Study on a known-plaintext attack on ZIP encryption

Dragos Barosan
dragos.barosan@os3.nl

February 8, 2015

Abstract

The ZIP file format is one of the most popular compression format and it provides a stream cipher encryption for protecting data. A successful known plaintext attack has been developed since 1994, but there is no open source implementation for it. The research has focused on the feasibility of a successful, since the necessary plaintext is considered hard to obtain, and analyzed the algorithm. It has been found that, while difficult, plaintext can be found through varied resources. From an implementation point of view the algorithm contains sections that can be run in parallel, improving the execution speed. As future work, a full implementation of the algorithm is planned and it will be released as open source.

Contents

Chapter 1

Introduction

The ZIP archive file format was originally created in 1989 by Phil Katz to support lossless data compression and replace the ARC archiving system. The first version has been has been released in the PKZIP package from the PKWARE software company[1]. Since then the format has been released in public domain and new versions are available on PKWAREs website under the name APPNOTE - .ZIP File Format Specification[2].

An important milestone in the history of the format is the introduction of the ZIP encryption starting with version 2.0 in 1993[3]. The encryption was based on an algorithm developed by the mathematician Roger Schlafly. This was the only available method of encryption specified in the standard until 2002 when support was introduced for other ciphers like 3DES, RC4 and AES[4].

The first attack against the ZIP encryption was developed in 1994 by Eli Biham and Paul C. Kocher[5]. They developed a known plaintext attack algorithm that is able to break the encryption and recover the original password in a reasonable time . The duration of a successful attack of this type depends primarily on the amount of known text that is available to the attacker. Although AES encryption has been introduced starting with the fifth specification of the zip file format, the old encryption is still used by tools today because it is considered to be difficult to obtain the required plaintext from the original file.

The paper is focused on analyzing the feasibility of obtaining the necessary amount of plaintext for a successful known plaintext attack and on exploring different implementation options because, although there are tools that make use of the algorithm, there is no open source implementation available. An open source implementation would help research the present weaknesses and raise awareness that people should switch to stronger encryption.

Chapter 2

Research questions

The investigation focuses on researching the feasibility of the known-plaintext attack on the ZIP encryption, what implementation possibilities are for it and providing a proof of concept. This can be formulated under the following research questions:

- How feasible is to obtain the known plaintext for a successful attack?
- What implementations are possible for the attack?

Chapter 3

Related work

Even though the encryption algorithm and the first attack on it were developed more than twenty years ago. Based on the literature review, not much research has been done on them. Further on there are mentioned works on which part of this paper is based on.

The algorithm on which this investigation is based was originally developed by Eli Biham and Paul C. Kocher[5]. They demonstrated that the encryption can be broken with as little as eleven bytes of known plaintext and with a complexity of at most 2^{38}. In the paper the complexity refers to how many items have to be processed at one stage in the algorithm.

Peter Conrad developed PkCrack, the only known tool with the source code available that implements the Biham and Kocher algorithm[6]. This implementation is in the C programming language and his tool has been used with good results. He explicitly states that any software using parts of his code without his consent is forbidden and so is the distribution of it in any commercial form. The last update to the code was in January 2003 according to the source files properties.

An improvement of the chosen plaintext attack was introduced by Michael Stay[7]. His paper illustrates ways of reducing the amount of plaintext required. First, he makes a refinement of the Biham and Kocher attack that results in a decrease to only six bytes of necessary plaintext if there are at least four encrypted files in the same archive at the trade-off of an increase in complexity to 11* 2^{40}. Secondly, he introduces a new attack approach that requires only two bytes of known plaintext with a complexity of 2^{63}. Using this attack, by exploiting the pseudo random number generator used by WinZip versions prior to 8.1, an attack could succeed without the need of any plaintext and with a complexity of 2^{39}, which is comparable to the original attack . A commercial tool that implements this is the Advanced Archive Password Recovery from Elcomsoft.

Another attack was developed by Mike Stevens and Elisa Flanders that exploits the pseudo random number generator provided by the library IBDL32.DLL[8]. Their attack does also not require any known plaintext. No implementations,

open source or not, could be found of this attack.

Chapter 4

Approach

For the development of the proof of concept the Python programming language was used, with the CPython version 2.7 reference implementation available from the Debian distribution packages. It was chosen because the absolute running time of the PoC was not of interest, but the relative speed between multiple implementations.

The Linux /bin/usr/time tool was used to measure the running time of the different applications tested. In some cases the Python datetime module was used to measure the running time of certain sections of code. All tests were run on an Intel Core I7-3610QM with four cores running on 2.3 Ghz frequency.

For compression and creating archives the Linux zip 3.0 utility was used unless specified otherwise.

The PkCrack software was used for multiple tests regarding the ZIP encryption. All tests that implied breaking the ZIP encryption were done with PkCrack.

The investigation first focused on the research how ZIP encryption and compression works and what solutions are available for obtaining plaintext. The second part of the research focused on implementing two proof of concepts: one that will run in parallel and a serial implementation. Only part of the whole algorithm was studied.

Chapter 5

Feasibility of obtaining plaintex

5.1 ZIP Defaults

The zip encryption is old so it is interesting to see if this method of protecting zip archives is still used. For the investigation, three of the most popular ZIP compression software applications[9] were selected: WinZip, WinRar and 7ZIP. To add to this ones the Linux zip utility and PKcrack from PKWare, which owns the ZIP specification, were also taken into consideration. To check the vulnerability of each tool, test files were archived and encrypted using the methods available. Then, using PKcrack, it was tested which ones can be decrypted to the original value. The tested versions and the results are presented in table 1.1

Application	**Version**	**Support for ZIP encryption**	**Support for AES encryption**	**Default**
WinZIP	19.0	Yes	Yes	AES
WinRAR	5.21	Yes	No	ZIP encyption
7ZIP	9.2	Yes	Yes	ZIP encryption
PKZIP	14.20	Yes	Yes	ZIP encryption
zip	3.0	Yes	No	ZIP encryption

Table 5.1: Zip utilities

As the results from the table illustrate, all considered applications support ZIP encryption, while only WinZip, 7zip and PKZIP support AES encryption. Furthermore, it was noticed that WINRAR and zip do not warn the user about

the insecure encryption that is used. The possibility emerges that the average user will choose weak encryption when using applications that do not default to AES as the encryption method. As a result, archives vulnerable to the known plaintext attack are still created and used.

5.2 ZIP Encryption

Here the ZIP encryption algorithm, which functions as a byte-oriented stream cipher, will be presented as specified in the zip file format specification[2]. It is important to mention that first 12 random bytes are prepended to the plaintext before the encryption process. No header fields are encrypted, only the data.

The cipher mechanism has an internal state of 96 bits that consists of three 32 bit words referred as key0, key1 and key2. These are initialized with 0x12345678, 0x23456789, and 0x34567890. From this internal state the actual encryption key is derived. The encryption key is referred to as key3 and represents an 8 bit value.

The internal state, and subsequently key3, is updated as follows:

$$key0_i = crc32(key0_{i-1}, character_byte) \tag{5.1}$$

$$key1_i = (key1_{i-1} + LSB(key0_i)) * 134775813 + 1(mod 2^3 2) \tag{5.2}$$

$$key2_i = crc32(key2_{i-1}, MSB(key1_i)) \tag{5.3}$$

$$temp_i = key2_i OR 3(2LSB) \tag{5.4}$$

$$key3_i = LSB((temp_i * (temp_i XOR 1)) >> 8) \tag{5.5}$$

Equations 5.1 and 5.3 use a linear feedback shift register known as CRC32, the Cyclic Redundancy Check function: given a 32 bit value and an 8 bit value as input it returns another 32 bit value. The polynomial used is represented by the 0xedb88320 value. Equation 5.2 uses a truncated linear congruential generator. The internal state is first updated with the password characters and then using the plaintext characters. The ciphertext characters result from the XOR operation between the plaintext characters and key3.

5.3 Difficulty of obtaining plaintext

The primary obstacle to a successful attack is the difficulty of obtaining the plaintext. As it is shown in[5] usually at least thirteen consecutive bytes are required for the original attack to succeed.

This difficulty comes from the fact that the plaintext does not refer to the contents of the original file but to the contents of the compressed form of the original file, using the same compression technique as the one used by the encrypted file. To understand this we must look at how compression works. The most used compression method is Deflate which consists of two steps[10]:

- The matching and replacement of duplicate strings with pointers.

- Huffman coding

The Huffman coding replaces symbols with new, weighted symbols based on frequency of use using the principle that the more likely a symbol is, the shorter its new bit-sequence representation will be. This fact induces the conclusion that compressed text will have a completely different bit representation than the uncompressed one. In the image below a binary difference is shown between archives of the same file, one deflated with one percent and one uncompressed. The data section that starts from offset Ox3e is completely different from one case to the other.

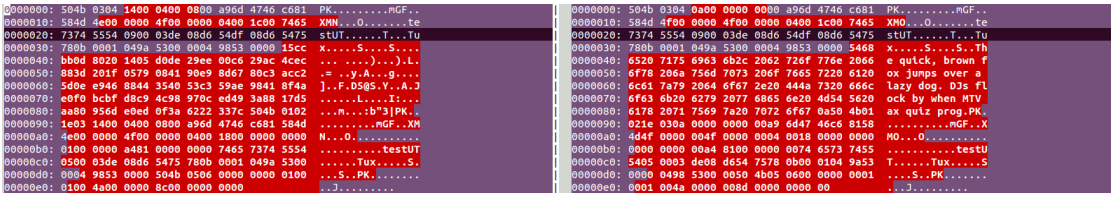

Figure 5.1: Binary difference between the file compressed and uncompressed

From this aspect the difficulty of obtaining the necessary plaintext emerges. Even though the attack requires only thirteen bytes, to have access to any amount of plaintext the entire original file is needed and it needs to be compressed in the exactly same way.

It was observed among the test files that some files, even though were compressed using maximum level compression, they were still in uncompressed form. The explanation of this is that, since the compression algorithm is using entropy encoding, redundancy is needed for an actual compression.

The research focused on determining the necessary amount of bytes necessary for compression to actually take place. Both compression algorithms offered by the zip utility, Deflate and bzip2, were tested with multiple inputs[11]. The results are presented in table 5.2.

Input	**Deflate 1-9**	**bzip2**
One Symbol	8	43
Lorem Ipsum	56	129
Kafka	64	140
Pangram	78	162
Random Symbols	127	237

Table 5.2: Minimum number of bytes for compression to work

It can be seen that for highly redundant input, with low entropy, (eg. Single symbol input) the necessary amount of bytes is very low for the compression takes place. When high entropy is present in the input the necessary size of the

file is increased to a factor of ten. This aspect could be exploited: if an attacker tries to break an encrypted archive and it notices that one of the files is below a certain threshold it can assume that the file is in an uncompressed form and thirteen bytes from the original file could be used.

5.4 Solution

Further on some solutions to obtaining the necessary plaintext are presented.

- One known file from an archive

The Zip file format specification states that:

> Each encrypted file has an extra twelve bytes stored at the start of the data area defining the encryption header for that file. The encryption header is originally set to random values, and then itself encrypted, using three, 32-bit keys. The key values are initialized using the supplied encryption password. (Section 6.1 from [2])

From this it can be inferred that each file in an archive is encrypted independently with the same password, so they all have the same initial encryption internal state. If this is true, then it implies that we only need to know the contents of one file, which has at least thirteen bytes, in the archive and the encryption can be broken. To test this, an archive with multiple encrypted files was created and, by using as known plaintext the contents of only one file, PKcrack managed to decrypt the entire contents. This confirms the above hypothesis. Examples of known files are images and presentations from the internet, executables, and research papers.

- File headers

In case that one of the files in in an unencrypted form then the necessary plaintext could be obtained from the file headers. One of the studied file formats was the windows executables. From the comparison of multiple files binaries it could be seen that, for the first 128 bytes, they all had the same value except the byte at offset 3c. According to the Windows documentation this bytes represent a DOS stub program that allows the executables to display an information message to the user instead of an error[12].

Another studied format was the Microsoft office documents format[13]. For documents created with Office 2007 or earlier versions the file format used was based on Compound file Binary Format that uses a 496 byte header .Comparing multiple office files it was observed that they all have the same value for the first 44 bytes. This would be enough for a successful attack and more bytes could be deduced by investigating the header further. Other investigated file formats which have at least thirteen common bytes were wmv files, torrent files, jpg files and png files. From this we can conclude that it is probable to find the

necessary plaintext from file headers if the encrypted files are not compressed in the archive.

- Encryption header

A small amount of plaintext can be obtained from the encryption header used by the ZIP encryption. According to the specification:

> After the header is decrypted, the last one or two bytes in Buffer should be the high-order word/byte of the CRC for the file being decrypted, stored in Intel low-byte/high-byte order. Versions of PKZIP prior to 2.0 used a 2 byte CRC check; a 1 byte CRC check is used on versions after 2.0 (Section 6.1 from [2])

This suggests that last byte of the header should be the most significant byte of the archived file CRC stored in plaintext. This value is used for a quick filter of the wrong passwords so calculation of the full CRC is not necessary each time a password is introduced. To test this the PKcrack tool was used with the plaintext specified as starting at offset -1 so it will start with the last byte of the prepended header. The result was a successful decryption with only twelve bytes of plaintext.

Chapter 6

Attack Implementation

6.1 Overview

In this chapter the original known plaintext attack on zip encryption will be presented. The main goal is to recover the complete internal state corresponding to a known plaintext byte. From there the file can be decrypted backward and forward[5]. This implies that the necessary plaintext does not necessarily have to be at the beginning of the text, but it has to be consecutive. The attack exploits the limited diffusion implemented by the encryption process and recovers information by reversing the functions used. An overview of the attack is presented next[5]:

- The encryption key can be recovered for each known plaintext byte by using XOR on the ciphertext with the right offset
- Using the list of key3s, multiple possible key2 lists are generated
- For each key2 list, multiple key1 lists are generated
- For each key1 list, one key0 list is generated
- Using the plaintext, the true key0 list can be obtained
- Using the true key0 list we can use the encryption update mechanism to determine the full internal state

The algorithm can be separated in two stages. The first stage is the initialization phase in which the data is located, the key stream is generated, key2s are generated. The second one comprises of the remaining steps in the algorithm. The research focused on the first stage and provides a proof of concept for it.

6.2 Locate Data

In order to generate the keystream, which will be used later on in the attack, the file data for which plaintext is available needs to be extracted in both encrypted and unencrypted form. It is assumed that the input for the Proof of Concept is the name of the archive which contains the encrypted files, the name of the archive that contains the file for which the plaintext is known and the name of that file. The necessary information can be obtained from the zip file format specification[2]. As figure 6.1 illustrates, each file in a zip archive has a corresponding Local Header. The zip local file header is shown in figure 6.2.

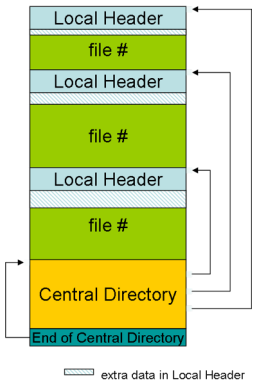

Figure 6.1: ZIP archive format[14]

ZIP local file header

Offset	Bytes	Description[5]
0	4	Local file header signature = 0x04034b50 (read as a little-endian number)
4	2	Version needed to extract (minimum)
6	2	General purpose bit flag
8	2	Compression method
10	2	File last modification time
12	2	File last modification date
14	4	CRC-32
18	4	Compressed size
22	4	Uncompressed size
26	2	File name length (*n*)
28	2	Extra field length (*m*)
30	*n*	File name
30+*n*	*m*	Extra field

Figure 6.2: ZIP file header[15]

The first important observation is that each file header begins with a specific signature. The signature can be used while parsing the archive binary to determine the start of the structure. The next step is to read the File name length field and check if it matches with the length of name that was provided as input. If it does not match then it is safe to go on to the next header until a match is found. Afterwards the extra field length value is saved and the actual file name is tested against the value in the File Name field. A match implies that the local header is associated with the file we are searching for. With this information the data can be located with the following equation:

$$Filedatastart = Localfilesignatureoffset + 30 + filenamelength + extrafieldlength \tag{6.1}$$

To extract the encrypted file data the same process is used, but the encryption header has to be taken into account, so formula 6.1 has to be slightly modified:

$$Filedatastart = Localfilesignatureoffset + 30 + filenamelength + extrafieldlength + 12 \tag{6.2}$$

The keystream is then generated by iterating, in byte increments, over the two extracted data sets and applying the XOR operation.

6.3 First stage of the attack

This section describes the first stage of the attack that includes the generation of key2s and the reduction phase of their number. In the Biham and Kocher paper it is stated:

> The value of key3 depends only on the 14 bits of key2 that participate in temp. Any value of key3 is suggested by exactly 64 possible values of temp (and thus 64 possible values of the 14 bits of key2). The two least significant bits of key2 and the 16 most significant bits do not affect key3 (neither temp). (Section 3.1 from [5])

To understand this we need to look at the equations used for encryption starting with the last two, specifically equation and . From we can see that temp represents the 16 least significant bits of key2 to which the OR 3 operation is applied. From this we know that the last 2 bits of temp are 11 so only 14 bits of key2 influence the computation of key3. If we change the terms in equation 5.5 we have the following:

$$key3 << 8 = temp * (tempXOR1) \tag{6.3}$$

Given a key3 there are 8 bits that we dont know for the left hand side of equation 6.3 but because we know that the last 2 bits of temp are 11 we can determine that the last 2 bits of the right hand side of the equation will be 10. Taking this into consideration there are only 6 unknown bits for the left side of which results in 64 possibilities. As a consequence there are only 64 possibilities for temp and implicitly 64 possibilities for bits 15-2 of key2, given key3. Adding up the unknown 16 most significant bits of key2 we are left with 2^{22} possibilities for the 30 most significant bits of key2. We do not care about the least significant 2 bits because they do not influence any step in the algorithm.

For generating the key2 lists we use the inverse of the CRC32 function:

$$key2_i = crc32_1^{-1}(key2_{i+1}, MSB(key1_{i+1})) \tag{6.4}$$

The paper describes the process:

> Given any particular value of $key2_{i+l}$, for each term of this equation we can calculate the value of the 22 most significant bits of the right hand side of the equation, and we know 64 possibilities for the value of 14 bits of the left hand side. (Section 3.1 from [5])

The known bits from the left hand side are 2 - 15 and on the right hand side there are 10-31. We can see that there are 6 bits in common between the two sides.

The following is stated in the paper:

> Only about 2 -6 of the possible values of the 14 bits of $key2_i$ have the same value of the common bits as in the right hand side, and

> thus, we remain with only one possible value of the 14 bits of $key2_i$ in average, for each possible value of the 30 bits of $key2_{i+1}$. When this equation holds, we can complete additional bits of the right and the left sides, up to the total of the 30 bits known in at least one of the sides. Thus, we can deduce the 30 most significant bits of $key2_i$. We get in average one value for these 30 most significant bits of $key2_i$, for each value of the 30 most significant bits of $key2_{i+1}$. Therefore, we are now just in the same situation with $key2_i$ as we were before with $key2_{i+1}$, and we can now find values of 30 bits of $key2 + i - 1$, $key2_{i-2}$, ..., $key2_1$ (Section 3.2 from [5])

The result of this is that we will have 2^{22} possible lists of key2. If we have more plaintext than the required thirteen bytes for the attack there are two possibilities: either discard the extra plaintext and start the algorithm from $key3_{13}$ or use it to reduce the number of possible lists. This is possible because if for every $key2_n$ we compute $key2_{n-1}$, the resulting values will contain duplicates which can be discarded. This process of key reduction can be safely repeated until $key2_{13}$ and will result in a significant reduction of the number of key2s.

6.4 Implementation

The Proof of concept implements the details of the algorithm described so far. The focus was to find what options are available for making the implementation efficient without investigating the source code of any other implementations. The first step that can be taken is to precompute certain functions and store them as an association of values in hash maps so time is not wasted during execution. Two of such structures are crctab and crcinvtab. They are used for computing more efficiently the CRC32 function and its inverse using the following equations:

$$CRC32(X, b) = (24 most significant bytes of X) XOR CRCtab[LSB(X) XOR b] \tag{6.5}$$

$$CRC32^{-1}(B, b) = (B << 8) XOR CRCinv[MSB(B)] XOR b \tag{6.6}$$

Another hash map structure was generated for associating temp values with key3s. So for each key3 there will be associated a list of all 64 possible temp values. This can be done in two ways: by solving equation for all possible key3 or by iterating over all possible temp values, do the computation described in equation and associate the result with the temp value that was used to produce the result. The latter option was chosen because it results in more readable code and there is no significant computation difference between the two options. The key2 reduction step is the section that requires the most computation time since it is necessary to iterate over the 2^{22} key2 possible values in the first iteration and then again for the remaining possible key2s after the duplicates are discarded.

The discarding operation can be done by sorting the resulting list and checking for neighbor duplicate values, but in Python there is a set function which does this automatically provided a list of values.

Two approaches were considered for the implementation of the key 2 reduction phase: A serial one and a parallel one so advantage can be taken if the program would be run on a system with multiple cores. A parallel approach is possible because each $key2_{i-1}$ is computed only based on the dependency of one $key2_i$.

For the parallel implementation the best option would have been the use of threads since they could operate on shared data. This was not possible because in the implementation of Python there is a Global Interpreter Lock which forbids threads running in parallel[16]. As an alternative multiple processes were considered for running the computation in parallel. This raised some difficulties because processes do not share the same memory space so two solutions were tested: one using a shared array between processes and one where every time an iteration would return a reduced list of possible key2s the old processes would be killed and new processed would be created so each will have its own copy of the new list of key2s. The latter option works because each process only reads data from the list and does not modify it.

6.5 Measurements

The serial implementation was tested using different plaintext sizes as input and its results were compared with the data available from [5] and with the results from using PKcrack. These results are shown in table 6.1

Plaintext bytes	**PkCrack**	**PoC**	**Paper**
122	59584	73175	70000
506	15471	17009	16800
1002	8080	8101	7780
3990	2832	3416	4000
10000	2325	2593	1857

Table 6.1: Number of remaining key2's after the reduction stage

The number of possible key2s returned by the PoC serial implementation are close to the values from the other 2 sources. They do not match exactly with the ones from the paper because it is a probabilistic process and in their computation different plaintext was used. The reason for which it does not match the numbers from PkCrack could not be explained since the source code was not explored. In the graph provided in [5] it can be seen that the number of key2 can increase not just decrease after a certain amount of plaintext. An option that could be explored is to always keep track of the smallest number of key2s and the offset and return the corresponding list.

The two parallel versions were run and the solution with a shared array was quickly discarded because the parallel computation was approximately 80 times slower than the other option. The reasons for this result were not investigated further.

The running time of the remaining parallel implementation and the serial one were compared and the results are shown in table 6.2.

Plaintext bytes	**Execution time Parallel (minutes)**	**Execution time Serial (minutes)**	**System/User time Parallel**	**System/User time Serial**
40	0:34.44	1:03.6	0.0647	0.0026
122	1:08.5	1:38	0.1648	0.0017
309	1:49.3	1:56	0.3411	0.0014
506	2:29	2:07.2	0.5066	0.0012
1002	3:28	2:22	0.7455	0.0011
3990	10:07	3:02.1	1.455	0.0009

Table 6.2: Comparisson between the serial and parallel implementations of the PoC

It was observed that the parallel section ran as expected approximately four times faster than its serial counterpart, since a four core system was used, but the total running time increases significantly if the size of the plaintext is increased. This can be explained by the fact that the time cost of killing and creating new processes at each iteration stays constant while the benefit of running parallel computations gets smaller at each iteration since the list of values it is operated on decreases in size at each iteration. This is supported by looking at the measured proportion of the System and User time for each run of the PoC. It can be seen that while the plaintext size is increased the amount of time spent in the kernel, where processes are managed, is also increasing for the parallel implementation, in the same time the proportion stays constant for the serial version.

Chapter 7

Conclusions and Future Work

In conclusion, the ZIP encryption can be considered a serious security vulnerability since there are multiple feasible ways of obtaining the necessary amount of plain text for an attack to be successful. In cases where the pseudo random number generator, used to produce the random encryption header, is weak, attacks are possible that do not require any plaintext. For encrypting archives it is recommended to use tools that support secure encryption. If this is not possible then care should be taken with the contents of the archives to not contain any well known file that is available on the Internet or any uncompressed files. The investigated section of the algorithm can be implemented efficiently in parallel taking advantage of multiple cores, but only using threads so no extra overhead is necessary for the working threads. Python makes this difficult since it does not support parallel threads. From the available results it emerges that a parallel implementation using multiple processes is no feasible.

This investigation mostly focused on the original plaintext attack and only on one of its stages. It is of interest to study in more detail the next stages of the attack. The other known plaintext attacks will be studied in more detail and a conclusion will be drawn on the exact benefits of each method. A complete implementation of the algorithm is a planned in a language that will allow the benefits of threads and also provide fast execution times. This software will be released under an open source license so everyone will be able to improve upon it.

References

[1] NYTimes, *Phillip Katz, Computer Software Pioneer, 37*, 01 May 2000, http://www.nytimes.com/2000/05/01/us/phillip-katz-computer-software-pioneer-37.html

[2] PKWARE, *APPNOTE.TXT - .ZIP File Format Specification*, 01 October 2014, https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

[3] Paul Lindner, *Registration of a new MIME Content-Type/Subtype*, 20 July 1993, http://www.iana.org/assignments/media-types/application/zip

[4] PKWARE, *APPNOTE.TXT - .ZIP File Format Specification*, 02 February 2003, http://web.archive.org/web/20030702014023/http://pkware.com/products/enterprise/white_papers/appnote.html

[5] Eli Biham and Paul C. Kocher, *A known plaintext attack on the PKZIP Stream Cipher*, 1994, http://link.springer.com/chapter/10.1007%2F3-540-60590-8_12#page-1

[6] Peter Conrad, *PkCrack*, 2001, https://www.unix-ag.uni-kl.de/~conrad/krypto/pkcrack.html

[7] Michael Stay, *ZIP Attacks with Reduced Known Plaintext*, 2002, https://www.cs.auckland.ac.nz/~mike/zipattacks.pdf

[8] Mike Stevens and Elisa Flanders, *Yet another plaintext attack to ZIP encryption scheme*, 8 February 2003, http://www.securityfocus.com/archive/1/311059

[9] tucows, *Compression utilities downloads*, 2015, http://www.tucows.com/Windows/is-it/file-management/compression-utilities/?f=pop

[10] P. Deutsch, *DEFLATE Compressed Data Format Specification version 1.3*, May 1996, https://www.ietf.org/rfc/rfc1951.txt

[11] Nicolai et all, *Text Generator*, 2015, http://www.blindtextgenerator.com/

[12] Microsoft, *MZ Stub*, 2015, https://msdn.microsoft.com/en-us/library/7z0585h5.aspx

[13] Microsoft, *Compound Binary File Format*, 2015, https://msdn.microsoft.com/en-us/library/dd942193.aspx

[14] Emanuele Ruffaldi, *Extracting-files-from-a-remote-ZIP-archive*, 30 October 2004, http://www.codeproject.com/Articles/8688/Extracting-files-from-a-remote-ZIP-archive

[15] Wikipedia, *Zip file header*, 2015, en.Wikipedia.org/wiki/Zip_(file_format)

[16] Sebastian Raschka, *An introduction to parallel programming*, 20 June 2014, http://sebastianraschka.com/Articles/2014_multiprocessing_intro.html

Chapter 8

Appendices

Both serial and parallel proof of concepts are available at the following link:
`https://github.com/dragosb91/RP1/tree/master`