirements, so random files could be written to the file system. In contrast to this, *exHide* requires a minimum number of files on the file system that are larger than 65 KiB (for embedding in the file size metadata fields) and a minimum total number of used clusters (for embedding in the *FirstCluster* field). This ensures that embedding data with exHide does not produce metadata entries that are not plausible. Based on these limitations, Figure 8(a) shows a listing of file system sizes and the corresponding number of files that would be available for embedding. Figure 8(b) shows the number of files needed for embedding different message sizes. To reach the maximum number of files with file system metadata suitable for embedding, we considered a file size of exactly one cluster for the stego-only approach and a file size of exactly 65 KiB for exHide. Additionally, we considered the file name to occupy only one *File Name Directory Entry*, i.e. to be shorter than 15 characters. In total, we considered a maximum storage space of 96 bytes per file metadata for both approaches. However, in a more realistic scenario the number of files typically existing on a file system is lower.

file system size	cluster count	cluster size	files (stego-only)	files (exHide)
1GiB	4KiB	262.144	256.047	16.359
16GiB	4KiB	4.194.304	4.096.762	261.754
32GiB	128KiB	262.144	261.952	261.952
64GiB	128KiB	524.288	523.904	523.904
128GiB	128KiB	1.048.576	1.047.808	1.047.808
256GiB	128KiB	2.097.152	2.095.616	2.095.616
512GiB	128KiB	4.194.304	4.191.233	4.191.233

(a) Embedding rate with stego-only and exHide

Input size	files (stego-only)	files (exHide)
1KiB	682	204
100KiB	68.266	20.480
500KiB	341.333	102.400
1MiB	699.050	209.715
5MiB	3.495.253	1.048.576
10MiB	6.990.506	2.097.152
50MiB	34.952.533	10.485.760

(b) Files needed per embedding data for stego-only and exHide

Figure 8 Embed rate and files needed for both approaches.

In the following we show graphs depicting the values of different metadata fields of files within an exFAT file system. The corresponding files on the file system were selected randomly from a large pool of test files and written to the file system. The metadata fields were then measured before and after embedding.

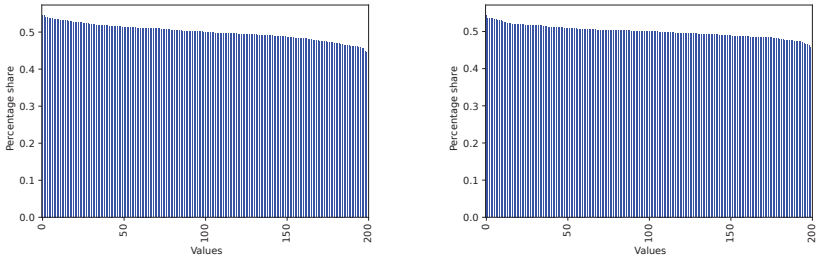
Based on this information, exFAT file systems could be created to evaluate the solution approaches. For this purpose, exFAT file systems filled with random dat of 32 GiB were created. Here, this data was larger than 65 KiB in order to also be used by the DHide solution approach. 80,0000 random files were written to the file system from a larger pool of files, a portion of which was randomly deleted. This was done in compliance with the limit of files that must remain on the file system. Thus, the exFAT file systems created in this way could be used to evaluate both full stego and DHide. For each solution approach 25 file systems were created and evaluated.

Once the file system was prepared, embedding could be performed. A fixed amount of time was spent initializing the encryption system; the embedding itself varied with the amount of embedding. The difference in embedding and reading the DHide solution approach was minimal, as the same operations were performed in reverse order for both actions. For example, time it takes to encrypt before embedding was the same as that of decrypting after readout. In the full stego solution approach, when embedding and reading out 10 KiB, reading out was faster. However, the difference was only half a second, which could have been due, for example, to checking for the end marker each time a block was read.

When embedding a 100 KiB file, the embedding was much slower than the readout. This could have been due to the nature of the embedding, since a file had to be converted to a bit stream. This could have had poor performance when read out. Appending bits to the bit stream seemed to be more performant. All other operations were executed for both actions, embedding and readout, were executed.

5.1 10msIncrement Fields

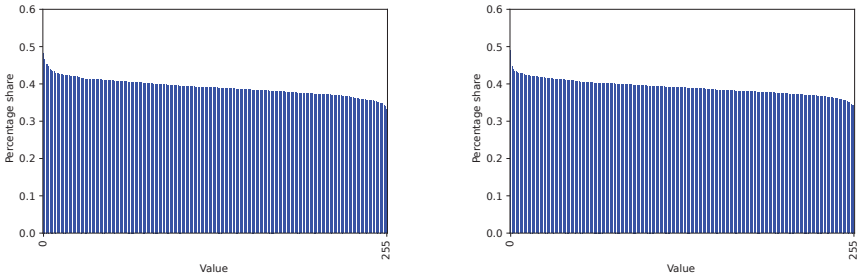
Figure 9 shows the file system before and after 100 KiB of data has been embedded into them using the stego-only approach. It was noticeable that the distribution of both the *LastModified10msIncrement* and *Create10msIncrement* field is only approximately a uniform distribution and small values occur more frequently than larger ones (0–7, 190–199). This would suggest that the additional embedding of the values from 192 to 199



(a) *LastModified10msIncrement* (before embedding)

(b) *LastModified10msIncrement* (after embedding)

Figure 9 Byte value distribution for *LastModified10msIncrement*.



(a) *Byte1* value distribution of the file size field (before embedding)

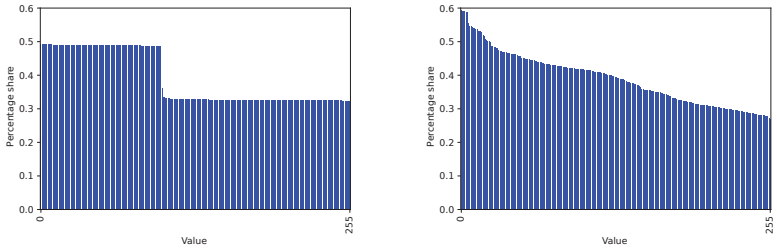
(b) *Byte1* value distribution of the file size field (after embedding)

Figure 10 File size fields.

was not sufficient. However, since the values 190 to 199 already occurred less frequently in the unmodified distribution, the embedding of the uniform distributed input data could also have resulted in a greater expression of the deviations from a uniform distribution.

5.2 FirstCluster and File Size

Figure 10 shows the distribution of *Byte1* from the file size field before and after embedding. It displays a similar distribution like the *LastModified10msIncrement* fields. Smaller values between 0 and 7 possessed a higher frequency than the other values. The distribution of file sizes are dependent on the files on the file system. Since these were synthetically generated, it cannot be ruled out, that they were not uniform distributed. Since *Byte1* and *Byte2* follow the distribution of the unchanged values, this makes it difficult to detect the embedding.



(a) *Byte2* value distribution of the *FirstCluster* field (before embedding)

(b) *Byte2* value distribution of the *FirstCluster* field (after embedding)

Figure 11 FirstCluster.

For the *FirstCluster*, an uniform distribution was assumed in the approach. While this assumption holds for *Byte1*, *Byte2* was split into two groups with different probabilities, as shown in Figure 11(a). The values from 0 to 99 occurred more often than the rest, which could be due to the fact that a structure was created in the *FirstCluster*, by synthetically creating the test file systems and their files. Since Byte2 is not uniformly distributed, the embedding completely changed the distribution, as seen in Figure 11(b) and is therefore easily detectable.

5.3 Double Seconds

The two double-second parts of the *CreateTimestamp* and *LastModifiedTimestamp* fields were used as an extension to compensate for the missing 2 bits from the *Create10msIncrement* field. Here only the least significant bit of the two timestamps were written in each case. Due to the only minimal changes of these values and the randomness of the time stamps, an uniform distribution could be assumed. Since the encrypted input data is uniform distributed, the distribution of the least significant bit was not changed.

Discussion

Compared to the stego-only approach, the *exHide* approach uses metadata from deleted files to allow embedding in more metadata fields. This was the only way to achieve a higher embedding rate using only metadata to hide information. The *LastModified10msIncrement* field is not used in the *exHide* approach because Windows does not use this field when writing files to the file system. Since *exHide* provides a reasonable embedding rate even without these 6 bits, the additional embedding capacity was neglected in favor of Windows support. Therefore, if Windows support is not important, an

additional 6 bits can be used for embedding, offering various possibilities to change the method. The most straightforward approach would be to increase the embedding capacity per file metadata location to 46 bits.

To reduce detectability, we could stop using *Byte2* of *FirstCluster* for embedding, thereby reducing the embedding rate. Further research is needed to find the cause of the byte value distribution of *Byte2* shown in Figure 11(a). In order to recreate the distribution after embedding with our method, one possible approach could be the use of a mapping table to map input bytes to different output bytes and then selecting output bytes using specific weights.

6 Conclusion

In this work we gave an deep dive into the exFAT file system by displaying the processes needed to store files and folders on the file system. Additionally the layout of an exFAT partition was described and how an File Allocation Table operates. Based on our exFAT analysis, we proposed two approaches to hide data within the file system, without using anything other than the metadata entries of files.

The first approach allows for an low embedding rate of 12 bits per metadata of a file, but allows embedding in both deleted and used files. In contrast the second approach, called *exHide*, allows for an embedding rate of 5 bytes (40bits) per metadata of a deleted file. While the first approach utilizes steganographic methods, *exHide* uses techniques from the field of data hiding and therefore supports error correction to provide robustness against partial deletion of embedded data. *exHide* has stricter requirements, because an exFAT partition needs deleted metadata of files to embed data into. For the steganographic approach we did not implement error correction because we did not consider a modification of the file system after embedding.

exHide has some downsides, because steganalysis showed, that the *First-Cluster* field is not equally distributed, therefore embedding location can be detected by analysing the byte value distribution for metadata fields. Solutions to the problem were discussed, such as reducing the embedding rate by omitting the *FirstCluster* field.

Acknowledgment

This work has been funded by the German Federal Ministry of Education and Research (BMBF) in the Fraunhofer Cybersecurity Training Lab (LLCS)

and by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Charles Arthur. China and the internet: Tricks to beat the online censor. www.theguardian.com/world/2010/mar/25/china-internet-how-to-beat-censorship?intcmp=239, 2010. Accessed: 2019-05-25.

[2] Niklas Bunzel, Martin Steinebach, and Huajian Liu. Non-blind steganalysis. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–7, 2020.

[3] Niklas Bunzel, Martin Steinebach, and Huajian Liu. Cover-aware steganalysis. *Journal of Cyber Security and Mobility*, pages 1–26, 2021.

[4] Sophia Cope. Law enforcement uses border search exception as fourth amendment loophole, 2016.

[5] Ingemar Cox, Matthew Miller, Jeffrey Bloom, Jessica Fridrich, and Ton Kalker. *Digital watermarking and steganography*. Morgan kaufmann, 2007.

[6] Knut Eckstein and Marko Jahnke. Data hiding in journaling file systems. In *Digital Forensic Research Workshop (DFRWS)*, 01 2005.

[7] Sean Gallagher. Steganography: how al-qaeda hid secret documents in a porn video. https://arstechnica.com/information-technology/2012/05/steganography-how-al-qaeda-hid-secret-documents-in-a-porn-video. Accessed: 2021-05-24.

[8] Thomas Göbel, Jan Türr, and Harald Baier. Revisiting data hiding techniques for apple file system. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Loren Grush. A us-born nasa scientist was detained at the border until he unlocked his phone, 2017.

[10] Thomas Göbel and Harald Baier. Anti-forensics in ext4: on secrecy and usability of timestamp-based data hiding. *Digital Investigation*, 24:S111–S120, 2018.

[11] Julian Heeger, York Yannikos, and Martin Steinebach. Exhide: Hiding data within the exfat file system. In *The 16th International Conference on Availability, Reliability and Security*, ARES 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[12] David Kahn. The history of steganography. In *International workshop on information hiding*, pages 1–5. Springer, 1996.

[13] Sebastian Neuner, Artemios G. Voyiatzis, Martin Schmiedecker, Stefan Brunthaler, Stefan Katzenbeisser, and Edgar R. Weippl. Time is on my side: Steganography in filesystem metadata. *Digital Investigation*, 18:S76–S86, 2016.

[14] Lily Hay Newman. Mysterious 'MuslimCrypt' App Helps Jihadists Send Covert Messages. https://www.wired.com/story/muslimcrypt-steganography/. Accessed: 2021-05-24.

[15] Martin Steinebach, Andre Ester, and Huajian Liu. Channel steganalysis. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–8, 2018.

[16] Martin Steinebach, Andre Ester, Huajian Liu, and Sascha Zmuzinksi. Double embedding steganalysis: Steganalysis with low false positive rate. In *Proceedings of the 2nd International Workshop on Multimedia Privacy and Security*, pages 38–47, 2018.

[17] Martin Steinebach, Huajian Liu, and Andre Ester. The need for steganalysis in image distribution channels. *Journal of Cyber Security and Mobility*, pages 365–392, 2019.

[18] Yves Vandermeer, Nhien-An Le-Khac, Joe Carthy, and Tahar Kechadi. Forensic analysis of the exfat artefacts. *arXiv preprint arXiv:1804.08653*, 04 2018.

Biographies

Julian Heeger became a researcher in cybersecurity at the Media Security and IT Forensics department of Fraunhofer SIT, after he completed his master's degree in IT security at the Technical University of Darmstadt.

York Yannikos is a Research Associate at the Fraunhofer Institute for Secure Information Technology, Darmstadt, Germany. His research interests include digital forensic tool testing, darknet marketplaces, and open source intelligence.

Martin Steinebach. Prof. Dr. Martin Steinebach is the manager of the Media Security and IT Forensics division at Fraunhofer SIT. From 2003 to 2007 he was the manager of the Media Security in IT division at Fraunhofer IPSI. He studied computer science at the Technical University of Darmstadt and finished his diploma thesis on copyright protection for digital audio in 1999. In 2003 he received his PhD at the Technical University of Darmstadt for this work on digital audio watermarking. In 2016 he became honorary professor at the TU Darmstadt. He gives lectures on Multimedia Security as well as Civil Security. He is Principle Investigator at ATHENE and represents IT Forensics and AI Security. Before he was Principle Investigator at CASED with the topics Multimedia Security and IT Forensics.