Shrinking the gap: carving NTFS-compressed files

Recovering deleted NTFS-compressed files

By Joachim Metz <forensics@hoffmannbv.nl>
Hoffmann Investigations

Summary

An important part of digital forensic investigation is the recovery of data, particulary files. The recovery of data and files highly depends on the recovery tooling used. This paper focusses on a part of file carving where most of the currently available data and file recovery tools do not provide support for, namely recovering NTFS-compressed data.

The paper also provides an overview of the consequences of NTFS compression on data and investigative techniques. In this paper a basic understanding of NTFS is assumed. For more information about NTFS check the references.

Background

In 2006 the Digital Forensic Workshop (DFRWS) provided a challenge in which fragmented files were recovered from a randomized data image. Joachim Metz and Robert-Jan Mora developed a proof of concept tool called ReviveIt (revit, which was later renamed to revit06) [DFRWS06]. Revit uses a carving technique called Smart Carving, which is more of a conceptual technique that uses the combination of other techniques such as: file structure based carving.

The first implementation (currently known) was by Nick Mikus in foremost [MIKUS05]. This allowed foremost far better results than before when using header/footer and header/maximum (file) size carving techniques.

Revit06 provided interesting results. Far better than carving tools that used the header/footer and header/maximum (file) size carving methods like at that time. Note that today carvers like scalpel and EnCase still use this naive carving approach. This called for some research into the quality of file carvers. In late 2006 Bas Kloet was working on the subject of measuring and improving the quality of carving tools. His findings were published in his Master thesis 'Measuring and Improving the Quality of File Carving Methods' [KLOET07].

Early 2007 the DFRWS came with a new and improved file carving challenge [DFRWS2007]. This motivated Joachim Metz and Robert-Jan Mora to start working on revit07 in combination with Bas Kloet, while doing his research on the quality of file carvers. This version of revit was also sent in for the 2007 challenge.

In the meantime a lot of work has been done to improve carving tools, e.g. PhotoRec by Christophe Grennier is a very useful and qualitative carver with support for a vast amount of files and file systems.

However, in a recent investigation we needed to recover NTFS-compressed data and little of the currently available carving tools provided support for this. This paper discusses a technique to recover NTFS-compressed and its application in forensic carving tools.

Document information

Author(s): Joachim Metz <forensics@hoffmannbv.nl>

Abstract: This document contains information about recovering NTFS-compressed files.

Classification: Public

Keywords: NTFS compression, carving, file recovery

License

Version

Version	Author	Date	Comments
1.0	Joachim Metz	September 2, 2009	Initial version.

Table of Contents

1. NTFS compression

Before discussing the details of recovering NTFS-compressed data, it is necessary to provide an overview of how NTFS compression works and what its effects are on binary data.

In the digital forensics field not much has been published about NTFS compression. [SANDERSON02] provides a discussion of NTFS compression and digital forensic investigation, but does not go into details of the compression algorithm used. Also [CARRIER05] does not provide any details of the NTFS compression algorithm.

Luckily for us, the Linux NTFS project has provided for elaborate documentation about NTFS including its compression algorithm [RUSSON05].

1.1. Block-based storage

It is important to know is that NTFS uses cluster blocks[1] to store file data. These cluster blocks are commonly 4096 bytes of size. Information about the usage of these cluster blocks is maintained in the allocation bitmap and in the Master File Table (MFT), in particular the data runs.

NTFS-compressed files can consist of both compressed, uncompressed and sparse cluster blocks. NTFS uses data runs to determine which cluster blocks make up the file data. These data runs also indicate which cluster blocks are compressed, uncompressed or sparse. Without these data runs there is no other information that can provide us with this information.

1.2. Block-based compression

The block-based storage technique is also used within NTFS compression. Data is compressed at 16 cluster blocks at a time.

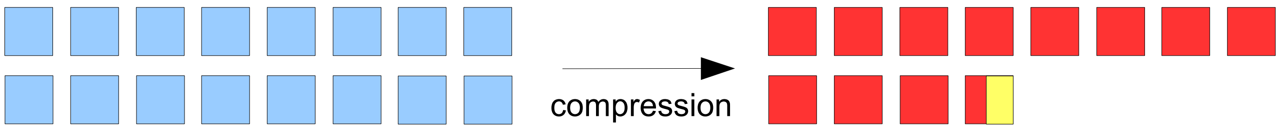

Illustration 1: Successful compression of 16 cluster blocks

If 16 cluster blocks of the original data (above in blue) can be compressed to 11.5 cluster blocks (above in red) the data is stored in 12 cluster blocks. Note that the last block contains slack space (above in yellow), which will be referred to as NTFS compression slack space. The 16 compressed cluster blocks are also referred to as a compression unit.

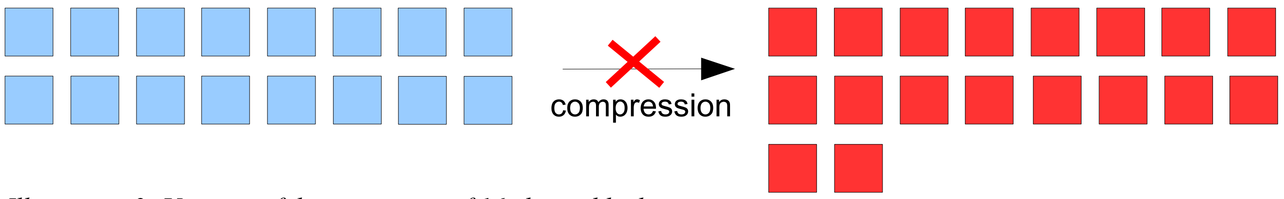

Illustration 2: Unsuccessful compression of 16 cluster blocks

If 16 cluster blocks of the original data (above in blue) cannot be compressed to fewer than 16 cluster blocks (above in red) the data is stored as the original 16 uncompressed blocks.

1 Note that in other documentation the cluster block is sometimes referred to as just cluster.

If the 32 cluster blocks would be part of a single file, the file would be stored similar to the diagram below.

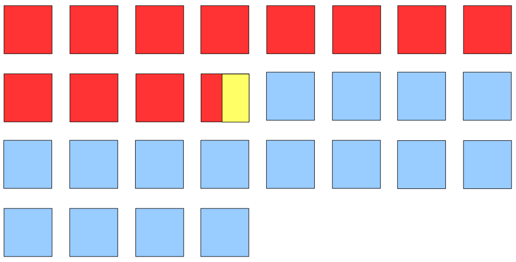
Illustration 3: 2 compression units stored sequentially

[SANDERSON02] states that files, that were initially stored uncompressed and then compressed, leave previously allocated cluster blocks in place:

```
On the face of it, it would seem logical for the compressed data from the
second compression unit to be saved to disk immediately after the first
compression unit. It is my belief that Microsoft has chosen this method for
reasons of efficiency. By leaving the saved clusters within a compression unit
initially empty means that if data is added to the first compression run then
it just needs to be expanded into the free space that was left after
compression.
```

This would mean that the compressed file would contain 4.5 cluster blocks of NTFS compression slack space.

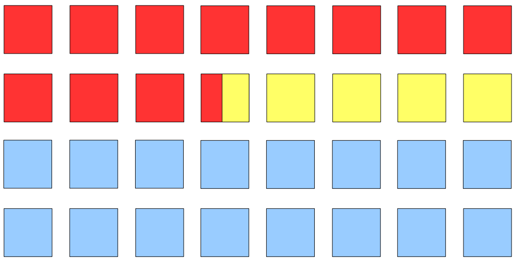
Illustration 4: 2 compression units stored sequentially with additional slack space

In our initial review of carving NTFS-compressed files we mainly found compressed units with less than 1 block NTFS compression slack space (such as in illustration 3). This could be dependent on how the compressed file was created or by the operating system and/or file system version used. Our tests were done on an forensic copy of a Microsoft Windows XP SP2 system containing NTFS version 5.1. It is unknown which operating system and version of NTFS were used in [SANDERSON02], probably Microsoft Windows 2000 with NTFS version 5.0. However more research is needed to get more conclusive answers about the allocation behavior of NTFS-compressed files.

An important aspect of carving is that an empty cluster block (a cluster block entirely consisting of zero byte values) can be stored as a sparse block. The only reference to these sparse blocks is in the NTFS data runs.

Another important aspect of for carving is that although a cluster block of data is stored within a compression unit, it still can consist entirely of uncompressed data. The compressed unit starts with a block header that contains the size of the compressed data within the block and a flag to indicate if the data within the block is compressed or not. In the illustration below the compression unit contains 2 uncompressed blocks.

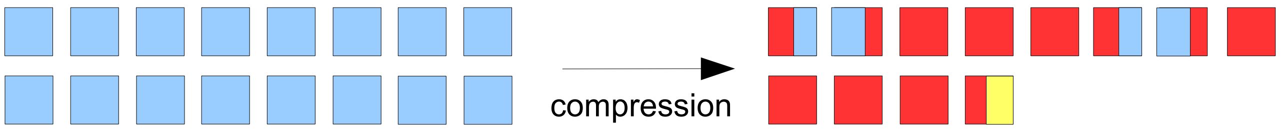

Illustration 5: a compression unit containing 2 uncompressed blocks

1.3. NTFS compression algorithm

NTFS uses a compression algorithm that is based on LZ77, this algorithm is referred to as LZNT1. The data is compressed a cluster block at a time. A 'compressed' cluster block starts with a 16-bit block header. This block header consists of the following values:

Bits	**Description**
0-11 (LSB)	Compressed data size minus 1
12-14	Unknown flags
15 (MSB)	Data is compressed

If the first block starts at offset 0, the next block starts at offset of the first block + the size of the block header + the size of the compressed data + 1, e.g. 0 + 2 + 3043 + 1 = 3046.

The block header is directly followed by data. The MSB in the block header indicates if the data is compressed or not. If the data is uncompressed, the data is stored unaltered. If the data is compressed, the data is stored in groups of 8 values preceded by a tag byte.

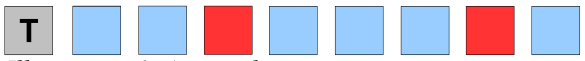

Illustration 6: A tagged compression group

Every bit in the tag byte indicates if a value is compressed or not. If the bit is set (1) the corresponding value is compressed if not set (0) the value is uncompressed. Uncompressed values are stored as bytes, compressed values are stored as a 16-bit compression tuple[1]. This compression tuple is a combination of offset and size values. The actual amount of bits used by both the offset and size values within the tuple is dependent of the amount of data already uncompressed by the algorithm. The less the amount of data already uncompressed, the smaller the offset and larger the size. The greater the amount of data already uncompressed, the larger the offset and smaller the size.

The offset value is relative from the end of the uncompressed data. The size contains a value that indicates how much bytes from the uncompressed data should be back referenced.

As long as the size is non-zero, the decompression algorithm will look for the byte at the offset in the uncompressed data and copy it to the end of the uncompressed data and reduce the size by one. Note that the offset does not change, but because the end of the uncompressed data has grown by one, the offset refers to the next byte in the uncompressed data.

To provide you with an idea of how the NTFS decompression algorithm works, consider the following compressed data:

1 Tuple meaning combination of two values.

```
00000000  00 23 69 6e 63 6c 75 64  65 00 20 3c 6e 74 66 73  |.#include. <ntfs|
00000010  2e 68 04 3e 0a 07 88 3c  73 74 64 69 01 01 80     |.h.>...stdio... |
```

In the example above the tag bytes have been made bold and the compression tuples bold and red. Below the same data is represented somewhat different.

```
00000000b '#' 'i' 'n' 'c' 'l' 'u' 'd' 'e'
00000000b ' ' '<' 'n' 't' 'f' 's' '.' 'h'
00000100b '>' '\n' 0x07 0x88 's' 't' 'd' 'i' 'o'
00000001b 0x01 0x80
```

The tag byte of the first compression group is 0, indicating that none of the values is compressed, so is the second. Therefore the values of the first and second compression group can directly be copied to the uncompressed data.

```
#include <ntfs.h
```

However, the third tag byte contains a value of 0x04, which indicated the third value is a compression tuple. This compression tuple consists of 0x8807.

For this compression tuple the amount of bits used for the offset is 5 (which results in an offset shift of 11 bits) and the amount of bits used for the size is 11 (which results in a size mask of 0x07ff). For more information about the actual calculation of the amount of bits used for the offset and size, check [RUSSON05].

The tuple consists of:

```
offset:    0x8807 >> 11     => -1 * ( 17 + 1 ) => -18
size:      0x8807 & 0x07ff =>  7 + 3           => 10
```

The tuple refers to an offset in the uncompressed data. The uncompressed values in a compression group can be added directly to the uncompressed data. Therefore, the uncompressed data we have up to now consists of the following 18 bytes:

```
#include <ntfs.h>

```

Note that the last byte is the end of line character represented above by the empty line.

Therefore the tuple (-18,10) refers to the following part of the uncompressed data.

```
#include <
```

This results in the following uncompressed data

```
#include <ntfs.h>
#include <
```

The remaining uncompressed values of the third compression group are added directly to the uncompressed data, which results in the following uncompressed data:

```
#include <ntfs.h>
#include <stdio
```

The fourth tag byte refers to another compression tuple that consists of:

```
offset:    0x8001 >> 11    => -1 * ( 16 + 1 ) => -17
size:      0x8001 & 0x07ff =>  1 + 3          => 4
```

As you might have guessed the tuple (-17,4) refers to the following part of the uncompressed data:

```
.h>
```

Eventually the uncompressed data is:

```
#include <ntfs.h>
#include <stdio.h>
```

1.4. Implications for digital forensic investigation

NTFS compression has several implications for digital forensic investigation. Strings in unallocated compressed data are not stored continuously. A string and an index-based search can only find strings of 8 bytes or less and only when the string is not separated by a tag byte.

NTFS-compressed data can be tricky to carve. Most of the currently available carvers do not handle NTFS-compressed data correctly. NTFS compression breaks most of the file structure based carving techniques like header/footer, header/embedded length and file structure based carving, but also other carving methods that assume the data is 'plain' representation of file data.

2. Carving for NTFS-compressed data

As explained in the previous chapter NTFS compression breaks several of the most commonly used carving techniques. So how to carve NTFS-compressed data? The solution is very simple: compensating for the NTFS compression.

Recently in a case we had to recover NTFS-compressed Microsoft Outlook message (.msg) files. A .msg file basically is a Message Application Programming Interface (MAPI) message stored in an OLE Compound File (aka Compound Binary File) [MICROSOFT09].

As you might know, the first 8 bytes of an OLE Compound File contain the following signature.

```
d0 cf 11 e0 a1 b1 1a e1
```

For uncompressed files these 8 bytes would be the first 8 bytes of the first cluster block of the file. However if the OLE Compound File is NTFS-compressed the signature is similar to the following:

```
09 80 00 d0 cf 11 e0 a1 b1 1a e1
```

The first 2 bytes are the compression block header, the third is the tag byte and the remaining 8 bytes are the actual signature. Because the NTFS compression algorithm uses the compression tuples to refer previously uncompressed data in the same compression block and the OLE Compound file signature consists of 8 different bytes, the 'uncompressed' signature can be found at the fourth byte in the first compressed cluster block of the file. Also note that the tag byte is always

0 for the OLE Compound File signature value.

Using revit07 we devised a carving configuration that searches for a compressed OLE Compound File signature with a tag byte at the third byte of a compressed cluster block (see appendix B). When the signature was found, revit07 was instructed to use the structure of the NTFS compression blocks to carve at least the first 16 compression blocks (compression unit). To understand the configuration note that the last compression block is always followed by a zero block header. The configuration must allow for less than 16 compression blocks for small files, which require only a smaller amount of blocks.

We decompressed these blocks by using an implementation of the NTFS decompression algorithm according to the information in [RUSSON05]. This accounted for a maximum of 16 x 4096 = 65536 bytes (64 KiB) of uncompressed data. For some of the .msg files this amount was sufficient to successfully carve the file.

Using this technique, we were able to carve near a 1000 fragments of the first 64 KiB of OLE Compound Files from the NTFS volume we were investigating.

In most cases, the first 64 KiB of a .msg file contains the RTF compressed[1] message body and the transport headers. The RTF compressed message body can be extracted uncompressed from fragments of the carved .msg files in a similar way as carving for the NTFS-compressed data.

2.1. Carving more than 64 KiB

However, 64 KiB was not sufficient for a certain .msg file. We had been able to recover the message body and the transport headers (SMTP headers in this case) but we also needed the attachments. From the transport headers it was clear that the attachments were JPEG files. When using the NTFS compression algorithm, JPEG files remain fairly uncompressed. Sometimes the first part of the JPEG file is NTFS-compressed.

Using a hex editor we were able to tell that most of the remainder of the OLE Compound File and the JPEG data it contained were stored directly after the first compression unit. This provided us with a rough end offset of the .msg file.

We devised the following approach to carve the remainder of the file.

```
While the end offset has not been reached:

          Try to decompress a compression unit (16 cluster blocks)

          Consider the decompression a success if the decompression was
          successful and ( if the end offset has not been reached and the
          uncompressed data is 16 x 4096 bytes of size or the if the end
          offset has been reached )

          If the decompression was successful copy the uncompressed data to
          the output file, otherwise copy the raw data to the output file

          If the decompression was successful, continue at the cluster block
          offset after the compression unit. Otherwise continue with the next
          cluster block
```

1 For details on the RTF compression (LZFu) see the Personal Folder File format of the libpff project

This provided us with large amount of the .msg file. Alas it was not complete. Because OLE Compound Files can have large parts of 0 byte data, this was probably caused by missing sparse blocks.

However, carving the uncompressed output file we were able to retrieve 2 of 4 the JPEG files directly from the file. Adroit Photo Forensics 2009 was even able to recover 3 of 4 JPEG files completely and 1 partially from the uncompressed output file . We finally obtained the evidence we needed.

2.2. Improving carving techniques

Although the method we used to recover most of the NTFS-compressed .msg file is a very crude one it worked sufficiently. A brute force approach to recovering NTFS-compressed data should work fairly well.

However, to improve file structure based carving techniques for handling NTFS they will need to operate on multiple levels, which fits neatly in the idea of Smart Carving, namely:

- at the lowest level the carver should handle file system characteristics e.g. like the cluster block size and NTFS compression. Handling the NTFS compression could be achieved by testing for both a NTFS-compressed as normal file signatures as provided by the next level, or decompressing cluster blocks if the higher levels indicate the file structure no longer matches the specifications.
- at the intermediate levels the carver should handle file and content characteristics, in this case the characteristics of an OLE Compound File and/or the JPEG files.
- at a highest level the carver should handle fragmentation and compensate for sparse blocks.

To create an implementation of this conceptual approach further research is needed.

3. Conclusion

NTFS compression has a high impact on how to proceed in a digital forensic investigation. However, little information is available about NTFS compression. Investigative techniques like string and index-based search are fairly limited when searching through NTFS-compressed data. Also, most of the carving tools fail to detect and recover NTFS-compressed files.

In this paper we saw that a brute force approach to decompress NTFS-compressed data works fairly well. But processing large hard disks in this manner can be time-consuming. In short, handling NTFS compression is definitely something that would be very useful if added to carving tools.

To be able to handle NTFS-compressed data within carving tools would require a multiple level (layer) approach. The lower level handling file system characteristics, intermediate levels handling the file format and the highest level handling fragmentation and compensating for sparse blocks. The implementation of this conceptual approach is a topic for another paper.

Appendix A. References

[SANDERSON02]
Title: NTFS Compression – a forensic view
Author(s): Paul Sanderson
Date: October 2002
URL: http://www.sandersonforensics.co.uk/Files/NTFS%20compression%20white%20paper.pdf

[CARRIER05]
Title: File System Forensic Analysis
Author(s): Brian Carrier
Date: 2005
ISBN-10: 0-321-26817-2

[MIKUS05]
Title: An analysis of disc carving techniques
Author(s): Nicholas A. Mikus
Date: march 2005
URL: http://handle.dtic.mil/100.2/ADA432468

[RUSSON05]
Title: NTFS Documentation
Author(s): Richard Russon, Yuval Fiedel
Date: 2005
URL: http://linux-ntfs.org/

[DFRWS06]
Title: File Carving Challenge 2006
URL: http://www.dfrws.org/2006/challenge/

[DFRWS07]
Title: File Carving Challenge 2007
URL: http://www.dfrws.org/2007/challenge/

[KLOET07]
Title: Measuring and Improving the Quality of File Carving Methods
Author(s): S.J.J. Kloet
Date: August 2007
URL: http://sourceforge.net/projects/revit/files/Documentation/Master%Thesis%20-%20Advanced%20File%20Carving/thesis.pdf/download

[Libpff08]
Title: The libpff project
URL: http://sourceforge.net/projects/libpff

[Microsoft09]
Title: [MS-OXMSG]: .MSG File Format Specification
URL: http://msdn.microsoft.com/en-us/library/cc463912.aspx

[ForensicWiki09]

Title: File Carving Taxonomy
URL: http://www.forensicswiki.org/wikwi/Carving

Appendix B. ReviveIT

```
#
----------------------------------------------------------------------------
#
# ReviveIt 2007 (revit07) configuration file
#
# Creation date:     August 9, 2009
# Modification date: August 9, 2009
#
----------------------------------------------------------------------------
#

#
----------------------------------------------------------------------------
#
# Copyright (c) 2009, Joachim Metz <forensics@hoffmannbv.nl>,
# Hoffmann Investigations. All rights reserved.
#
#
# This software is free software: you can redistribute it and/or modify
# it under the terms of the GNU Lesser General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This software is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public License
# along with this software.  If not, see <http://www.gnu.org/licenses/>.
#
----------------------------------------------------------------------------
#

#
----------------------------------------------------------------------------
#
# Identifier:                  NTFS-compressed OLE Compound File
# Category:                    binary data
# Modification date:           August 9, 2009
# References:                  ...
#
----------------------------------------------------------------------------
#

file       : ntfs_compressed_olecf
{
          extension  : inflated

          element    : compressed_olecf_signature
          {
                    pattern              : "\x00\xd0\xcf\x11\xe0\xa1\xb1\x1a"
          }

          layout
          {
                    variable   : size
                    {
```

```
                              value      : 2 little_endian
                              bitmask    : 0x0fff
                              add        : 1
                    }
                    compressed_olecf_signature

                    while size not_equals 0
                    {
                              alignment  : size
                              variable   : size
                              {
                                        value      : 2 little_endian
                                        bitmask    : 0x0fff
                                        add        : 1
                              }
                    }
          }
}
```

Appendix C. GNU Free Documentation License

Version 1.1, March 2000

```
Copyright (C) 2000  Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not