Decompression of Run-Time Compressed PE-Files

Miroslav Vnuk, Pavol Návrat

Slovak University of Technology

Faculty of Informatics and Information Technologies

Ilkovičova 3, 842 16 Bratislava, Slovakia

Abstract: In this paper, we tackle the problem of compression and decompression of PE-files (portable executable file format in MS Windows). We analyze decompression methods used in the PE-file unpackers. Based on that, we design a new decompression method, which however uses standard decompression algorithms. Finally, we compare various standard decompression methods to our method and analyze and comment the results that we were able to achieve.

Miroslav Vnuk received his BS. and Ing. (Master) degrees in Software Engineering from Faculty of Informatics and Information Technologies at Slovak University of Technology, Bratislava in 2004 and 2006. He currently works for Eset, Slovakia (the developer of NOD32 antivirus solution) as software developer. His research and professional interests include virus problematic, data mining (particularly in using to detect malware, etc.), software design and design patterns and software development.

Pavol Návrat received his Ing. (Master) cum laude in 1975, and his PhD. degree in Computing Machinery in 1984 both from from Slovak University of Technology. He is currently a professor of Informatics at the Slovak University of Technology. His research interests include related areas from Software Engineering, Artificial Intelligence, and Information Systems. Prof. Navrat is a Fellow of the IEE and a Senior Member of the IEEE and its Computer Society. He is also a member of the ACM, American Association for Artificial Intelligence, Association for Advancement of Computers in Education, Slovak Society for Computer Science and Slovak Artificial Intelligence Society

1. Introduction

One way that malware writers have been using for many years to disguise their malicious software creations, is to use some compression program. These programs (UPX, ASPack, FSG etc.) typically take a Windows PE-file (or another platform executable file) and compress it. File compressed in such a way decompresses itself in memory at runtime.

The PE-files compression presents two problems for antivirus engines. First, to detect a known malware, the file has to be decompressed first to reach the point where signature matching can occur. Second, it is necessary to decompress the file so that strong code analysis heuristics could be applied.

There are different methods that can be used to decompress these files to the point at which they can be analyzed further. They have both advantages as disadvantages. There is definitely a room for an improvement. That is why we attempted to devise new decompression method.

Reviewing the current state of art, there is not much literature on the subject. There are works on compressing executable files (e.g., [2], [3]) and even some works on decompression methods (e.g., [1], [4]) but strictly speaking, they do not tackle the problem we are attemtping to do. One reason may perhaps be that the developers of compression/decompression programs prefer not to reveal details of methods the programs are based on.

1.1. Analysis. Stand-alone unpackers

Some compression programs (packers), such as UPX, use their own decompression program (unpacker) included within themselves. Within a variety of all packers, they are a minority.

Stand-alone unpackers can be useful for research purposes, but they are not appropriate for incorporating into mainstream antivirus products. Possibly, they could be a useful add-on for free antivirus solutions.

1.2. Running and dumping

Another technique that is quite successful in decompression of run-time packed executable files, is running the code and then using a utility (such as Procdump, SoftIce, etc.) to capture the in-memory image (which is at certain moment unpacked) and saving it on a disc. Again, similarly to malware packed files, this is really only useful for research purposes, and this needs to be done in a controlled environment.

Disadvantage of this method is that it does not know the right point when a running program is to be stopped and then its image captured in the memory.

1.3. Single-purpose decompression programs

This is the most frequently used method of decompression of run-time packed files (not only PE-files).

This method works well when a packer uses the same unpacking code each time.

One possibility to write this decompression program is to disassemble a sample of a packed file and just copy the bulk of the resulting assembly language code and edit some parts of this code to be able to self-run (i.e., not run as part of compressed file) with a compressed file as input.

The resulting code will run quickly. However, the generated code will be processor-specific, and therefore in order to run on many different processors it will need to be patched to the correct form.

Duplicating the original code is not a safe strategy. A typical packer has a minimal error checking. If code from this packer is used to unpack the file, the product can overflow the memory allocated, which may eventually crash the process, crash the entire system, or allow an attacker to gain a complete control of the system. The solution of this most important problem is to encapsulate the code within a defensive environment. Another solution would be to add many error-checking controls (buffer underflow / overflow, etc.). Moreover, if there are used in the code some system calls, these must be substituted by own platform or system independent function calls.

Because of specific code is needed for each packer (and perhaps for each version of each packer), the size of the antivirus product will also grow quite large.

1.4. Code emulator

Generic code emulation is a very powerful decompression method. To program a code emulator is not simple. But once we have it, this can greatly speed up the process of adding new unpackers. Often, part of the process of adding a new packer can be skipped, because the emulator can decompress a file automatically.

The code emulator loads a program into a virtual environment and then runs until the file is decompressed– a point which is defined heuristically, not algorithmically. How the heuristic defines a state when a file is decompressed is the most difficult part of decompression by a code emulator.

An interesting thing is that once an emulator is able to cope with anti-debugging tricks used by one unpacker, it can cope with the tricks in PE-files compressed by any other packer.

Code emulation is slower than other decompression methods. This is because the code emulator always has to maintain the entire CPU state.

1.5. Mixed code emulator + specific routines

For particularly tricky packers, it might be a good strategy to mix emulation and specific routines. For instance, emulation might take place until polymorphic encryption key has been found. Then the particular encrypted data can be decrypted by a specific routine. Its use is faster then emulation. Mixing code emulator with specific routines, however, brings additiona complication.

1.6. Emulator for generic decompression

This method is a variation of the code emulation method. The difference is that the code emulator keeps on emulating until it heuristically determines that the code is unpacked. The emulator for generic decompression emulates code until a predefined stop event occurs. The event is defined statically.

In addition, some form of resource limitation is needed so that the unpacker will not be running „forever". Possible candidates are counting the number of emulated instructions or limiting the emulator time.

1.7. The new decompression method

The main idea

Idea of the new decompression method is based on a way of compression and decompression of a various compression programs such as the Winrar, bzip, etc. To decompress, these programs use standard decompression algorithms such as the Huffman algorithm, LZ77, LZW, etc. (see [6]). This idea is also based on the fact that designers of executable files packers can use these standard compression/decompression algorithms (perhaps sometimes slightly modified), too.

Based on this observation, we design a new decompression method for executable packed files. They can be decompressed in the following way. First, a compressed file is loaded into memory. Next, a statically (i.e., in advance) defined type of standard decompression algorithm is applied. (Sometimes, before applying decompression, it defines modification of decompression algorithm used in decompression routine and uses it to specify the decompression algorithm). Also, before applying decompression, the ranges of source compressed data and destination uncompressed data have to be specified.

New decompression method

We take into consideration that usually, the compressed executable files had also been encrypted in one way or another. Our new decompression method works therefore as follows:

1. Step. Identify and decode encrypted parts of the file following the rules set in advance.
2. Step. Identify places of compressed data and uncompressed target data following the rules set in advance.
3. Step. Decompress identified source compressed data by algorithms defined in advance and then save them to the defined destination.
4. Step. If necessary, go back to steps 1, 2, or 3.
5. Step. Reconstruct executable file body – it mostly includes only setting of new values of header structures (such as setting values of Entry point, Size of image, Section table, Import table, Import directory etc. see.[5]).

Encapsulation of the new decompression method into a decompression framework

As the next step, we designed an encapsulation of this method into a decompression framework (Figure 1). It was designed for general purpose with emphasis on flexibility. The idea is that a decompression program can be written with only use of decompression framework and some framework extensions.

Basic structure of the designed framework

The core part of the framework's design consists of five main parts: CFileStub, CSectionTable, CFileHeader, CImportDirectoryTable, Abstract_Algorithm. Parts CSectionTable, CFileHeader, CImportDirectoryTable represent main data structures of PE-files, which are needed to write the most of decompression programs to unpack PE-files.

CFileStub is an abstract class. It represents body of PE-file, as it is loaded into memory by operating system. By generalizing this class and properly implementing its abstract methods, anyone can create a new decompression program (e.g. CFileStub_FSG1_0 implements decompression PE-files packed by FSG version 1.0).

Abstract class called Abstract_Algorithm represents an interface to add standard algorithm to the decompression framework. The decompression algorithm that has just been added can be used to decompress via the defined interfaces. However, before using an added decompression algorithm, its specification should be amended to reflect concrete requirements of the compression program in question (e.g. the class *FSG20_control_stream* specifies modification of the LZ77 algorithm, represented by the class *LZ77_algorithm* to use in any type of file packed by FSG up to version 2.0.)
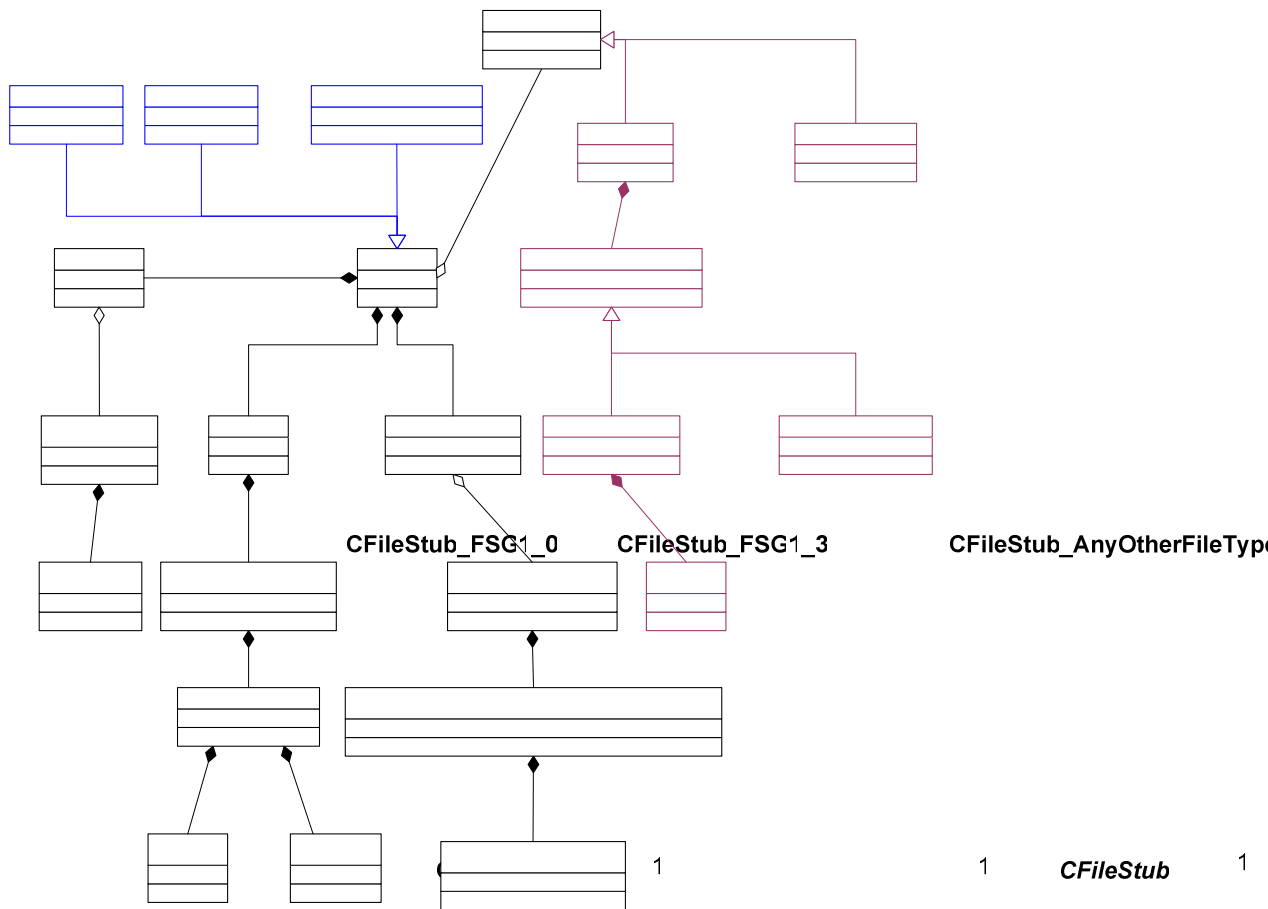
Figure 1: Class diagram of the design of the core part of the framework – it represents encapsulation of the designed method to decompress PE-file

1.8. Advantages and disadvantages of the new designed decompression method

Basic properties of the designed decompression method are similar to the single-purpose decompression programs method.

One of the advantages is speed of decompression. This is similar to the single-purpose method. However, our decompression methods are written in high level language C++. Contrary to that, the single-purpose method is usually written in an assembly language. Speed is ensured by extensive code optimization used in translators. Single-purpose decompression program method should incorporate also many checks (buffer underflow, overflow, etc.), because developer does not need to know how decompression routine really works.

Second advantage is high level of code reusing. This is at the same time the source of the next advantage.

Third advantage, as compared to the single-purpose decompression program method, is reduction of time necessary to implementation and testing of a new unpacker program. In general, it is not necessary to test the implemented reused decompression algorithms so extensively since they were tested many times before.

However, this method has not only advantages, it also has some disadvantages. The first disadvantage is that it is necessary to accomplish a detailed analysis of decompression routine and find out what kind of decompression algorithm (and its modification) is used.

Another disadvantage is that it is necessary to design and to implement flexible decompression algorithm, which enables to specify this algorithm in various ways (when we started this work, it was not immediately clear, if it could be done this way in a large scale). Yet another disadvantage is that the designed method constitutes no full alternative of the single-purpose decompression programs method, mostly in cases when there was used a nonstandard compression or a compression similar to encryption.

Let us present ant advantage, too. There is simple porting of the method used for creating of decompression programs. Reason for it is that these programs are written in high-level language C++, whereas the single-purpose decompression method is usually written in some assembly language.

1.9. Examples of implementation of some parts of decompression programs

Example of specification of decompression algorithm part

This example represents the specification of modification of the LZ77 algorithm - "control stream" used in decompression of PEDiminisher 0.1 packer by this algorithm. It does not describe implementation of override abstract methods. It describes only generalized class, which encapsulates the specification of the modification the LZ77 algorithm.

```
class PEDiminisher0_1_control_stream : public Abstract_control_stream_LZ77
{
public:
        PEDiminisher0_1_control_stream();
        PEDiminisher0_1_control_stream(BYTE *source, unsigned long int source_length, BYTE
*destination, unsigned long int destination_length);
        virtual ~PEDiminisher0_1_control_stream();
```

Example of specification of decompression algorithm (before its using) and the subsequent decompression

This example represents specification of modification the decompression algorithm and the next usage of specified decompression algorithm for decompression of two files compressed by PEDiminisher 0.1 and FSG 2.0. (Note: the parts specifying modifications of LZ77 algorithm are colored blue, parts defined for limits of source data, and target-uncompressed data are colored green).

Example of decompression program for PEDiminisher 0.1 packer:

```
PEDiminisher0_1_control_stream *control_stream = new
    PEDiminisher0_1_control_stream(&(FileBody[reg_ebx + 1]),   FileBodyLength - reg_ebx - 1,
    &(tmp_data[1]), reg_ecx - 1);

    algorithm = new LZ77_algorithm(&(FileBody[reg_ebx + 1]), FileBodyLength - reg_ebx - 1,
    &(tmp_data[1]), reg_ecx - 1, control_stream);

if (algorithm->unpack()) {
        delete control_stream;
        return 1;
```

}
Example of decompression program for FSG 2.0 packer:

```
FSG20_control_stream *control_stream = new
FSG20_control_stream(&(FileBody[source + 1]),source_length - 1 ,
&(FileBody[destination + 1]), destination_length - 1);

if (destination > FileBodyLength || source > FileBodyLength)
        return 1;
FileBody[destination] = FileBody[source];
    algorithm = new LZ77_algorithm(&(FileBody[source + 1]),source_length -1,&(FileBody[destination
    + 1]), destination_length - 1  , control_stream);

if (algorithm->unpack()) {
        delete control_stream;
        return 1;
}
```

1.10. Evaluation

The method that we propose in this paper aims at improving software process characteristics of design and implementation of decompression methods. How easy is to design and implement a decompression program is not easy to evaluate. We attempted to go beyond some qualitative evaluating comments based on experience of those who tested the method. As a simple quantitative measure, we tried to collect estimations of development times. We gathered time values from actual development efforts as performed by various programmers. It is clear that such values are afflicted by individual variations of programming performance. Nowadays, programmers' productivity can be considered more comparable mainly due to better formal education and more elaborated standard software methods and tools, thus making the quantitative values at least roughly objective.

2. Comparison of the new method to other decompression methods in terms of total development time

Using the method of experimental evaluation as described above, we wanted to find out how our method encapsulated by framework compares to development of a decompression program as code emulator on one side, and to development of single-purpose decompression programs on the other side. We were interested in finding out which of the three requires less time when several unpackers are developed, both similar to the first one in the row and also several different ones. As we can see, the most time-consuming method for development of the first (initial) unpacker is code emulator followed by our method encapsulated by decompression framework. As the new packers are developed, the total development time for all packers suggests that the single-purpose decompression program method is least effective and the code emulator is the most effective one.
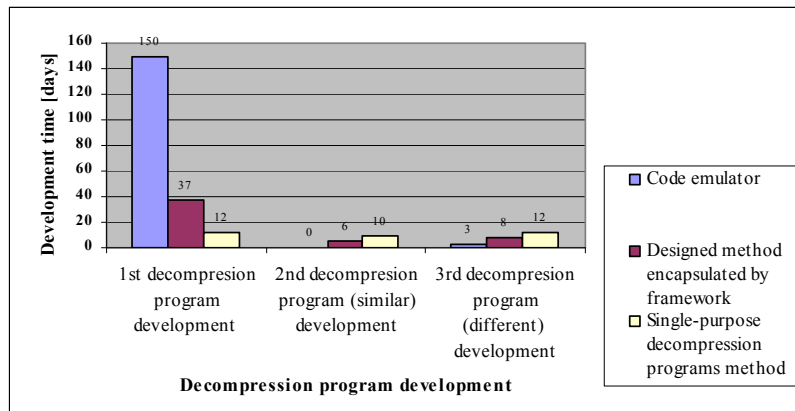
Figure 2: Graphical comparison of the new to other decompression methods in terms of total development time

3. Comparison of the new method to other decompression methods in term of initial development time

By initial development time we understand here time devoted to development that is needed before developing the first unpacker.
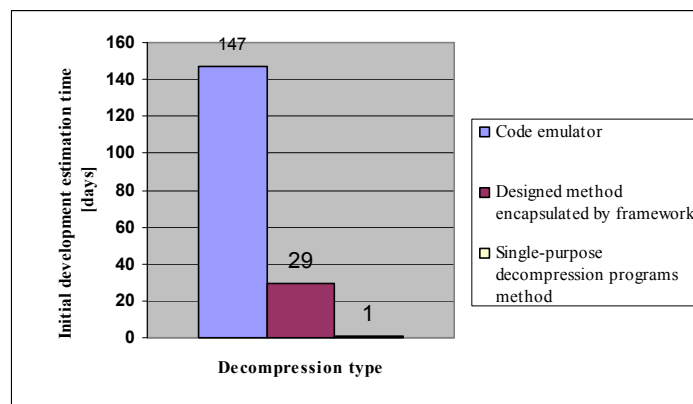

Figure 3: Graphical comparison of the new method to other decompression methods in term of initial development time

This graph helps understand why code emulator and the new method are quite time-consuming methods as far as developing first few unpackers is concerned. It is caused by the need to prepare the emulator resp. the framework before actual development of the unpackers can start..

4. Comparison of the new designed method to other decompression methods in terms of decompression routine analysis time.

As one can see from the graph, the most time-consuming method for analysis is our new method. The

reason is that it is necessary to find out how a decompression routine works and also the type of a decompression algorithm (or its modification). The depicted time of analysis of the code emulator method represents analysis if code emulator cannot emulate some packed files (e.g. it can be caused by occurrence of unsupported system call in decompression routine etc.).
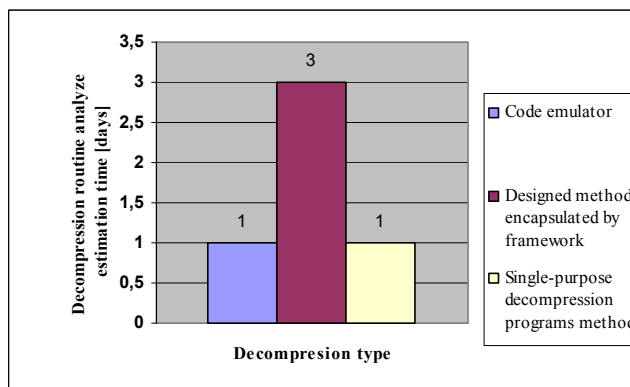

Figure 4: Graphical comparison of the new method to other decompression methods in term of decompression routine analysis time

Time interval of analysis of single-purpose decompression programs method represents the basic analysis of structure decompression routine.

5. Comparison of the new method to other decompression methods in terms of decompression routine implementation time.

Viewing the graph (Figure 5), following observations can be made:

- If the code emulator is implemented sufficiently comprehensively, only little implementation work is required to complete its emulation functionality for the first decompression program to be implemented. For subsequent implementations of similar decompression programs, it is reasonable to assume that no further implementation effort is needed at all. For implementations of different decompression programs, implementation effort should be comparable to that of the first one.

- Application of our new method encapsulated by framework makes the situation quite different. Our method requires implementing decompression routine whenever a new unpacker is implemented. This makes our approach similar to implementations of single-purpose decompression programs. The essential difference is that our method makes use or already implemented (and tested) standard decompression algorithm and reuses parts of code encapsulated in the framework. The extensive algorithm and code reuse are the reasons for shorter implementation time.
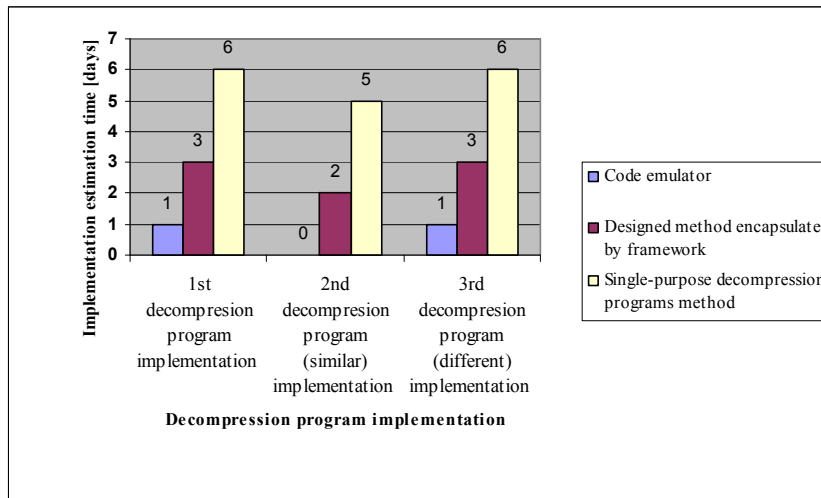
Figure 5: Graphical comparison of the new method to other decompression methods in terms of decompression routine implementation time

6. Comparison of the new method to other decompression methods in term of testing time

The graph (Figure 6) is similar to the previous graph. This fact can be justified by a general observation: "the more implementation, the more testing time is needed".
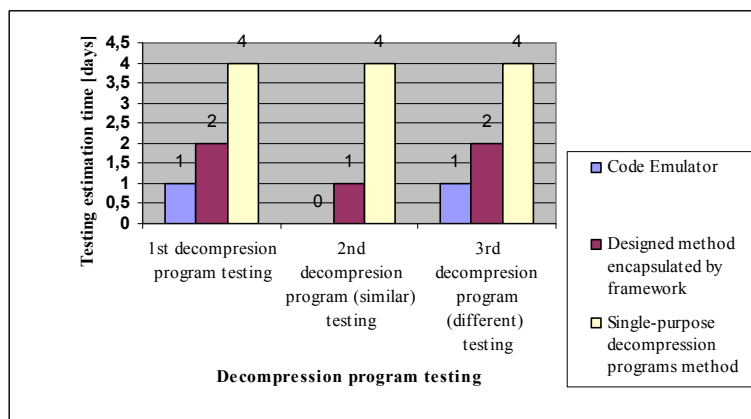

Figure 6: Graphical comparison of the new method to other decompression methods in terms of testing time

7. Comparison of the new method to other decompression methods in terms of file set decompression time

The measurements were performed on a set of various compressed files. Their decompression is supported by all the decompression methods. The measurement results support our hypothesis concerning the properties of our new decompression method encapsulated by framework.

In particular, as a positive result, we see the fact that there is only a small difference in decompression times between the designed method encapsulated by framework and single-purpose decompression programs method. This being (roughly) equal, what matters are the advantages of design with decompression framework over traditional single purpose decompression programs.
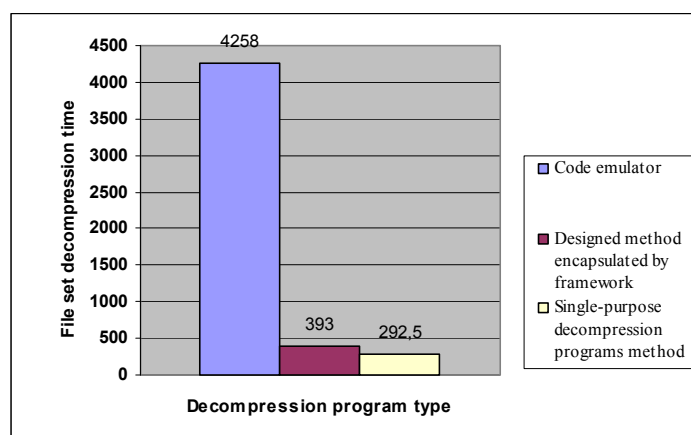

Figure 7: Graphical comparison of the new method to other decompression methods in terms of file set decompression time

8. Conclusion

We designed new method of implementing decompression programs. It is based on the idea of creating an encapsulating framework. Designing decompression programs in this way turns out to be a more efficient way of programming while allowing to keep essentially the same level of efficiency of the resulting programs. It was practically verified, tested and the achieved results were compared with other decompression methods. The results suggest the designed method is sufficiently robust, comprehensive, fast to implement and test.

Using of the designed method encapsulated in decompression framework depends on what we expect from the decompression program to be implemented. Since our method gives decompression programs of comparable efficiency to single-purpose decompression ones, the positive effects of encapsulating framework are likely to prevail. On the other hand, if the expectations are flexibility and ability to support as many compression programs as possible, code emulator might remain a better choice.

Acknowledgements

REFERENCES

1. CORLISS, M.L., LEWIS, E.C., ROTH, A.: **A DISE Implementation of Dynamic Code Decompression**. ACM SIGPLAN Notices, Volume 38 , Issue 7 (July 2003), pp. 232-243.

2. DEBRAY, S., EVANS, W.: **Profile-Guided Code Compression**. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, Berlin 2002, ACM SIGPLAN Notices Volume 37 , Issue 5 (May 2002), pp. 95-105.

3. HOOGERBRUGGE, J., AUGUSTEJN, L., TRUM, J., VAN DE WIEL, A.R.: **A Code Compression System Based on Pipelined Interpreters**. Software – Practice and Experience, 29, 11, 1005-1023- 1999.

4. LEFURGY, C., PICCININNI, E., MUDGE, T.: **Reducing Code Size with Run-Time Decompression.** In: Sixth International Symposium on High-Performance Computer Architecture, 2000, IEEE Computer Society, pp. 218-230.

5. SHIPP, A.: **Unpacking strategies**. Virus Bulletin conference, September 2004:

6. Microsoft Corporation, Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0 - February 1999, [13.5.2005].
   http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/pecoff.doc

7. Data compression algorithm, [13.5.2005].
   http://encyclopedia.thefreedictionary.com/Data%20compression%20algorithm.