

## Data Extraction from Damage Compressed File for Computer Forensic Purposes

Bora Park<sup>†</sup>, Antonio Savoldi<sup>‡</sup>, Paolo Gubian<sup>‡</sup>, Jungheum Park<sup>†</sup>, Seok Hee Lee<sup>†</sup> and Sangjin Lee<sup>†1</sup>

<sup>†</sup>Center for Information Security Technologies, Korea University, Seoul, Korea

<sup>‡</sup>Department of Electronics for Automation, University of Brescia, Via Branze 38, Brescia, Italy

<sup>†</sup>{danver123,junghmi,gosky7,sangjin}@korea.ac.kr, <sup>‡</sup>{antonio.savoldi,paolo.gubian}@ing.unibs.it

### Abstract

*Nowadays compressed files are very widespread and can be considered, without any doubt, with regard to the Digital Forensic realm, an important and precious source of probatory data. This is especially true when in a digital investigation the examiner has to deal with corrupted compressed files, which have been gathered in the collection phase of the investigative process. Therefore, in the computer forensic field, data recovery technologies are very important for acquiring useful pieces of data which can become, in a court of law, digital evidence. This kind of technology is used not only by law enforcement, but also by the multitude of users in their daily activities, which justify the relevant presence of tools in the software market which are devoted to rescue data from damaged compressed files. However, state-of-the-art data recovery tools have many limitations with regard to the capability of recovering the original data, especially in the case of damaged compressed files. So far, such recovery tools have been based on a schema which controls the signature/header of the file and, thus, provides the offset to the raw compressed data block. As a result, they cannot recover the compressed files if the first part of the raw compressed data block, which pertains to the header, is damaged or the signature/header block is corrupted. Therefore, in order to deal with this issue, we have developed a new tool capable of rescuing damaged compressed files, according to the DEFLATE compression scheme, even though the header block is missing or corrupted. This represents a new interesting opportunity for the digital forensic discipline.*

### 1: Introduction

Recovering damaged files is a very important and fundamental topic in computer forensic discipline. This fact can be easily understood by having a look at the field of digital forensics [5] and by considering that every source of potential digital evidence need to be examined, even though we are dealing with corrupted digital compressed files. Hence, plenty of different methods for recovering data from damaged files, compressed and not, have been presented. Undoubtedly, it is quite tough to recognize the original uncompressed file type starting with the analysis of the corrupted one. This fact rises from the high level of entropy which is a well-known feature for the category of compressed files. So far, in the field of proprietary as well as in the open source market there are some recovery tools, which are able to rescue damaged compressed files, even though, as will be made more clear later, only for a specific subclass of cases. Indeed, these tools cannot deal with corrupted files whose header section block is damaged. Thus, they do not allow the determination

---

<sup>1</sup>Corresponding author: Sangjin Lee. Email: sangjin@korea.ac.kr, PhoneNumber: +82-19-328-5586

of the offset to the so-called raw data block, which defines the compressed block within the compressed file. Interestingly, as will be clearer in the following sections of this paper, we managed to solve, at least partially, this problem, by implementing an innovative solution which permits the recovery of a damaged compressed file even though the first block is corrupted. Our approach, which has been named *bit-by-bit* processing, acts at the level of the raw compressed data block, by removing, gradually, the corrupted header, thus permitting to regain the data block which can be subsequently decompressed. In this paper, a new data recovery method, which can be used to recover damaged compressed files, is proposed.

The remaining part of the paper is organized as follows. Firstly, a detailed and comprehensive explanation of common compression and decompression algorithms will be provided, giving at the same time information about the structure of the compressed blocks. Secondly, a review of state-of-the-art data recovering algorithms for rescuing damaged compressed data will be described. Afterwards, our original method for recovering corrupted compressed files, especially those having a damaged header, will be discussed, also detailing the advantages and disadvantages of the solution and proving the effectiveness of the method. Finally, we will sketch our future work plans.

## 2: Related Work

Before describing the related work and going ahead with the main topic of this paper, some fundamental terminology has to be introduced.

- **Raw compressed data:** it refers the purely compressed data, without any header block and which can be obtained with a specific compression algorithm.
- **Compressed data:** it indicates generic compressed data which has been obtained by using a specific algorithm (tool). The resulting file has its own structure with a header which is normally required by the decompressor to reconstruct the original data.
- **Decompressor:** it defines any algorithm for reconstructing the uncompressed original data.
- **Damaged data:** it refers to any compressed file which has been altered in some part.

### 2.1: Data Recovery Algorithms: State-of-the-art

Current data recovery tools are able to recover a specific category of damaged compressed files. Specifically, those compressed files with an integral (not altered) header will be able to be recovered. Figure 1 illustrates an example of damaged compressed file according to the 'ZIP archive' file [10] format, with the case of a damaged header and with a normal header. As a matter of fact, among corrupted data blocks, only (F), (G) and (H) will be able to be recovered. This fact is a logical consequence of the current approach which is used for data rescuing.

We can analyze how the ZIP recovery algorithm works. It tries to find the local header, in the form 0x504B0304, within the compressed file. If such a signature is found, then the algorithm will read the offset to the raw compressed data blocks and, consequently, it will rearrange the ZIP archive to the correct format. As we mentioned earlier, this kind of approach works only for those files which do not have a damaged header. So far, with regard to Figure 1, if block (A) is damaged such approach will be certainly not effective, as it is not able to identify correctly the signature or the offset for the raw data blocks. Thus, it is clear now that current state-of-the-art approach for damaged compressed data recovery uses the information contained in the header. As a consequence, to deal with this great limitation, especially from a forensic perspective, we have suggested and

Damaged header (A)	
Damaged Raw Data	1 <sup>st</sup> Block (B) (damaged)
	2 <sup>nd</sup> Block (C) (damaged)
	3 <sup>rd</sup> Block (D)
Normal header (E)	
Normal Raw Data	1 <sup>st</sup> Block (F)
	2 <sup>nd</sup> Block (G) (damaged)
	3 <sup>rd</sup> Block (H)

**Figure 1. Data blocks which can be recovered by using current tools.**

implemented a new method for extracting raw compressed data blocks, from damaged compressed files, without having any header information. It is important to point out that after obtaining the raw compressed data blocks, it will be easy to reorganize them according to a certain compression scheme. Even though there will be a damaged raw data block, it will certainly be possible to recover a part of the original file, which is extremely important in a digital forensic context.

## 2.2: Limitations of Current Data Recovery Tools

As already mentioned, current ZIP recovery tools are not able to recover the (B), (C) and (D) raw data blocks when the file header is damaged. This is because they cannot find the initial part of the raw compressed data whose offset, necessary to point to the raw data blocks, is written in. In addition, even considering the case of an integral header, there could be other problems. Indeed, if raw data blocks are dependent, as it will be detailed later, it will be impossible to recover the raw data block section (e.g. blocks (G) and (H)). This because of the interdependency features of the compressor algorithm.

Since the purposes of a computer forensic investigator are to regain the original data or, at least, a part of that data, which can be used to identify the kind of material under examination, it would be really useful to be able to decompress some part of the compressed data, even when dealing with heavily damaged compressed files. Before providing all the necessary details about our data recovery method, we will illustrate the algorithm for data compression and decompression.

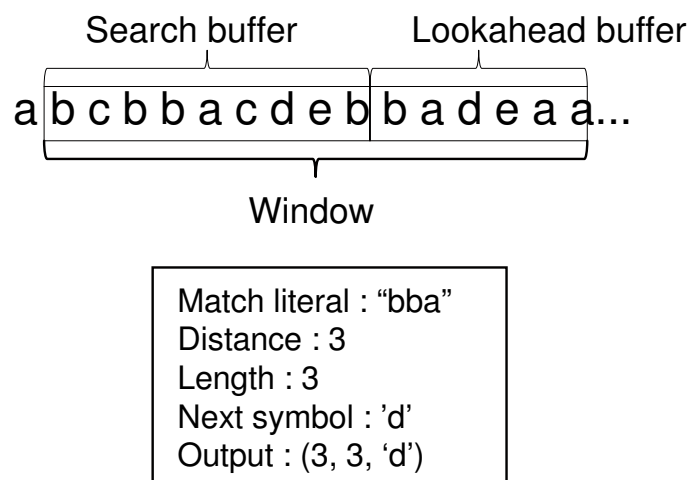
## 3: Compression and Decompression Algorithms

According to user statistics, among the most common compression tools used by plenty of computer users, we can mention WinZip, Alzip and WinRAR [1]. These applications support various compressed formats and are mainly used for archiving purposes. Interestingly, the main compression algorithm for WinZip and Alzip is the so-called *DEFLATE* whereas WinRAR uses a modified version of this algorithm. For example, .zip, .gz and .alz are extensions of the mentioned compression algorithms which use the *DEFLATE* data compression algorithm. The dual algorithm for decompression is known as *INFLATE*. Since the *DEFLATE* and *INFLATE* algorithms are very

common among compression utilities, we have chosen them to test our damaged compressed file recovery methodology.

### 3.1: Basic Principle of DEFLATE Compression Algorithm

The DEFLATE compression algorithm [4] is based on a variation of LZ77 [6] combined with an Huffman encoder scheme. The principle of LZ77 is to use part of the previously-seen input stream as the dictionary. That is, the stream which is used more than one time is written on compressed data as an offset of the previous stream and with the length of it. The encoder maintains a window over the input stream and shifts the input in that window from right to left as strings of symbols are being encoded. Thus, the method is based on a sliding window. The window is divided into two parts. The part on the left is the so-called *search buffer*, which represents the current dictionary, and includes symbols that have recently been input and encoded. The part on the right is the so-called *look-ahead buffer*, which contains text yet to be encoded [1] [6]. It is interesting to mention that in practical implementations the search buffer is some thousands of bytes long, whereas the look-ahead buffer is only tens of bytes long. An example of LZ77 compression is shown in Figure 2.



**Figure 2. LZ77 algorithm.**

A compressed data set consists of a series of blocks, corresponding to successive blocks of input data. The block sizes are arbitrary, except that non-compressible blocks are limited to 65,535 bytes. Each block is compressed using a combination of the LZ77 algorithm and Huffman coding. The Huffman trees for each block are independent of those for the previous or subsequent blocks; the LZ77 algorithm may use a reference to a duplicated string occurring in a previous block, up to 32K input bytes before. Each block consists of two parts: a pair of Huffman code trees that describe the representation of the compressed data part, and a compressed data part. (The Huffman trees themselves are compressed using Huffman encoding.) The compressed data consists of a series of elements of two types: *literal bytes* (of strings that have not been detected as duplicated within the previous 32K input bytes), and *pointers* to duplicated strings, where a pointer is represented as a pair (length, backward distance). The representation used in the DEFLATE format limits distances to 32K bytes and lengths to 258 bytes, but does not limit the size of a block, except for non compressible blocks, which are limited as noted above. Each type of value (literals, distances,

and lengths) in the compressed data is represented using a Huffman code, using one code tree for literals and lengths and a separate code tree for distances. The code trees for each block appear in a compact form just before the compressed data for that block.

Thus, *literal bytes*, (*length,backward distance*) and *next literal* are the basic parts of the so-called compression block, the basic atom in the LZ77 compressed data file. In the case of DEFLATE compression, only *literal bytes* and *length,backward distance* parts are present and Huffman scheme is applied as an additional step. Huffman coding is a popular method of data compression. This scheme is based on the frequency of occurrence of a character. The main principle of Huffman coding is to use a lower number of bits to encode the data that occurs more frequently. The algorithm starts by building a list of all the alphabet symbols in descending order of their probabilities. It then constructs a tree, with a symbol at every leaf, from the bottom up. This is done in steps, where at each step the two symbols with smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing the two original symbols. When the list is reduced to just one auxiliary symbol, the tree is complete. The tree is then traversed to determine the codes of the symbols [1] [8].

### 3.2: Types of Compression Modes in DEFLATE Algorithm

In DEFLATE compression algorithm there are 3 modes for data compression. The first one, 'Mode 1' does not compress data. The second, 'Mode 2' compresses data by using LZ77 and a fixed Huffman coding, according to a fixed table within the compression algorithm. The last one, 'Mode 3', compresses data with LZ77 and a dynamic Huffman coding. In mode 3, the encoder generates two code tables, which are located after the header of the compressed file. After that, it uses the tables to encode the data that constitutes the compressed raw data block. The format of each block [1] is detailed in Figure 3.

Mode 1
3-bit Header ("000" or "100")
Data Area (up to 65,535 bytes)
LEN (unsigned 16-bit numbers) data bytes

Mode 2
3-bit Header ("001" or "101")
Literal/Length prefix codes and distances prefix code
Code 256 (EOB)

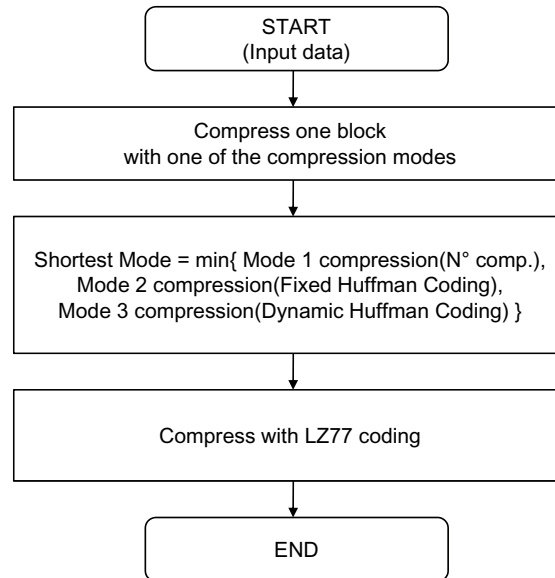
  

Mode 3
3-bit Header "010" or "110"
Table information
Compressed Data(Encoded with the 2 prefix tables)
Code 256(EOB)

**Figure 3. Format of each block in DEFLATE compression algorithm.**

The encoder compares lengths for each compression mode and then compresses data by selecting

the shortest length of the compressed data [9]. A simplified DEFLATE compression algorithm is shown in Figure 4.



**Figure 4. Simplified DEFLATE algorithm.**

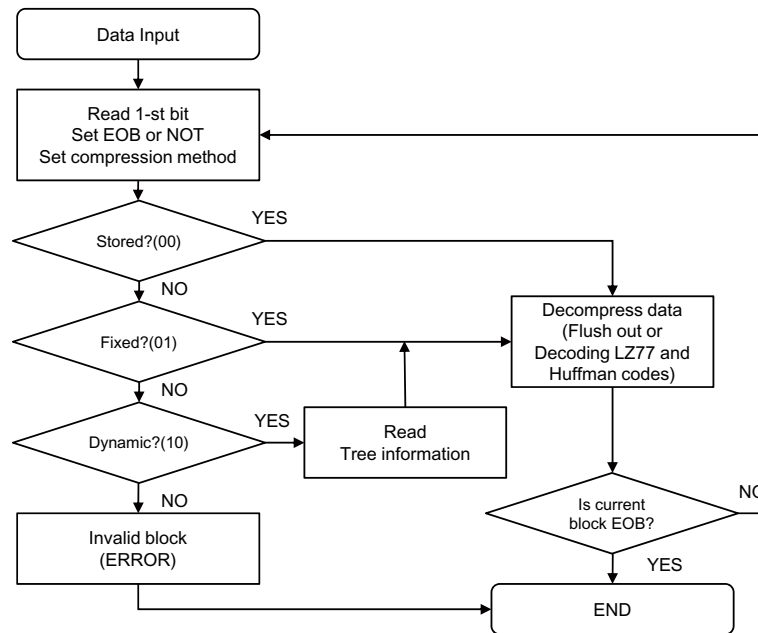
In order to decompress or rescue damaged compressed files, we have to recognize the format of compressed data 3. The block in Mode 1 is composed of an header which signals the decoder (also referred to as decompressor) which is the compression mode and the raw compressed data section. Mode 2 is composed of an header, like Mode 1, and of a raw compressed data part. Mode 3 is composed of an header, an Huffman table, and, finally, a raw compressed data block [8].

### 3.3: Basic Principle of INFLATE Decompression Algorithm

It is worthy to mention that a DEFLATE stream consists of a series of blocks. Each block is preceded by a 3-bit header, specified as follows. The 1st bit asserts whether we have the last block in stream sequence. A value of '1' specifies that this is the last-block in the stream whereas a '0' indicates that there are more blocks to process after this one. The next 2 bits refer to the encoding method used for the current block, which specifies the encoding mode:

- 00: a stored/raw/literal section follows, between 0 and 65535 bytes in length (Mode 1).
- 01: a static Huffman compressed block, using a predefined Huffman table (Mode 2).
- 10: a compressed block with a dynamic Huffman encoding scheme (Mode 3).

We can briefly detail how the INFLATE algorithm, the decompressor, works. If the data stream has been compressed by Mode 1, the decompressor just flushes out the data. In the case of Mode 2, initially, as a first step, the decompressor decodes the Huffman table, built within the algorithm, as already mentioned, and, then, as a second step, it decodes the LZ77 blocks. Finally, for Mode 3, the decompressor reads the Huffman tree information within the current block and then decodes Huffman and LZ77 blocks. When the decompressor meets an EOB string, it stops the decoding phase. Figure 5 illustrates the mentioned algorithm.



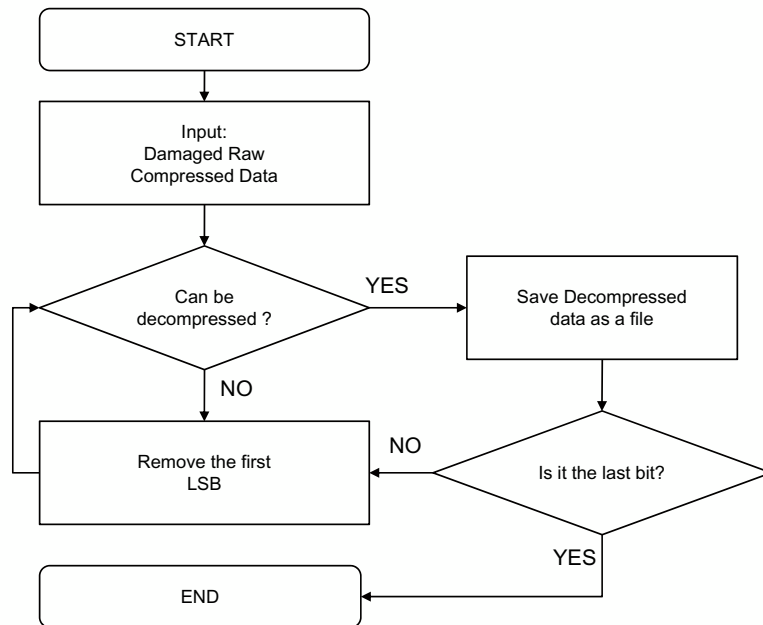
**Figure 5. Simplified INFLATE algorithm.**

#### 4: A New Method for Rescuing Damaged Raw Compressed Files

The main goal of this section is to provide a solution for decompressing and recovering raw data streams. At the end of the decompression phase we will have to deal with many chunks of data, according to the compression scheme and thus we will be able to apply a further analysis step toward the reconstruction of the original uncompressed material, final phase of the forensic analysis. Even though it will not be possible to recover the original uncompressed file, by using data carving techniques it will be certainly worthy to extract as much information as possible from these pieces of data.

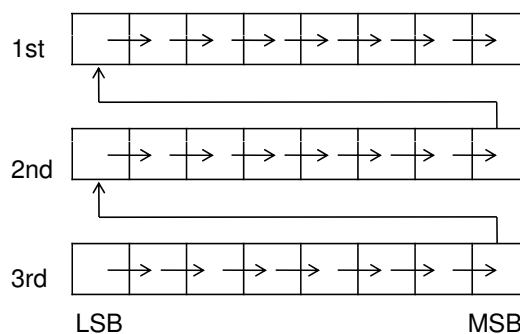
So far, in mode 2, the Huffman table is located at the level of the DEFLATE compression algorithm, whereas in mode 3 the Huffman table is read within the compressed file, according to the layout shown in Figure 3. Therefore, if the first part of the Huffman table is damaged, in the case of mode 3, it will not be possible to decompress that block, because of the features of the dynamic Huffman coding scheme.

However, if we consider the compression scheme for mode 2, we can agree that even though the header is corrupted, it will be likely possible to recover the subsequent raw compressed data streams. This follows from the knowledge of the fixed Huffman table, which is directly coded within the compression algorithm. Interestingly, as already pinpointed, previous recovery tools, which deal with the DEFLATE compression scheme, try to find the header and, by reading the offset information, try to sort out the compressed raw data blocks. Our approach acts at the bit level by considering mode 2. Even though the header is corrupted, it is certainly possible to remove one bit per time, to apply the INFLATE decompression algorithm and, as a result, decide whether we are able to obtain any decompressed block. This *bit-by-bit* process can be reiterated as many times as possible, according to the length of the compressed stream. The detailed algorithm of the *bit-by-bit* decompression is shown in Figure 6.



**Figure 6. Damaged raw compressed data recovery process.**

When given a damaged input file, our algorithm acts by trying to decompress, according to the LZ77 scheme, the raw data block. If the decompression can be done, then the resulting data block is saved as a file (chunk). Conversely, if the input data can not be decompressed, the next bit is removed and the process is repeated until the end of the block (EOB string). The method of *bit-by-bit* is shown in Figure 7.



**Figure 7. Bit-by-bit algorithm.**

Since data blocks are packed as bytes, e.g. starting with the least-significant bit [9], the process of removing bits can be represented like in Figure 7. Firstly, the 8th bit is removed from the first byte of data and then the remaining part of the data stream is shifted like in a register. The process is repeated as many times as necessary while the block is decompressed or the end of the block is reached.



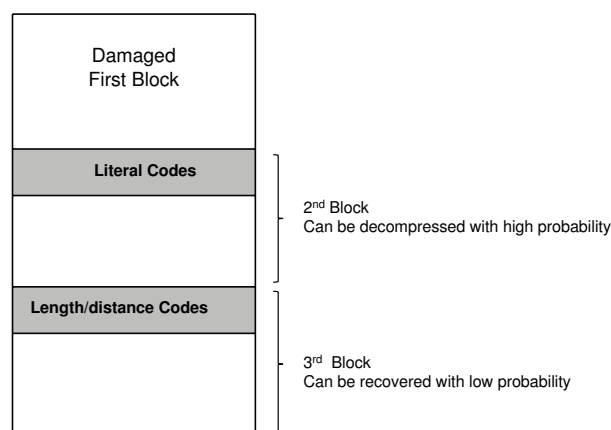
#### 4.1: Damaged Raw Data: Possible Scenario

We have already detailed the basic structure of compressed data blocks. These blocks can be sometimes independent or dependent among each other. Normally, this interblock dependency does not permit the recovery of a single raw compressed data block (data stream). The mentioned dependency is obtained by acting on a specific parameter of the DEFLATE compression algorithm, as follows.

- `ret = DEFLATE(&strm, 'flush value')`

The *flush value* determines the degree of dependency among different blocks. More precisely, the flush value signals the end of a data compressed block pertaining to a certain file (boundary block), which has been compressed. A possible value for such a flag is `Z_SYNC_FLUSH`, which creates a special NULL block to define the boundary among different blocks. Conversely, when the flush value is set to `Z_FULL_FLUSH`, then the whole set of blocks in the compressed file is independent. That is, even though there may be some damaged compressed data blocks which have been created with the DEFLATE algorithm using `Z_FULL_FLUSH`, the entire set of compressed data blocks can be recovered successfully.

As a matter of fact, by analyzing the binary compressed data there is no information to tell whether the `Z_FULL_FLUSH` flag has been applied. Thus, most of the damaged compressed files look impossible to recover. However, if some raw compressed blocks start with *literal bytes* LZ77 codes, it will be possible to decode the compressed data block by using the INFLATE algorithm (only for compression mode 2). This feature strictly depends on the characteristics of the original uncompressed data. In some cases, there could be some compressed blocks which cannot use the characters previously seen (literal bytes) to create the new compressed data stream. That is, if the uncompressed data has a low compression ratio, according to the LZ77 algorithm, the block independency will be assured even if '`Z_FULL_FLUSH`' has not been set. In this case, a dependent compressed block looks like an independent compressed block. Thus, at least partially, it should be possible to recover some compressed data blocks even though there are dependencies among blocks. Figure 8 details what it has already been described.



**Figure 8. Block recovery.**

In Figure 8, the first raw data block is damaged. The first part of the second block specifies *literal bytes* information, whereas the first part of the third block contains length and distance information.

As stated, when we find such an information block it means that the block does not depend on the previous one, according to the coding scheme. As a consequence, since the second block is independent from the first, it can be likely decompressed. However, in the case of the third block, its initial part is referred to as literal/distance information block, which pertains to a sequence of stream blocks of a certain file being compressed. Thus, in such a case, it is unlikely that the raw compressed data block can be recovered.

#### 4.2: Decompression Helper

Suppose that a damaged ZIP file has been found on a binary hard disk image. This file has no ZIP header, and thus there is no way to know where the first part of the raw compressed data is located. Moreover, if we try to apply the decoding algorithm, the data will not be decompressed. Therefore, we could conclude that the first part of this raw compressed file is damaged. Thus, this corrupted file could not be decompressed or recovered. However, by using the method proposed in this paper, the *bit-by-bit* processing approach, some data blocks might be recovered. The main algorithm has been implemented in a software tool shown in Figure 9, which is able to recover chunks of raw data blocks, belonging to a certain file, or, more interestingly, even entire damaged files.

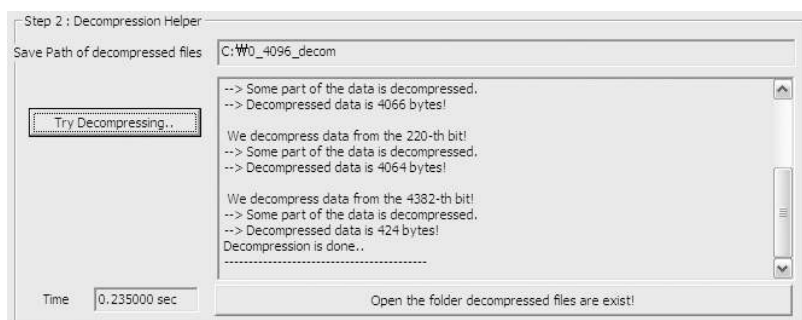


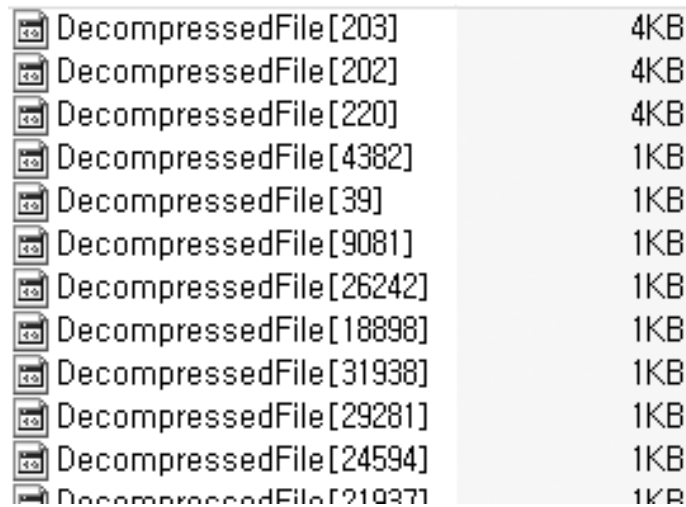
Figure 9. Screenshot of Decompression Helper

The results of the execution of *Decompression Helper*, the name of the tool we have implemented, are shown in Figure 10. In that Figure, we can see the decompressed data chunks which have been successfully decompressed. The *decompressed[i]* is referred to as the decompressed data block which starts from the *i*-th bit in the original damaged data file. As already explained, all chunks are the uncompressed stream, or data block, of the original file. Clearly, the next phase of the work is to sort out different chunks and to figure out important details about the data content.

As already pointed out, even one single part of a file can be decisive in a forensic investigation. A forensic practitioner, by using our proposed recovery algorithm, based on the so-called *bit-by-bit* processing, should be able to discover the contents of a fragment of compressed file, according to the DEFLATE scheme, recovered, for example, from a memory dump, in the context of a volatile forensic memory analysis scenario. The important thing to understand, is that at the moment, our approach is the only one which tries to automate this recovery process, which could be very hard if done manually.

Table 1 shows how the proposed tool can be helpful. The only assumption is that the first block of the compressed data is always damaged.

In addition, Table 1 shows which damaged compressed data blocks can be recovered and which can not. We have tested our solution on plenty of corrupted compressed files, which have been created ad hoc by removing or altering the header block, of different size, in the range of 100 bytes

A screenshot of a file explorer window showing a list of decompressed files. Each file is represented by a small icon of a document with a magnifying glass, followed by the filename and its size. The files are: DecompressedFile[203] (4KB), DecompressedFile[202] (4KB), DecompressedFile[220] (4KB), DecompressedFile[4382] (1KB), DecompressedFile[39] (1KB), DecompressedFile[9081] (1KB), DecompressedFile[26242] (1KB), DecompressedFile[18898] (1KB), DecompressedFile[31938] (1KB), DecompressedFile[29281] (1KB), DecompressedFile[24594] (1KB), and DecompressedFile[21037] (1KB).

DecompressedFile[203]	4KB
DecompressedFile[202]	4KB
DecompressedFile[220]	4KB
DecompressedFile[4382]	1KB
DecompressedFile[39]	1KB
DecompressedFile[9081]	1KB
DecompressedFile[26242]	1KB
DecompressedFile[18898]	1KB
DecompressedFile[31938]	1KB
DecompressedFile[29281]	1KB
DecompressedFile[24594]	1KB
DecompressedFile[21037]	1KB

**Figure 10. Decompressed files.**

to 10 Mbytes. One important fact is the decompression (recovery) time, which strictly depends on the size of the file being recovered. So far, we have experimentally determined that the extraction time is linearly dependent on the damaged file size. Moreover, according to the range in size previously specified, we can expect an extraction time in the range 15 seconds - 15 minutes (100 bytes, 10 Mbytes). Clearly, this is a worst-case scenario. Indeed, in future versions of the tool we will optimize the extraction algorithm by improving, as a consequence, the computation time.

As already stated, the main result of our tool is the set of uncompressed data blocks which will be used, in a further analysis step, to reconstruct the original file.

Table 2 illustrates the main differences between current state-of-the-art recovery tools, which are able to deal with the DEFLATE/INFLATE compression scheme, and the one we have created, Decompression Helper.

## 5: Experimental Results

In our experiment we considered 100 corrupted ZIP files, which were obtained from allocated and unallocated hard disk space. Moreover, as can be seen in Table 3, there are two groups of damaged files: one with header information and the other with raw compressed data.

The result of the experiment obtained by using the new method stated in this paper is shown in Table 4.

Considering damaged ZIP files which come from allocated space, it can be seen that there are 13 files whose raw data section has been damaged. After recovering 8 files, it can be noticed that 5 of them are ZIP files and 3 are text-based files. The results are illustrated in Figure 11.

## 6: Conclusion and Future Works

In this paper we have illustrated a comprehensive and effective method which can be used to rescue a corrupted compressed file according to the DEFLATE compression algorithm. It has been pointed out that such method is totally new in the realm of recovery tools and, interestingly, it can

**Table 1. Different cases that can be treated by ‘Decompressor Helper’.**

Single or multiple block	Order of block	Compression mode	LZ77 code at the first part of block	Can be decompressed or not
Single	(1st)	1		Can be compressed with low probability
Single	(1st)	2		Can be compressed with low probability
Single	(1st)	2		Can be compressed with low probability
Single	(1st)	3		No
Multiple	1st	1		Can be compressed with high probability
Multiple	1st	2		Can be compressed with low probability
Multiple	1st	3		No
Multiple	not 1st	1		Can be compressed or not
Multiple	not 1st	2	Maximum distance is smaller than the Current offset of the current block	Yes
Multiple	not 1st	2	Maximum distance is greater than the Current offset of the current block	No
Multiple	not 1st	3	Maximum distance is smaller than the Current offset of the current block	Yes
Multiple	not 1st	3	Maximum distance is greater than the Current offset of the current block	No

**Table 2. Comparison results between previous recovery tools and ‘Decompression Helper’.**

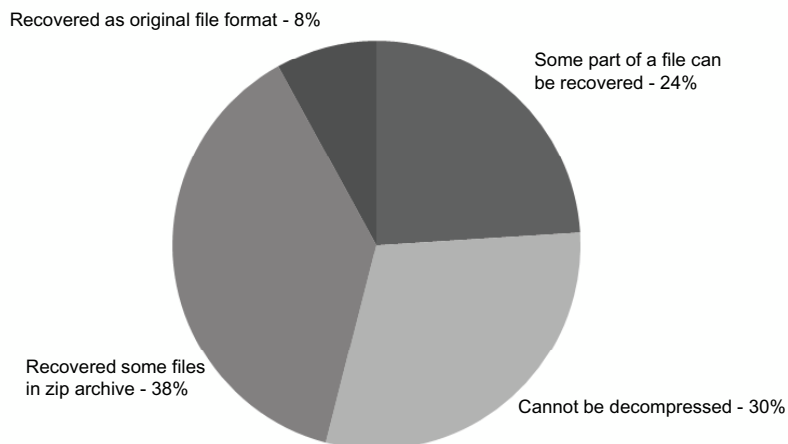
		Preexistence recovery tool	Decompression Helper
Can get raw compressed data without the file header?		No	Yes
In case that only raw compressed data are available, is it possible to decompress raw compressed data?		No	Yes
In case that first block is damaged is it possible to decompress the rest of the blocks		No	Sometimes Yes
In case of Single block, if the first part of the compressed data is damaged, can it be decompressed?	Fixed block	No	Sometimes Yes
	Dynamic block	No	No

**Table 3. Distribution of experimental files**

Properties of damaged compressed data			Quantity (#)
ZIP files from allocated space	Damaged Header	Damaged Signature	17
		Damaged offset information	12
		Damaged footer	8
	Damaged Raw compressed data		13
ZIP files from unallocated space	Damaged Header	Damaged Signature	7
		Damaged offset info.	16
		Damaged footer	3
	Damaged Raw compressed data		24

**Table 4. Experimental Results**

Properties of damaged compressed data		Quantity (#)	Recovery Rate (%)	
ZIP files from allocated space	Damaged Header	Damaged Signature	17	100%(17)
		Damaged offset information	12	100%(12)
		Damaged footer	8	100%(8)
	Damaged Raw compressed data		13	61%(8)
ZIP files from unallocated space	Damaged Header	Damaged Signature	7	100%(7)
		Damaged offset info.	16	100%(16)
		Damaged footer	3	100%(3)
	Damaged Raw compressed data		24	66%(16)



**Figure 11. Recovery results in case of damaged raw data(2)**

be profitably used by digital forensic practitioners for dealing with data recovery from live memory dumps. After having described the DEFLATE compression algorithm, we have approached the problem of recovering corrupted compressed files with a damaged or missing header, by detailing why it is worthwhile to have a such tool and illustrating some differences of currently used recovery algorithms. As a further step in the research, we would like to optimize the extraction time and define a better strategy to identify boundary compressed blocks in the case of interblock dependencies.

## 7 Acknowledgments

This work was supported by the IT R&D program of MKE/IITA. [2007-S019-02, Development of Digital Forensic System for Information Transparency].

## References

- [1] D. Salomon, *Data Compression - The Complete Reference - 4th edition*, Springer-Verlag, 2007
- [2] M. Nelson, *The Data Compression Book - 2nd edition*, MIS Press Inc., 1996
- [3] K. Sayood, *Introduction to Data Compression - 2nd edition*, Academic Press, 2000
- [4] G. Roelofs, *PNG: The Definitive Guide*, O'Reilly, 1999

- [5] G. Palmer, *A Road Map for Digital Forensic Research*, Tech. Rep. DTR-T001-0, Utica, New York, 2001
- [6] G. Langdon, *A Note on the Ziv-Lempel Model for compressing Individual Sequences*, IEEE Transactions on Information Theory. IT-29(2):284~287, 1983
- [7] S. Klein and Y. Wiseman, *Parallel Lempel Ziv Coding*, Discrete Applied Mathematics, Vol. 146(2):180~191, 2005
- [8] *ZLIB Compressed Data Format Specification version 3.3*, RFC 1950, 1996, <http://www.ietf.org/rfc/rfc1950>
- [9] *DEFLATE Compressed Data Format Specification version 1.3*, RFC 1951, 1996, <http://www.ietf.org/rfc/rfc1951>
- [10] *appnote.txt - ZIP file format specification*, PKWARE Inc.