PEA

Document revision: 2025 10 11

Author: Giorgio Tani

Translation: Giorgio Tani

This document refers to:
   PEA file format specification version 1 revision 6 (1.6);
   PEA file format specification version 2.0;
   PEA 1.27 binary release;

Present documentation is released under GNU GFDL License.
   PEA executable implementation is released under GNU LGPL License; please note that all units provided by the Author are released under LGPL, while Wolfgang Ehrhardt's crypto library units used in PEA are released under zlib/libpng License.
   PEA file format and PCOMPRESS specifications are hereby released under PUBLIC DOMAIN: the Author neither has, nor is aware of, any patents or pending patents relevant to this technology and do not intend to apply for any patents covering it. As far as the Author knows, PEA file format in all of it's parts is free and unencumbered for all uses.

Pea is on PeaZip project official site:
https://peazip.github.io **, and** **https://peazip.sourceforge.io**

For more information about the licenses:
   GNU GFDL License, see http://www.gnu.org/licenses/fdl.txt
   GNU LGPL License, see http://www.gnu.org/licenses/lgpl.txt

Content:

Section 1: PEA file format

Description

PEA, Pack Encrypt Authenticate, is a general purpose archiving format, featuring compression and multiple volume output, aiming to offer a flexible security model through Authenticated Encryption, which provides both privacy and authentication of the data, and redundant integrity checks ranging from checksums to cryptographically strong hashes, defining three different levels of communication to control: streams, objects, volumes.

Stream control level is actually the only level featuring Authenticated Encryption option while integrity checking can be performed at each of the three levels; each one of the three levels of control can be omitted to fit particular user's needs.

PEA archive data (except for archive header) is organized in objects; objects are organized in streams (1.x file format allow archives containing a single stream while 2.0 allow unlimited streams), being a stream a group of objects sharing the same security constrains, satisfied by the stream level checking algorithm (checksum, hash or AE).

Stream beginning is marked by a "stream header" type of object and stream termination is marked by an "end of stream" type of object or by "end of archive" if the stream is the last one in the archive; defining PEA 1.1 file format only archives containing a single stream, it's always terminated by an "end of archive" object.

First field of an object is the object's input name size (word sized), when the field is nonzero, the archive object has an associated input name, that means the archive object maps an input object (file or dir) from the system, otherwise if the name size is zero it means the archive object is a trigger, a functional field for PEA program (as aforementioned "stream header" , "end of stream" and "end of archive" objects), and that no external data is mapped fro the system to the archive object.

Objects of "trigger" class are not checked by object level check, while archive objects with associated input object ("input object" class) are checked; both classes of objects are checked by stream level check.

If the object is a trigger, the second field is the trigger type, 4 byte sized; defined types are:

- POD (followed by $00 byte): "stream header" type of object, marks the beginning of the stream, it is followed by 4 byte-sized fields defining stream properties (compression scheme, stream control scheme, single objects control scheme, stream-specific error correcting strategy) and, if Authenticated Encryption is used for stream control, by a field containing an header similar to the one defined for Wolfgang Ehrhardt's FCA cryptographic application:
  - o If single encryption is used, 16 bit header is generated:
    - o 1 byte FCA sig ($FC): in PEA file format it is zeroed since file format disambiguation is performed on archive's header rather than on this field;
    - o 1 byte flags field: in PEA file format it is zeroed since the information about encryption scheme is given in "stream header" area;
    - o 12 byte (96 bit) pseudorandom salt;
    - o 2 byte key verification field;
  - o If triple cascaded encryption is used, variable sized header is generated (each 16 bit sub-header area is composed as the previously described encryption header, unless differently specified)
    - o 16 bit sub header for first encryption algorithm, with key verification zeroed
    - o 16 bit sub header for second encryption algorithm, with key verification zeroed
    - o 16 bit sub header for third encryption algorithm, with key verification which is the result of xor of the three key verification tags
    - o 1..128 byte variable sized sub header containing random data to be discarded, meant to mask exact output size; first byte contains the number of following bytes to be discarded. This sub header the first area to be encrypted.
- EOS (followed by $00 byte): "end of stream" type, marks the end of a stream, it's followed by the stream control tag (variable sized), exists only in 2.0 file format specification;
- EOA (followed by $00 byte): "end of archive"; and marks the end of the last stream of the archive, it's followed by the stream control tag and from the last volume control tag, then the archive ends.
- MSG: triggers a message, the 4th byte define the size in byte of the message, allowing short messages up to 255 bytes; if the 4th byte is $00, the message is a long message and in this case next 4 byte field (dword) define the size in byte of the message. It may be useful to insert comments or metadata (i.e. graphic, list of stream content to improve user experience allowing faster stream content preview etc).
  MSG type exists only in 2.0 file format specification.

If the archive object is not a trigger, the second field (variable sized, with size defined by previous field) is input object's name, the qualified name of the file or dir in the system where the archive is created, followed by 4 byte (dword) sized field for object's last modification time and another 4 byte (dword) sized field for object attributes.

If the input object is a dir, no more fields are needed by PEA, next field will be the object's control tag; otherwise if the input object is a file, the next object's field is 8 byte (qword) sized file size, allowing max input size of 2^64 byte.

If the size is zero, the file is empty and next filed will be the object's control tag; otherwise next field will be the file content (structure of that data may vary due to the compression model used), and finally the object's control tag.

Object name size: 2 byte
word sized variable: if $0000 then the object is a trigger, no input object mapped to the archive object
else, the value is the size of the name of the input object in the host system

Tigger type: 4 byte declaring trigger type
POD: marks beginning of the stream (a "pea pod");
EOS: marks end of the stream (only in PEA 2.0 file format);
EOA: marks end of the archive;
MSG: message (only in PEA 2.0 file format).

Object name, variable sized
Object date and time, last modified (dword)
Object attributes (dword)
if object is a dir (from attributes, otherwise it could be conventionally marked in the object name ending the name of dirs with system's directory separator), next field is object control tag
else next field is object size

Class "trigger object"
object name size = 0

Class "input object"
object name size <> 0

Object size (qword)
if object is an empty file (size is 0), next field is object control tag
else next field is file data (may be compressed)
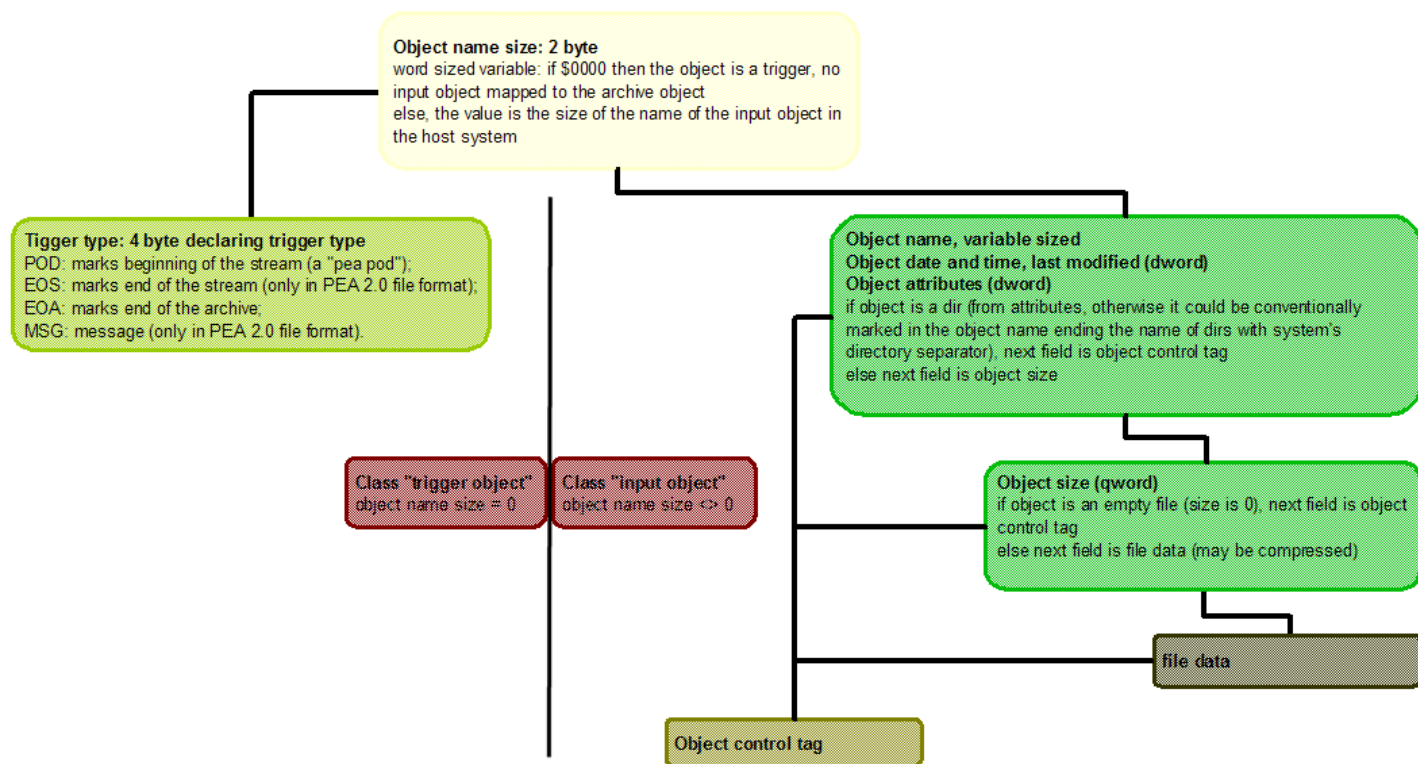
file data

Object control tag

Image 1: PEA version 1 revision 1 object taxonomy flowchart

Object control is performed on all input objects, on uncompressed and unencrypted content (only if the object is a nonempty file) and all associated data (qualified name, object last modification time, object attributes and finally file size if object is a file).

Stream level control instead allow to embed different objects in different archive areas (streams) each with different control features and/or, in case of encryption, with different passwords (not mandatory different, since unique salt is used for each stream). PEA 1.x file format allow archives containing a single stream while PEA 2.0 format allows unlimited streams in a single archive, hypothetically even a distinct stream for each distinct object, the space overhead would be of 16 byte per object (POD and EOS objects) plus size of cryptographic subheader and authentication tag.

Streams could easily be nested (simply making the EOS tag closes the last opened stream) however this will have a bad performance impact since for each nesting level stream check and object control check would be duplicated, so PEA 2.0 format define that streams will not be nested (it's necessary to close the n-th stream with EOS trigger before opening the n-th+1 stream).

PEA stream control doesn't apply to raw input data but rather to data after compression (if used), plus all associated data including object level control tags; EOS or EOA trigger is included.

The stream level check includes also POD object (not including, if used, salt and password verifier in cryptographic subheader since are jet part of the authenticated encryption) and, for the first stream (or for the sole stream for PEA 1.1 format), the archive header; in that way all data included in the archive is subject to stream level check.

If stream level check is checksum or hash based, content of that area (POD object and archive header for first stream) is passed to the function as first block to check.

If AE is used, that area is appended to the passphrase and passed to the key derivation function, so data corruption in the area will be noticed at key verification stage, avoiding the needing to perform the full decryption process with unchecked parameters.

Using AE at stream level check means also that digests of input objects are kept private, otherwise digests could be useful (for small objects, i.e. dirs, empty or small files) to cast meaningful guesses on object's content (data and associated data).

Volume level control is finalized and restarted for each volume and is performed on all volume's data after all PEA steps (so on encrypted and compressed data, if those features are used), allowing integrity check of each volume without needing to know other volumes and without needing to perform other PEA steps before volume checking, so it can be used to decide if to repudiate or accept an incoming volume due to it's integrity before starting unpacking the PEA archive, only providing that the check algorithm is known.

If the volume control algorithm is not known to the user (i.e. the parts didn't agreed on a communication standard defining a fixed volume control strategy), the first volume is needed since the check algorithm is declared in the archive header, which is not repeated in each volume.

PEA 1.6 file format details

Archive header, 10 byte:

    1 byte "magic byte" field for file format disambiguation: $EA;

    1 byte version number field;

    1 byte revision number field; ← this byte has been changed to 6 in 1.6 revision

    1 byte for volume control scheme;

    1 byte for archive-wide error correcting scheme;

    1 byte declaring the OS were the stream was built;

    1 byte declaring OS date and time encoding;

    1 byte declaring objects name character encoding;

    1 byte declaring CPU type (encoded in 7 bit) and endianness (in msb);

    1 byte declaring KDF work load for cascaded encryption, depending on the KDF (which is dependant by the strean encryption type, see below stream header section), if zero KDF will use default workload

- For PBKDF2 it is the number of extra KDF rounds (multiplied x100,000)
- For scrypt it sets memory and parallelism parameters as explained below in cascaded encryption chapter
- For hybrid KDF it sets memory / parallelism parameters for scrypt, and number of rounds for PBKDF2 as explained below in cascaded encryption chapter

Stream header, 10 byte:

    word object name size field, $0000, declaring the object is a trigger;

    4 byte trigger type filed: POD$00, marking beginning of the stream;

    4 byte area with declaration of stream's properties (4 1 byte fields):

        compression scheme;

        stream-wide error correcting scheme (zeroed, not implemented in 1.x PEA revision)

        stream control / encryption scheme;

        single objects control scheme;

- if Authenticated Encryption is used, the stream header is followed by
  - for single encryption, a 16 byte cryptographic subheader, similar to the one defined in Wolfgang Ehrhardt's FCA cryptographic application:
    - 1 byte zeroed (was FCA sig ($FC));
    - 1 byte zeroed (was flags field);
    - 12 byte (96 bit) pseudorandom salt;
    - word key verification field
  - for multiple encryption, a variable sized cryptographic subheader (each 16 bit sub-header area is composed as the previously described encryption header, unless differently specified)
    - 16 bit for first algorithm, with key verification zeroed
    - 16 bit for second algorithm, with key verification zeroed
    - 16 bit for third algorithm, with key verification equal to xor of the tree key verification words
    - 1 random byte (containing 0..127) providing the number of following random bytes (containing 0..255) to be discarded, meant to mask information of exact output size – this area is the first to be encrypted.
- If compression is in use, the stream header is followed by 4 bytes defining the size of the compression buffer (after encryption header, if applicable).

Area of objects belonging to the stream:

object name size: word;

if size is > 0, the object belong to "input object" class and following fields apply;

- object name (variable sized), last character is conventionally the DirectorySeparator if the object is a directory;
- object last modification time: 4 byte (possibility to restore this information vary between host systems);
- object attributes: 4 byte (possibility to restore them vary between host systems); if the object is a file the following fields apply:
  - file size: qword; if size is >0 the file is not empty and the following field apply:
    - file data (variable sized)
- object control tag (variable sized, depending from the single object control algorithm), generated from the all the featured fields between object name and object control tag (in other words, all object's data and associated data, uncompressed and unencrypted);

else (if name size is 0) the object belongs to "trigger object" class, the following filed apply;

- trigger type 4 byte;

In PEA 1.x there may only be declared EOA$00, triggering end of the archive: close the stream, check or generate the stream control tag, check or generate the last volume tag and stop;

Volume control tags break the flux of aforementioned fields appearing at –n bit from the end of each volume where n is the size of volume level control tag.

At the present state of documentation please refer to the sourcecode for details about values for declaring control algorithms, compression schemes and so on.

Differences between last and older revisions

PEA 1.6 shares all specifications with previous format revisions, but
- changes third byte in header to 6 to declare new revision's specifications are used
- introduces hybrid KDF combining scrypt and PBKDF2 for triple cascaded encryption; hybrid is selected by the algorithm parameter in the command line (see Algorithms chapter), tenth byte of the header specify the KDF workload as for other KDFs.

PEA 1.5 shares all specifications with previous format revisions, but
- changes third byte in header to 5 to declare new revision's specifications are used
- introduces scrypt as KDF for triple cascaded encryption as alternative to PBKDF2 supported by previous versions, in order to increase resilience against dictionary attacks on specialized hardware, increasing the resources needed to derive the key from the password (up to 1GB RAM per instance); scrypt is selected by the algorithm parameter in the command line (see Algorithms chapter)
- tenth byte of the header specify the KDF workload, if PBKDF2 is used the byte is composed as in 1.4 specifications, if scrypt is used the byte will specify the memory and parallelism level as explained in cascaded encryption chapter

PEA 1.4 shares all specifications with previous format revisions, but
- changes third byte in header to 4 to declare new revision's specifications are used
- uses tenth byte of the header to specify the multiplier (by 100,000) for the number of KDF rounds of cascaded encryption; the byte is zero (as in previous revisions) if unused – no extra rounds, or no cascaded encryption

PEA 1.3 shares all specifications with previous format revisions, but
- changes third byte in header to 3 to declare new revision's specifications are used
- introduces multiple cascaded encryption:
  - AES, Twofish, and Serpent, 256 bit in EAX mode, are used for encryption of each block after key verification tag and before stream authentication tag
  - 3 cascading modes supported: AES+Twofish+Serpent, Twofish+Serpent+AES, Serpent+AES+Twofish

PEA 1.2 shares all specifications with previous format revisions, but:
- changes third byte in header to 2 to declare new revision's specifications are used
- introduces support for BLAKE2S 256 bit and BLAKE2B 512 hash algorithms for object, volume, and stream level control

PEA 1.1 shares all specifications with 1.0 format, but:
- changes third byte in header from 0 to 1 to declare new revision's specifications are used
- introduces support to more EAX mode authenticated encryption options (AES finalists Twofish and Serpent algorithms, both 128 and 256 bit) for stream level control
- introduces support for newly designed SHA-3 256 and 512 hash algorithm (not available when PEA 1.0 specifications were published) for object, volume, and stream level control

Old Pea executable implementations built with previous level specifications cannot open PEA 1.3 files if newly supported algorithms are employed, so the third byte in the header (revision filed) is set to 3 to avoid files built with new format specifications being opened by an old executable implementation - it will be raised Error parsing archive header 9: PEA_REVISION_NOT_SUPPORTED
If needed it is possible to try to open 1.3 files simply setting third byte in header to a lower revision number, but file will be readable only if no new algorithm was employed – also, if encryption is used then tampering header data will be detected, making this solution not viable.

New Pea executable implementations built with last level of PEA specifications can open all PEA 1.x files created with previous specifications, so updating Pea executable is strongly recommended.

PEA 2.0 file format details (proposal)

PEA 2.0 file format allow unlimited number of streams, not nested, each one closed by a EOS tag but the last one, closed by EOA tag. First stream authenticates also the archive's header, as in PEA 1.0.
The format allows MSG type of triggers, meant for embedding meta-information where needed in the archive.
2.* file format is aimed to be retrocompatible with 1.*, being a PEA 1.* format archive a special case of the more general 2.* format, containing a single stream and no messages.
Please note that PEA 2.0 file format specifications are actually not finalized and may undergo to further revisions before being usable and used.
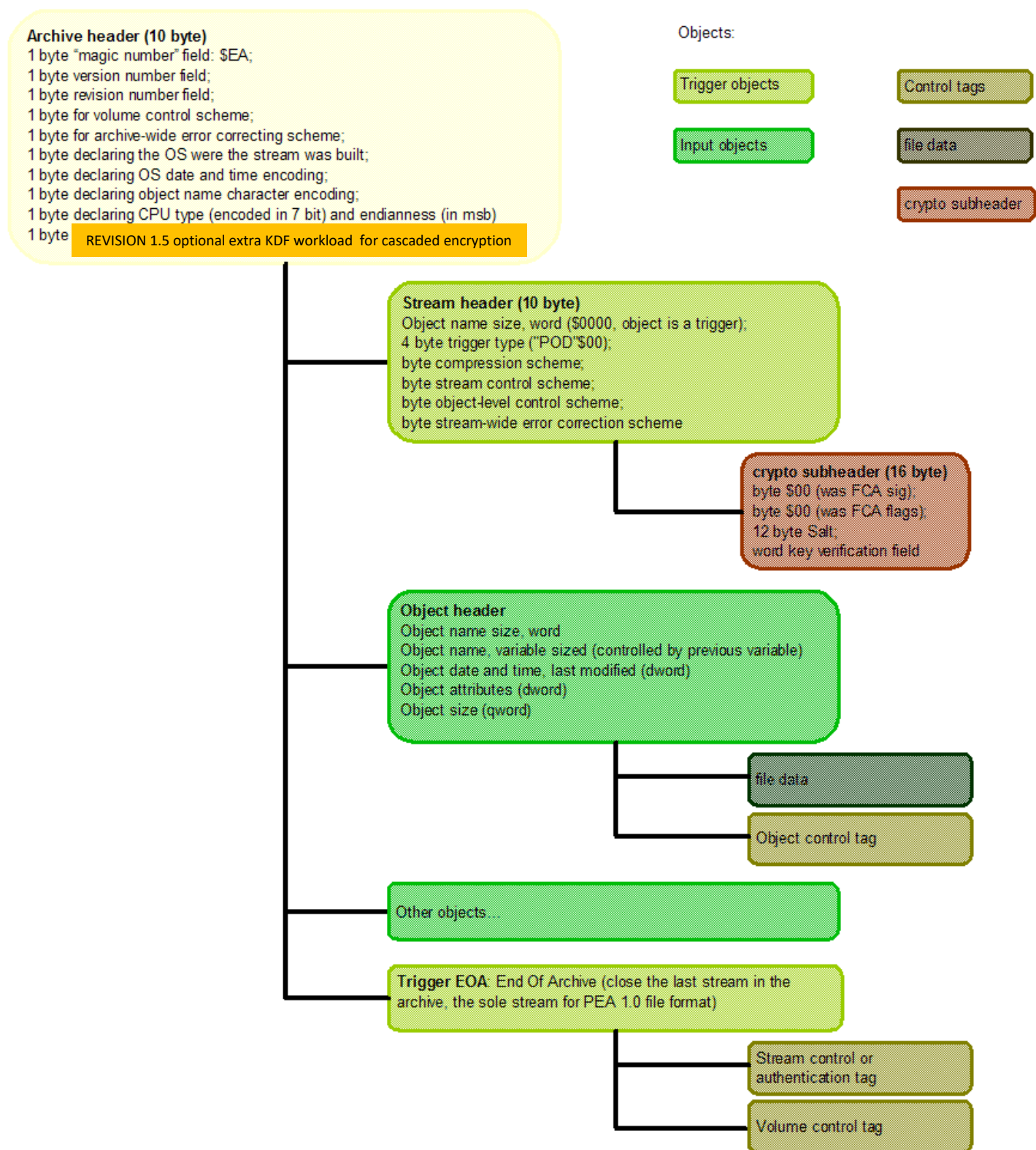
Archive header (10 byte)
1 byte "magic number" field: $EA;
1 byte version number field;
1 byte revision number field;
1 byte for volume control scheme;
1 byte for archive-wide error correcting scheme;
1 byte declaring the OS were the stream was built;
1 byte declaring OS date and time encoding;
1 byte declaring object name character encoding;
1 byte declaring CPU type (encoded in 7 bit) and endianness (in msb)
1 byte   REVISION 1.5 optional extra KDF workload for cascaded encryption

Objects:

Trigger objects

Input objects

Control tags

file data

crypto subheader

Stream header (10 byte)
Object name size, word ($0000, object is a trigger);
4 byte trigger type ("POD"$00);
byte compression scheme;
byte stream control scheme;
byte object-level control scheme;
byte stream-wide error correction scheme

crypto subheader (16 byte)
byte $00 (was FCA sig);
byte $00 (was FCA flags);
12 byte Salt;
word key verification field

Object header
Object name size, word
Object name, variable sized (controlled by previous variable)
Object date and time, last modified (dword)
Object attributes (dword)
Object size (qword)

file data

Object control tag

Other objects...

Trigger EOA: End Of Archive (close the last stream in the archive, the sole stream for PEA 1.0 file format)

Stream control or authentication tag

Volume control tag

Image 2: PEA version 1 revision 6 file format flowchart (archive saved as single volume)

PEA file format's and implementation's limitations

feature	PEA file format	Current implementation
Archive		
Max archive size	unlimited	up to 999999 volumes of 2^64-1 byte each; using 128 bit block encryption it would be safe not to encrypt more than 2^64 byte with same key, better staying one or more orders of magnitude below
Stream number	1.0: single stream; 2.0 unlimited number of streams;	Single stream (1.1 file format)
Output		
Security	Optional Authenticated Encryption, at stream level only.	
Integrity check	AE tag or hash or checksum at stream level, hash or checksum for input objects and output volumes	
Error correction	No scheme featured	
Communication recovery	Independent volume control check allow to identify corrupted volumes (first volume may be needed to know volume check algorithm)	No specific tool developed; volume check is done during extraction and then, allowing to repeat download only of corrupted volumes
Data recovery	Stream control tags allow to recognize correct streams, if better granularity is needed object control tags allow to recognize correct objects; input object names and POD trigger allow to identify objects and stream between the archive data;	No specific tool developed to try error resistant data extraction, however object check errors are reported to identify corrupted and non corrupted data if the extraction is successful
Support for multi volume output	Native, requires a single pass	
Volume number	1..unlimited	1..999999 (counter in output name)
Volume size	Volume tag size +1.. unlimited; first volume must contain at least 10 byte of data to allow parsing of the archive header, to allow unpacking application to calculate volume tag size	Volume tag size +1.. 2^64-1 (qword variable) ; first volume must contain at least 10 byte of data
Compression	Native, requires single pass; schemes: PCOMPRESS0: no compression; PCOMPRESS1..3 based on deflate using zlib's compres/uncompres, level 3, 6 and 9 respectively (see PCOMPRESS chapter)	
Solid archive	Not implemented compression modes featuring the possibility of creating solid archive	
Input		
Input types	1.0: files and dirs; 2.0: files, dirs, metadata stored as messages triggers	Files and dirs (1.0)
Input objects number	1..unlimited	Host system memory limited (input object list is stored in a dynamic array of strings)
Input object size (of single objects)	0..2^64-1	0..2^64-1
Input object qualified name size (size 0 mean that archive object is a trigger, no input object mapped to the archive object)	1..2^16-1	1..32K (exceeding needs, longer values are considered errors)
Metadata	Objects attributes and last modification time, optionally comments and any kind of meta content using messages	Save object attributes and object last modification time. Restore only object attributes (on Windows), nothing on *x

Notes on input object name character encoding

Current PEA implementation works only on ANSI-encoded object names (not support UNICODE encoding), however PEA file format allow a field in the archive header to declare the character encoding for object names.

With 16 byte-wide field for object name defined by PEA file format the room should be enough for any practical use, even with encoding using more bytes per character (bytes necessary to encode characters can be stored in an array of byte up to 64KB -1 byte); however implementing that feature would require a careful evaluation of validity of object names for the system where the archive is targeted to be restored.

Notes on volume sizes

- Volumes can be arbitrarily sized, however they need to be at least one byte wider than volume control tag, in order that at least one byte of input data is stored in the volume.
- Since the unpacking application need to know what is the volume control algorithm used, in order to be able to distinguish between input data and volume tag data, PEA file format defines mandatory that first volume of the archive must contain at least 10 bytes (so, sized 10+volume control tag size bytes) to allow all information in archive header (that contains volume control algorithms, that define volume control tag size) to be red at once.
- Volumes could have different sizes.

PCOMPRESS compression scheme

PCOMPRESS0 is a non compression scheme; it will simply copy data from the input to the output.
PCOMPRESS1..3 is a deflate-based scheme of compression that allows decompression of single blocks without need of decompressing preceding blocks: that slightly degrades compression compared to classical schemes (effect may be minimal, due to algorithm and block size chosen).
The output data structure allows fast access to arbitrary sectors knowing position in input data and makes easy to implement highly parallel compression and decompression routines (features actually not exploited in PEA file format and implementation).
Compressed data (from PCOMPRESS1..3) is organized as follows:
- buffer size field, dword; declare the size of buffer of uncompressed data to compress at once; it's needed to be declared a single time if multiple objects are compressed;
- until end of input data:
  - size of the compressed buffer (dword);
  - compressed buffer (variable sized field);
- uncompressed size of the last buffer

Each buffer is in-memory compressed with zlib's compress, level 3, 6 and 9 for PCOMPRESS 1, 2 and 3 respectively, and decompressed with zlib's uncompress, uncompressed size is equal to buffer size for all blocks but the last (declared in the last field), size of the compressed buffer is declared before each block allowing to know how much data is needed to be read for decompression.
If the compression result in expansion or no compression of the buffer, the compressed buffer is discarded and the uncompressed buffer is stored, the size of the compressed buffer will be set equal to the size of uncompressed buffer.
In decompression, finding an input buffer sized as the expected uncompressed buffer result in treating it as non compressed, no decompression is done and data is simply copied to output.
Expected uncompressed size is equal to buffer size for all blocks but the last, whose size can be calculated as residual uncompressed size (being the uncompressed size declared as file size in object header) or read from last field (uncompressed size of last buffer).
That allow to:
- spare CPU time when decompressing hard to compress files, since many buffers will simply need to be copied to the output, at the cost of simply verifying if declared compressed size = compression buffer size;
- assure that each output buffer will be at most sized as input buffer;
- for the aforementioned reason, the output will be expanded, in the worst case, by only the size needed for compression scheme's fields, not regarding the underlying compression algorithm.

The scheme would also make quite simple replacing the underlying compression strategy without changing the output data structure.

Algorithms used in PEA format

This is a quick description of algorithms allowed in PEA 1.1 file format definition; please refer to more specific publications for more detailed descriptions of the algorithm's features and performances.
Evaluations of relative algorithm's speed are taken from [6] and [7].
Algorithms available for objects, volumes and streams:
- Adler32: a very fast algorithm generating a 32 bit checksum, useful to detect casual data corruptions; it is internally called with **ADLER32** string: this name is used for invoking that algorithm in command line or procedures (see "Usage" chapter).
- CRC32: 32 bit checksum through cyclic redundancy check, it's slower than Adler and have different error detection characteristics, it's useful to detect casual data corruptions; internal name: **CRC32**.
- CRC64: 64 bit checksum through cyclic redundancy check, slower than CRC32, useful to detect casual data corruptions; internal name: **CRC64**.
- MD5: 128 bit hash, slower than aforementioned checksums but faster than other featured hashes. It's no longer considered cryptographically secure, however is still useful to detect casual data corruptions and, generating a longer tag than featured checksums, minimizes the probability to have two objects with same control tag, even in an huge archive; internal name: **MD5**.
- SHA1: 160 bit hash whose strength is actually disputed, it's slower than MD5; it's however fit to recognize casual data corruption and may be fit in near future for detecting even malicious data tampering against low profile opponents; internal name: **SHA1**.
- RIPEMD-160: 160 bit hash, similar in speed to SHA1, some concerns about his strength exist however it's still well reputed; internal name: **RIPEMD160**.
- SHA256/512: based on a newer SHA revision and generating a 256 bit and 512 bit tag respectively, are still reputed secure up to detect malicious data tampering, it's slower than SHA1: internal names **SHA256, SHA512**
- SHA-3 256/512: currently reputed to be the strongest choice between available hash algorithms, generating a 256 bit and 512 bit tag respectively: internal names **SHA3_256, SHA3_512**
- **BLAKE2S** (256 bit()) and **BLAKE2B** (512 bit): strong hash algorithm, SHA-3 finalist

- Whirlpool: based on heavily modified AES algorithm, generate a 512 bit tag, slightly speedier than SHA512; internal name **WHIRLPOOL**.

Please note that being the tag transmitted within the communication (the archive), all of those algorithms are meant to protect data integrity against casual data corruption rather than against malicious attacks, since the attacker could generate a new valid message/digest pair.

However, if the sender can publish the digests on a secure server the receiver can securely access, original tags can be verified, not allowing the aforementioned type of attack. Under those conditions choosing a cryptographically strong hash for generating the digest make unfeasible for the attacker to calculate a modification allowing the modified data to generate the same digest.

Algorithms available for streams:
- HMAC Authenticated Encryption using AES128 in CTR mode for the encryption: composition scheme providing the data in the stream is private and subject to authentication; internal name **HMAC**.
- EAX Authenticated Encryption using AES128 or AES256 in CTR and OMAC mode: a provably secure mode to provide privacy and authentication of the data relying on a single algorithm and a single key (in other word no longer two algorithms, but a single one, has to be proven secure to prove the mode is secure); internal name **EAX** (128 bit key size), **EAX256** (256 bit key size).
- EAX AE using Twofish (128 and 256 bit) in CTR and OMAC mode; internal name: **TF**, **TF256**.
- EAX AE using Serpent (128 and 256 bit) in CTR and OMAC mode; internal name: **SP**, **SP256**.
- Triple cascaded encryption combining AES+Twofish+Serpent, Twofish+Serpent+AES, Serpent+AES+Twofish, each with 256 bit key in EAX mode, generating each a 128 bit authentication tag.

  In the command line those algorithms are referred as
  - **TRIATS**, **TRITSA**, **TRISAT** ("tri" followed by first letter of the algorithms in the order they are performed) if using PBKDF2 key schedule, followed by the extra KDF rounds parameter (0 to 255 each unit adding 100K rounds to KDF for each algorithm) which can be omitted for 0 (standard number of KDF rounds are performed – example: TRIATS10 performs AES+Twofish+Serpent and adds one million rounds to KDF of each algorithm.
  - **SRIATS**, **SRITSA**, **STRISAT** ("sri" followed by first letter of the algorithms in the order they are performed) if using scrypt key schedule, followed by the extra KDF workload level 1 to 7 (from 64 MB memory default, 128 MB memory at level 1, up to 1 GB memory and parallelism level = 8 at level 7)
  - **HRIATS**, **HRITSA**, **HTRISAT** ("hri" followed by first letter of the algorithms in the order they are performed) if using hybrid key schedule, followed by the extra KDF workload level 1 to 7 (from 64 MB memory default + 75,000 KDF rounds, 128 MB memory + 275K KDF rounds at level 1, up to 1 GB memory and parallelism level = 8 + 25M+75K KDF rounds at level 7).

Please note under current understandings 128 bit sized keys are commonly regarded to be probably lifetime secure (or at least secure for some decades) against bruteforce under foreseeable technology advance, 256 bit sized keys makes bruteforcing $2^{(256-128)}= 2^{128}$ times harder making them probably secure forever against bruteforcing.

All Authenticated Encryption modes generate a 128 bit tag for authentication, regardless the key size.

Being the tag verification key dependent, it is possible to verify it only knowing the key, so an attacker will not be able, given the message, to recalculate the tag until the key remains secret.

PEA security model

PEA security model assumes the application to run on a trusted environment both during the archiving and unpacking stages, aiming to protect solely the data during a communication process: the protection apply when a PEA archive is sent to the receiver and end when the archive is opened by the receiver.

PEA implementation cannot assure that the system is not subject to attacks that compromise the security of the data at a lower level allowing bypassing protection (encryption, authentication error checking) provided by the application, like i.e. keylogger logging the password provided by the user, or malware as viruses or rootkits exposing directly the clear data.

The implementation doesn't save temporary data, but cannot assure that the system will not cache data used by the application to the disk.

Aim of PEA is offering a flexible security model given the three different levels for data control.

Object level integrity checking is done on raw input data to detect errors with object level granularity; if for some improbable but not negligible cases (memory banks errors, bugs etc) errors are introduced with compression or encryption steps, object level control is able to detect that event for maximum security of operation.

If encryption is used (at stream level), object level tags get encrypted, not allowing to cast guesses on object's content independently from the object's size.

Stream level check offers up to authenticated encryption feature, protecting privacy and authenticity of a group of objects with same security needs, including tags generated by object level checks. That allow to don't need a cryptographic header and authentication tag for each object transmitted (a sizeable space overhead) and to don't need to finalize and restart encryption more times than needed, since it poses serious issues to collect enough entropy to seed each encryption and costs many CPU cycles especially in the Key Derivation and seed generation phase (see [1] analysing WinZip's AE2 scheme, that is object oriented).

Volume level integrity check are communication oriented and allow to discard single corrupted volumes in order to minimize, in case of error, the overhead of resending or re-downloading the data. Volume level integrity check is performed on the data in its final form, encrypted and compressed if those features are used, that means that volume tags, independently form volume size, cannot be used to cast guesses on volume's plaintext content if encryption (at stream level) is used.

Moreover, if compression is used, the structure of the data subject to volume level check is rather different from the one subject to object level check, allowing a redundant error detection on two rather different data structures.

That mean that each of the two levels may identify kinds of errors that pass undetected in the other level, due to the different structure of the data checked, even using the same algorithm for objects and volumes; of course, chosing different algorithms may increasing furtherly the ability of error detection of the system.

Examples of use:

Alice and Bob need to exchange data as fast as possible on a wide, secure and not error-prone channel.
In those ideal condition all checking and compression features could be turned off with process speed similar to copying files from one location to another, resulting in consolidating the n input in m output volumes to fit the destination space constrains (i.e. destination is a removable archive, or destination filesystem support limited file size).

Alice and Bob need to exchange non-private data and want to be able to know if the transmission was corrupted or tampered, in which case they want simply to discard the data and repeat the communication.
All those conditions can be accomplished with the sole stream level control, all data transmitted is checked and in case of error anywere the check will fail to match.

Alice and Bob need to exchange private data at the aforementioned conditions.
It can be accomplished using authenticated encryption all data (as input data and all associated content) is kept private and checked with cryptographically strong authentication to guarantee is neither corrupted nor tampered and that it comes from the sender that decided the given password.
Multi stream archives (PEA 2.0 file format only, actually not implemented) may allow different input object group to be subject of different kind of stream level control, allowing i.e. using different password or different algorithms for different streams.

Of course, with this security model a single corrupted byte can result in needing to repeat the whole communication (or the whole stream), such model should not be used when transmitting data on channels prone to data corruption or a severe overhead for retransmitting the data will be imposed, proportional to the stream size and to the probability of data corruption.
An efficient countermeasure will be in using volume control check: in this case a corruption event will impose the need of resending only the interested volume(s) rather than the whole communication; decreasing the size of the volumes will make the event of corruption of a volume less costly.
In example Alice may publish on a ftp server m volumes, when Bob's download of a volume gets corrupted he need only to re-download the given volume rather than all the volumes; Alice can chose a volume size and nature of the checking algorithm fit to efficiently deal with corruption probability.

A severe communication problem or media failure result in the corruption most or all volumes, moreover Bob may not have the possibility to request Alice to resend the data and need to try to extract all single objects that result not corrupted.
In this critical case object level check can help Bob to recover non corrupted objects from Alice transmission, since each embedded object has its own control tag.
However see "Data Recovery" chapter below that analyse in details the practical issues in recovering data from a corrupted PEA archive (if it's not possible to resend it), since data corruption may be relevant in preventing the possibility for a program to extract data at all from an archive, altering the fields structure needed to automate the data extraction.

Examples of use against attackers, using stream level AE:

The eavesdropper Eve try to get useful information from object level tags
Object level tags are not accessible in clear, so no information can be recovered without decrypting the archive.

The eavesdropper Eve try to get useful information from volume level tags
Volume level tags are in clear, but are calculated on data in encrypted form, so no information is leaked about plaintext.

The malicious active attacker Mallory try to modify the archive's or stream's header area to see what happens if the decryption program runs with unproper parameters (control algorithms, compression scheme etc) or to mount a DOS attack on Bob.

All data in headers area, except salt and key verifier, are appended to the password and run, along with the salt, in the key derivation function; decryption is then tested with key verifier.

That makes cryptographically hard, if the key derivation function is secure (PEA uses PBKDF2, widely accepted to be secure), to find a combination of salt and password (that includes the header's data) that matches the key verification value.

Since key verifier is word sized, that means that a modification in archive's or stream header (including salt and key verifier, since used in a secure way in the verification procedure above described) has $1/2^{16}$ probability to pass undetected to this stage.

Moreover, only few header are possible since non valid header will make the archive to don't even be opened.

Since only one message out 2^{16} will run the full decryption and authentication process, this attack has a low impact on Bob, that has two means (header validity check and key scheduling level check, the latter being cryptographically strong) to discard most of the wrong headers without spending the full decryption time.

Mallory can decide to mount a DOS attack tampering with all the communication he can intercept imposing Alice and Bob a severe overhead for repeating several communications.

Case1: private communication

Mallory could alter a volume and then substitute the volume tag with a matching one: the tampering will be reported at stream level since the altered data will not allow stream's authentication, but it will not be possible to identify the volume needed to be retransmitted.

However, if object level integrity check is used, that action will modify one of the embedded objects or related object level control tags (both protected by encryption), making them no longer match. The fact will be reported as an object level error, allowing a) to identifying tampered content b) to probably narrowing the field of possible tampered volume(s).

To avoid the detection of a tampered object Mallory should know all the content of the object and its exact position in the archive in order to be able to calculate the original control tag and the tampered one and to alter in the proper way (i.e. by difference with original values, if CTR mode encryption is used, since bit position of plaintext is not altered) the object data and related object's control tag areas.

This level of knowledge is unlikely, since object names and all related metadata, including size and control tag, are encrypted, but not impossible, since Mallory could by hypothesis have complete knowledge of all the objects need to calculate were exactly the desired object reside in the archive; this make the event non trivial but not non feasible.

If Mallory succeed in hiding both volume's and object's tampering, archive will however fail the authentication but it will not be possible to reduce the field of data to be discarded, and the DOS attack would have been successful.

The only cryptographically strong fix for this case of DOS attack is in providing authenticated encryption features for the volume level, which is actually not implemented.

Mallory could even switch the order of the volumes in the archive, since volume sequence is declared in clear (in the volume's name), however this will lead to fail the stream's authentication.

Probably the integrity checking of objects partially embedded in the volume (i.e. at the beginning and at the end), will fail, but as shown before Mallory could have enough knowledge of the objects to fix the problem of object level checks.

In both case the attacks doesn't reduce the security of the encryption nor of the authentication of the stream, but may impose to Bob the overhead to discard the whole transmission.

Case 2: publication

Issues in Case 1 becomes not meaningful when the data get published and hash values are i.e. exposed for control on a public server; if Bob can access the server securely (out of the scope for PEA security model), the hash value can verified and Mallory cannot rely on creating a new valid modified message / hash pair; the only possible attack remaining is finding a collision (modified message that generate the original hash), that is unfeasible if a cryptographically strong hash was chosen.

The volume sequence swapping attack becomes unfeasible too, unless two volumes have the same hash value, however the attacked archive will still fail at least the stream level authentication.

Cryptanalysis of PEA format

This chapter was written mainly applying to PEA file format the observations made by Tadayoshi Kono [1] on WinZip Authenticated Encryption scheme, so please note that it may be far than exhaustive since the cited study is actually focused on zip format, not on PEA format.

Basically, the difference between zip's AE-2 encryption scheme and PEA's one are:
- PEA features uses EAX mode along with classic composition method; EAX mode security for both privacy and authentication was clearly proved (providing the underlying cipher is secure) in [2];
- PEA encrypt a group of objects (and authenticate them and the salt that was used) as a single encrypted stream so it's not needed to encrypt each file separately; moreover PEA 1.1 file format doesn't even allow to make archives with multiple streams: those factors makes infeasible the attacks described in [1] in chapter 7, 8 and 9:
    o 7: archives with mixed content of encrypted and unencrypted files - such case is not possible with PEA 1.1 file format since only one stream is possible;

- o 8: insufficient entropy in salt that may lead to keystream reuse; and 9: concerns in making easier dictionary attacks - becomes infeasible since salt is generated a single time, see below for implemented entropy sampling scheme in current PEA executable implementation;

That also leads to:

- having the encryption initialised only once means offering higher performances when encrypting many small files and allowing choosing slower key derivation and salt generation functions, if desired, with generally smaller impact on overall performances.

- don't have the overhead of the salt and the authentication tag for each single object archived, resulting in a smaller output size overhead for cryptography related data.

- PEA encrypt all associated data of each input object (name size, name, last modification date, attributes, size and optional checksum/hash of uncompressed, unencrypted object), not only file content, making infeasible attacks described chapters 3 (leakage of data) and 5 (exploiting filenames associations)

- Attack described in chapter 4 changing compression scheme and output size to produce garbage, but unencrypted, output: being the headers data appended to the user-provided passphrase and sent to key derivation function, only $1/2^{16}$ modification will pass the key verification stage, and even in this case the modification will make the authentication fail (unless finding a collision in the 2^{128} authentication tag field);

Moreover objects sizes are encrypted and authenticated, trying to modify them will result in stream failing the authentication;

It's possible to make this kind of attack even more unfeasible using object level integrity check, being the check tag subject to authenticated encryption:

- o if the attacker try to modify the object level control tag to match with the new expected output, the stream will not authenticate and the user will be warned to don't trust it;
- o if the attacker doesn't try to modify the tag, the object level check will report the error (unless the attacker can find valid PEA header settings producing a collision for the object level check algorithm and the given object, that's quite unfeasible);

It would be however a nice security policy to delete by default objects failing the authentication/integrity check in order to prevent the user to be tricked by the attacker to misuse them.

Other important details about operations for generating a .pea archive, as featured in current PEA implementation:

- PBKDF2 key derivation is used for keying; SHA1 160 bit hash is used as primitive function in the kdf for generating keys for 128 bit encryption, while Whirlpool 512 bit hash is used as primitive in kdf for generate keys for 256 bit encryption.
- Optional two factor authentication is offered: a passphrase (may contain any typeable character) is mandatory, a keyfile can be optionally used: any filetype can be specified, although it's strongly recommendable to use random generated files. Passphrase, archive header and the content of the keyfile are concatenated and sent as an array of byte to the PBKDF2 key derivation function.

Random number generation in current PEA implementation was developed trying to take in account basic concepts contained in literature, as in [8].

Please refer to pea_utils unit, in the section "functions related to keying, salting and entropy collection" for more details; in brief:

- The random number generator, from 0.11 release, uses a 2048 bit sized (256 byte) persistent randomness collector to propagate randomness collected to following sessions of use; Pea and PeaZip (a frontend for Pea executable) save that session seed as "rnd" file in "res" path.
- A "fingerprint" of the system is taken at program start, from several system- and session-specific variables, some changing slowly some others changing quickly, in example: environment variables; current disk size and free space; system timers and CPU cycle counter; memory status; PID; last OS error; list of temporary file names, attributes, file dates/times; etc… values are collected and hashed, with the fingerprint being the digest of the hash of the system's entropy pool.
- Three user dependent entropy sources can be initialised:
  - o Mouse: at each mouse movement coordinates and timing (and memory status for Windows only) are used to update mouse's entropy pool;
  - o Keyboard: each time a key is pressed the key and the timing (and memory status for Windows only) are used to update keyboard's entropy pool;
  - o Files: when a file is opened for collection entropy purpose, content, name, timing (and memory status for Windows only) etc are used to update file's entropy pool;
  The entropy pool update process is updating the hash status with newer sampled values.
- When the generation of a random number (key, salt…) is requested, the three user dependent pools are finalized (the digest of each hash is calculated); and other two digests are created:
  - o a time digest (digest of hashing of timing and CPU cycle counter;
  - o a memory status digest (Windows only) generated when the random number generation is requested;

The random number generation function usually passes the values of those 6 digests (fingerprint, mouse, keyboard, files, time and memory) and the data from the persistent randomness pool through an hash and/or to seed cryptographic prng and csprng functions.

Addendum: cascaded encryption

PEA 1.3 introduced support for multiple encryption, triple cascading encryption with AES, Twofish, and Sepent, 256 bit in EAX mode.
Revision 1.4 added optional variable extra rounds for KDF.
Revision 1.5 added scrypt KDF as alternative to PBKDF2 for triple cascaded encryption, increasing resilience to dictionary attacks.
Revision 1.6 added hybrid KDF for triple cascaded encryption using scrypt for AES and Twofish (with comparable workload but different parameters) and PBKDF2 for Serpent.
While PBKDF2 can only increase the CPU workload adding iterations of the hash primitive, scrypt also increases the memory requirements in order to make harder and more costly to build specialized equipment for password recovery.
Hybrid KDF uses two instances of scrypt with different r block size parameter (requiring different memory and CPU optimizations) and one instance of PBKDF2, which requires the attacker to build different specialized equipment for password recovery further increasing the cost of attacking the KDF stage of encryption.

- **Keying**
    - o Each cipher is separately keyed through PBKDF2 or scrypt
    - o Following steps are taken to ensure the 3 keys are statistically independent after the key schedule:
        - ▪ Well established key derivation functions alternative are available: PBKDF2 and scrypt (please note that the KDF will be anyway repeated 3 times, one per algorithm, further increasing the workload)

PBKDF2 sets the key schedule of each cipher on a different hash primitive which is run for a different number of iterations: Whirlpool x 25000 for AES (Whirlpool is significantly slower than other algorithms), SHA512 x 50000 for Twofish, SHA3-512 x 75000 for Serpent (SHA3-512 is faster than SHA512)
Revision 1.4 introduces an optional variable number of extra rounds added to KDF of each algorithm, encoded as 0..255 multiplier (x100,000 rounds) in the tenth byte of the header allowing up to 25,500,000 rounds (25,5 millions of iterations for each algorithm)

scrypt (introduced in 1.5 revision) uses 64 MB memory per instance, with r parameter = 8 and p parameter = 1, KDF work flow can be increased as specified in the tenth byte of the header
- 1 will use 128 MB RAM, with r = 8, p = 1
- 2 will use 256 MB RAM, with r = 8, p = 1
- 3 will use 512 MB RAM, with r = 8, p = 1
- 4 will use 1 GB RAM, with r = 8, p = 1
- 5 will use 1 GB RAM, with r = 8, p = 2
- 6 will use 1 GB RAM, with r = 8, p = 4
- 7 will use 1 GB RAM, with r = 8, p = 8

Hybrid KDF (introduced in 1.6 revision) uses scrypt for AES (as specified in scrypt section) and for Twofish (with half the N parameter and doubling the r parameter, same p parameter), and uses PBKDF2 for Serpent (75,000 iterations as specified in PBKDF2 section). KDF work flow can be increased as specified in the tenth byte of the header
- 1 will use 128 MB RAM for scrypt KDFs, and +200K iterations for the PBKDF2 KDF
- 2 will use 256 MB RAM for scrypt KDFs, and +500K iterations for the PBKDF2 KDF
- 3 will use 512 MB RAM for scrypt KDFs, and +1M iterations for the PBKDF2 KDF
- 4 will use 1 GB RAM for scrypt KDFs, and +2M iterations for the PBKDF2 KDF
- 5 will use 1 GB RAM, with p = 2 for scrypt KDFs, and +5M iterations for the PBKDF2 KDF
- 6 will use 1 GB RAM, with p = 4 for scrypt KDFs, and +10M iterations for the PBKDF2 KDF
- 7 will use 1 GB RAM, with p = 8 for scrypt KDFs, and +25M iterations for the PBKDF2 KDF

- ▪ Key schedule of each cipher is provided a separate 96 bit pseudorandom salt
- ▪ Password provided to key schedule is modified for each cipher into three sub-passwords *
    - modification are trivial xor of password with non-secret values and counters, with the sole purpose to initialize the key derivation function with a different values, being an additional factor (alongside different salt, and - in case of PBKDF2 - different hash / iteration number) to further guarantee the keys derived from password input are statistically independent - the details of the operation are available in the source code.
- **Password verification** tag is result of xoring the 3 password verification tags of each encryption function, and is written / verified after all 3 key initialization functions are run – not allowing separate verification for each algorithm
- Variable sized sub-header area meant to be discarded: first byte contains pseudorandom 0..127 number of residual bytes to be discarded, all other bytes in the area contains 0.255 pseudorandom data. This area is the first one to be encrypted / decrypted.
    - o This area is meant to mask exact size of archive, which can mitigate oracle attack toward compressed-then-encrypted data - in some scenarios (as in BREACH and CRIME attacks) the output size can leak relevant

information about compressibility of the input given specific modifications of the uncompressed plaintext; those type of attack is however unlikely due typical usage of Pea format for archiving tasks.
- Each block between password verification tag and authentication tag is encrypted with all 3 ciphers
- Each cipher generate its own 128 bit sized **authentication tag**, tags are concatenated and hashed with SHA3-384; the SHA3-384 value is written / checked for authentication – this requires all the 3 tags to match to expected values and does not allow to authenticate each cipher separately.

* PLEASE NOTE: the same password, trivially modified by fixed rules to originate 3 sub-passwords, is used for the 3 algorithms
This means that - even if the 3 keys are made statistically independent after the KDF stage - it is still possible to try to brute-force (or dictionary attack) the KDFs by guessing the password and calculating the 3 sub-passwords, using the simple aforementioned rules, to run the 3 KDFs at once.
However, as no separate key verification is available, it is not possible to mount an attack against one of the three 2^256 bit keyspaces to recover the sub-passwords.
Only correctly guessing the password (whose complexity is up to the user) will allow to correctly initialize all the 3 256 bit subkeys.

Multiple encryption, if correctly implemented, is meant under current understandings to:
- Provide a larger keyspace than each single cipher (2^256), but smaller than the sum of the lengths of keyspaces (2^256x3) due possibility of meet-in-the-middle type of attacks, reducing the computational complexity of the attack at cost of memory usage
  Such large keyspace may be overkilling even in event of significant quantum computing advancements: Grover's quantum algorithm which is the best-possible known attack for NP-complete problems provides a quadratic speed-up over a classic computing. Under those assumptions, as a role of thumb, a quantum computer will be able to brute force a 256 bit keyspace not faster than a classic machine can brute force a 128 bit keyspace, which is currently considered safe by a wide margin.
- Provide a security margin even in case all but one of the algorithms used as cipher (or key schedule hash) is compromised by a breakthrough in cryptanalysis, which seems unlike due the amount of theoretical work and real life testing behind mainstream primitives available today.

As drawbacks;
- The inherent added complexity makes multiple encryption more prone to implementation errors - anyway multiple encryption is quite thoroughly understood by current cryptanalysis, making such errors easier both to avoid and to spot.
- Performing multiple algorithms requires more computing power and consequently reduces performances, in this case by a factor of 3.
  This drawback may be decisive for some classes of applications, but in case of file archiving as for PeaZip, where many other operations (potentially far slower as read / write to disk) are involved, the performance hit is quite reasonable, and in case of KDF - slower better, the more hardware resources intensive the better - even desirable.
  o Test machine: notebook with Intel Core i7-8565U CPU, 4 physical cores with hyper-threading (8 logical cores), 8 GB RAM, 512 GB PCIe NVMe SSD, NTFS filesystem
  o Benchmark creation of PEA archive from 100MB input
    ▪ 7 seconds archive creation, 3 seconds archive extraction with AES256 EAX, deflate compression, CRC32 and SHA3-256 integrity checks
    ▪ 8 seconds archive creation, 4 seconds archive extraction with Serpent 256 EAX, deflate compression, CRC32 and SHA3-256 integrity checks (slower than AES and Twofish)
    ▪ 10 seconds archive creation, 6 seconds archive extraction with AES+Twofish+Serpent 256 EAX, deflate compression, CRC32 and SHA3-256 integrity checks – the purposely slower key schedule, employed at startup for multiple encryption modes, also account for the extra time

Data recovery from PEA format

Information in this chapter doesn't mean to matter for common usage of the program, becomes relevant only when rather uncommon errors or willing tampering of the data occurs.

A true data recovery mechanism with Error Correcting Code fields (i.e. like, optionally, in RAR format) is not integrated in current PEA archiver implementation, however PEA file format allow to specify an archive-specific and a stream-specific error correction scheme for future usage.

It's however possible to use PEA executable in cascade with other applications that generates ECC fields, in order to overcome this limit when data integrity is crucial, i.e. if a single copy of the archive exist and it's not possible to download it again or recover it from a backup.

On the other side, error and tampering detection is strong and can be flexibly adapted to the user's model of treat about data corruption or forgery, making extremely difficult that a casual or intentional modification get unnoticed.

As trade-off of PEA approach it's generally not possible to partially extract and authenticate a single object from a stream since it is the whole stream the subject of authentication and all the data embedded in the stream need to be correct to obtain a positive authentication.

Volume level integrity check can help to find in which volume the problem is, otherwise object level check can allow extraction of data with object level granularity, discriminating successfully and unsuccessfully object-level integrity checked objects if it's acceptable for the user's treat model.

From the point of view of the implementation of the extraction program (UnPEA), it's critical also to understand how errors may impact with program operations:

- If encryption is used, errors in archive or stream header, in encryption salt or in the key validation field will bring to unsuccessful key validation, stopping the extraction from the encrypted archive. Generally this problem cannot be fixed, unless the user made a backup of the headers area, including cryptographic subheader (first 36 byte of the file);
- If encryption is not used errors in functional fields in the archive or stream header (fields defining archive's and stream's properties) can be detected only at stream check stage; driving UnPEA to use wrong settings in extracting the archive, they may prevent the application to work properly, not allowing to complete the extraction and the check of the content. The backup of the header material can fix that kind of errors, otherwise the header functional fields may even be rewritten manually using an hex editor and looking for reference the documentation or sourcecode to find values to write in each field.
- Errors in the file data generally allow total archive extraction (allowing further examination of the data), unless some bits are lost or added and file size is altered, making not possible to automatically find where the next file begin and so triggering errors of different kinds;
- Errors in file age and file attributes are not meaningful if UnPEA application is told to reset those attributes, otherwise incorrect date or attributes could be not accepted by the host system that may not allow to save the object;
- Errors in the declaration of the width of a variable sized field (file name size, file size, size of compressed and uncompressed block when PCOMPRESS1..3 compression is used), generally make not possible to find were next fields begin not allowing the automatic extraction process to continue, however if the data is not encrypted, or when encryption is removed, objects could be extracted using an hex editor since the object names are human readable and may help to find the desired data in the archive.

  Current implementation consider wrong input names longer than 32K characters (exceeding actual needs), warning the user and stopping; that mean that an error in input name size (word sized field) has ½ probability to pass undetected to this check, for each object in the archive.

  If PCOMPRESS1..3 compression is used, each block of compressed data is preceded by it's compressed size (4 byte field, allowing values up to 2^32-1 byte); since for PCOMPRESS* scheme output (compressed) blocks of data can have mandatory at most the same size of input (uncompressed) blocks of data, it's easy to check against most errors in those areas: if compressed size is declared bigger than the buffer size (in the current implementation 1 MB), it's certainly an error (so for the current implementation a random error in those fields has about 1/4000 probability to pass undetected).

  Moreover, the last field of a PCOMPRESS1..3 compressed files declare the uncompressed size of the last buffer (since it probably don't match the compression buffer's size), which is checked against the size of residual data expected due to the size declared for that file; if they differs a decompression error condition is raised and operation is interrupted.

Current UnPEA implementation will try to intercept error conditions and in case of unexpected types of errors will try to save an automated job report to allow further analysis.

Error checking volumes before extraction may be a way to avoid operational problems triggered by data errors to the extraction procedure; however this approach has two shortcomings:

- doing so will require the extraction procedure to run not synchronously all operations (anticipating the volume level checks), which leads to performance drawbacks;

- volume integrity check is meant to detect casual data corruption, if the volume was forged the volume tag may have been recalculated and replaced, making useless this approach (that issue doesn't apply if volume's hashes are published and safely accessible from the receiver).

Section 2: PEA implementation

Usage

PEA executable doesn't feature a way to input parameters directly into the application's graphical interface (except, when applicable, for passphrase and keyfile).

It is the "engine" performing actual work on data and can be invoked from a user-friendly GUI (i.e. PeaZip, from the same Author; for usage please refer to PeaZip's documentation), or can also be launched from command line and batch scripts. In both cases PEA executable elaborate the parameters received from calling application or script.

For using PEA features as a library, within third part software, please refer to "Notes for development" chapter.

PEA featured functions are:

- PEA: archives files and directories (even entire filesystems can be passed); compress, encrypt, authenticate, error check and split in multiple volumes the output (a .pea format archive) in a single passage, with extensive job report;
- UNPEA extract and check .pea format archive, with extensive job report;
- RFS "raw file split": byte-split an input file "as is", optionally featuring integrity checking through external control data, keeping full compatibility with other raw file splitting/joining applications (*x split, hjsplit, File Tools 1.x etc);
- RFJ "raw file join": joins raw split files (featuring also integrity checking if files were splitted with RFS);
- WIPE: securely delete files and folders;
- SANITIZE: securely delete free space;
- COMPARE: byte to byte compare two files;
- CHECK: perform selected checksum and hash algorithms on selected files;
- ENVSTR: display host system's environment variables' strings;
- LIST: list files and folders (legacy, superseded by CHECK);
- HEXPREVIEW and TEXTPREVIEW: preview input file as hexadecimal or text file
- CRC32SAVE, MD5SAVE, SHA1SAVE, SHA256SAVE, BLAKE2BSAVE: directly save crc/hash value(s) of input file(s) in a file compatible with GNU Coreutils hashing utilities
- SHA256G, SHA256J, SHA256V: search SHA256 hash value of input file on Google (G), Jotti VirusScan (J), or VirusTotal (V) databases for possible match with known values / known malware
- MOTW: check for Zone.Identifier (Mark of The Web) metadata in NTFS Alternate Data Streams of input files. Directories are expanded recursively to test contained files. MOTW test stops at first positive match. Finding MoTW metadata means at least one file in input comes from a remote machine, which may be relevant for some user's security models.
- MOTWFULL: as MOTW, but test all input files (rather than stopping at first positive match) and displays results in a table showing which files have Mark of The Web metadata and which does not.
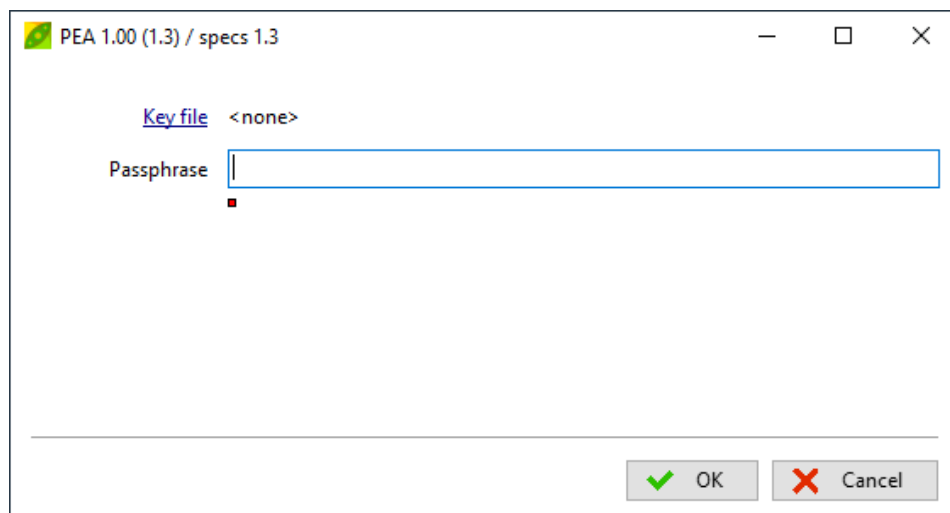

Image 3: PEA asking for password; in the form's caption is declared: the release and the PEA file format supported.

When the program starts, the user may be requested (if the stream control algorithm and the mode of use of the program request it) to type a passphrase (and retype it in the verification field if creating a PEA archive) and optionally select a file as key (it's strongly recommended to use a randomly generated keyfile).

The passphrase may contain spaces and any typeable char but remember that an user may have some troubles in figuring how to input some special characters specific to another language keyboard.

The user will need to remember the passphrase (and to have the key file, if used) to be able to open an encrypted archive, there is no key escrow mechanism in PEA.

PEA will try to give a rough rating about the entropy entered with the passphrase, the higher the better (until the passphrase can be memorized!) however note that PEA obviously cannot figure if the text is easily guessable or not.

When creating an archive using encryption PEA also needs to generate an unique salt, it uses system's and session's variables, timers etc for salting with unique and unpredictable values the salt generation procedure; if keying material (passphrase/keyfile) is entered interactively (i.e. as in case of PEA launched by PeaZip) mouse movements (coordinates and timings) are used to improve the amount of entropy introduced in seed generation process.

PEA executable try to esteem the amount of free space needed for requested operation (that phase is usually needed to make PEA capable of correctly reporting operation's progress); if during execution of the program the free space results not sufficient the application will ask for user's intervention, i.e. clean up destination drive, or change support if output is directed to a removable drive.

When the program ends the execution, most relevant information are shown in the form and a label "job log" can be clicked, opening a second form displaying more detailed information about input, output and program settings; the job log can be saved to a plain text file as tabulated text that can be easily imported in most analysis software, i.e. a spreadsheet, for a more convenient displaying.

On Windows builds is also shown a label that allow to open the object or to explore it's path.
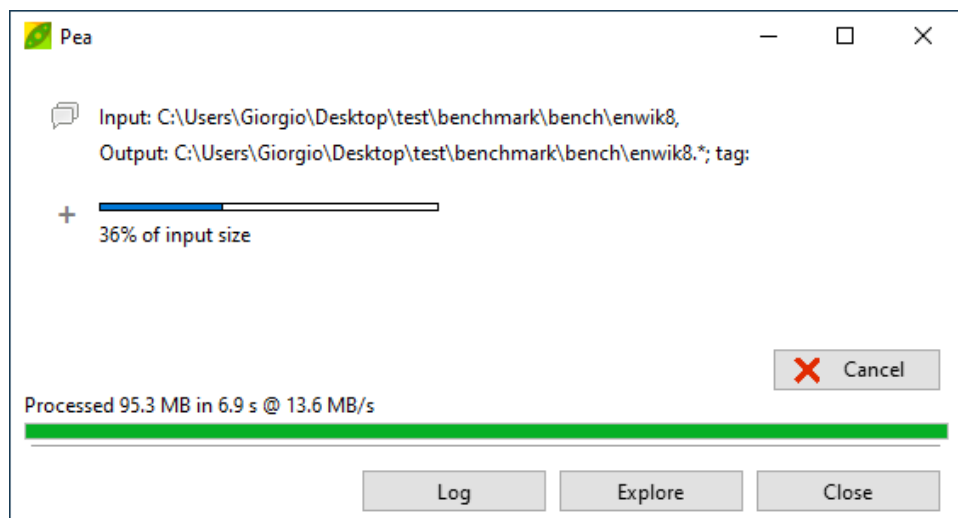

Image 4 PEA job log

Screenshots in images 3 and 4 refer to PEA executable performing "PEA" feature, the creation of archive in .pea format, however general concepts are applicable to any of the operations featured by the application

Command line parameters

RFS

Command line parameters	Example
Executable name	pea
Feature called: RFS, **raw splitting of a file**	RFS
Output full qualified name or AUTONAME to automatically derive it from input name	AUTONAME
Volume size in bytes	123456
Volume control algorithm	WHIRLPOOL
Mode of operation	BATCH
Input full qualified name	"c:\testdir\a1.txt"

RFJ

Command line parameters	Example
Executable name	pea
Feature called: **join raw splitted files**	RFJ
Input: full qualified name of first volume	"c:\testdir\a1.txt.001"
Mode of operation	BATCH
Name of the merged filed, AUTONAME to derive it from input name	AUTONAME

PEA

Command line parameters	Example
Executable name	pea
Feature called: PEA, **create a .pea archive**	PEA
Output name, full qualified name or AUTONAME to automatically name output starting from the name of first object in input list	"c:\testdir\out"
Volume size in bytes, 0 for not creating multiple volumes; volume size must be at least 10 byte more than volume control tag size, if a lower parameter is passed the volume size will be silently adjusted (it's only relevant for very small sizes)	200001
Compression scheme to use: PCOMPRESS0..3 (see "PCOMPRESS compression scheme" chapter)	PCOMPRESS2
Algorithm to be used for volume level check (see "Algorithms used in PEA format" chapter), NOALGO for omit this level of operation (all three levels of operation can be omitted in the same way)	SHA256
Algorithm to be used for object level check	RIPEMD160
Algorithm to be used for stream level check/encryption **HINT: see the full list of algorithms in the Algorithm chapter at page 9**	EAX
Mode of operation INTERACTIVE: the form is visible, user's input is requested if needed, this mode is recommended using PEA from command line since it is safer than passing secret parameters through command line (INTERACTIVE* modes can be used only for PEA and UnPEA since only those features may require keying, other modes can be used also for RFS and RFJ); BATCH: the form is visible, user's input not requested: if passphrase/keyfile are needed are got from <u>next two parameters</u> of the command line; HIDDEN: the form is not visible, user input not requested (as for BATCH); *_REPORT modifier can be applied to each mode: the program operates as described for the * mode used but an automated job report is saved at the end of the operation. INTERACTIVE_REPORT mode will ask where to save the report file, while batch and hidden *_REPORT modes will automatically save the report to output path without requiring further interaction.	INTERACTIVE
Object's names input method: FROMFILE: a single object name is passed as next command line parameter, the object is a plain text file listing full qualified names of objects to archive, one per line; number of parameters that can be passed in that way is practically limited by the structure internal to the program that stores input names (a dynamic array of string, that usually cannot grow over 2 GB). FROMCL: multiple object name are passed as next command line parameters, each one is the full qualified name of an input object; number of parameter passed will be limited by the host system's command line max buffer size (usually in the order of 32KB, system dependent). When input objects don't exist, this will be recorded in the job log. Received input names are "expanded" by the program in the sense that directories passed as input object are recursively listed and each single object contained will be internally listed by the program (applying the aforementioned limitation on the size of internal structure, a dynamic array of strings). Each single input object name can be up to 32KB in size, exceeding actual needs.	FROMFILE flist.txt

UNPEA

Command line parameters	Example
Executable name	pea
Feature called: **open a .pea archive**	UNPEA
Input archive (full qualified name), first volume if volume spanning was used	"c:\testdir\out.000001.pea"
Output name, full qualified name or AUTONAME to automatically name output from the archive's name	AUTONAME
Date and time parameter: RESETDATE use current date and time for extracted objects, ignoring date/time of archived object actually is the only option available	RESETDATE
Attributes parameter: SETATTR: restore archived object's attributes (apply them to output object) RESETATTR: create output objects with default attributes, ignoring archived attributes	SETTATTR
Extraction mode: EXTRACT2DIR is actually the only mode available, restore archived objects in a directory with name and path as in "Output name" parameter; restored objects have the shortest possible path to preserve directory structure; if duplicate objects are found, create progressive numbered copies (position in archive determine the progressive number)	EXTRACT2DIR
Mode of operation (as in PEA), except that if password is requested interactively (regardless the mode) if it is needed	BATCH_REPORT "This is the passphrase" NOKEYFILE

WIPE

Command line parameters	Example
Executable name	pea
Feature called: **wipe files and directories**	WIPE

Level of security: NONE, ZERO, ONE, HEADER, VERY_FAST, FAST, MEDIUM, SLOW, VERY_SLOW
Level none performs simply a non-secure direct file delete, making the file not recoverable from recycle bin-like utilities.
Level zero and one overwrite free space with all 0 and all 1 respectively, and are intended for improving compressibility of the filesystem (i.e. disk image backup, or virtual machine optimization) rather than for secure deletion.
Level header overwrites with random data first 64KB of data of the file, destroying the header and rendering unusable most file types, but it is not meant to sanitize the full content of file(s).
Very fast and fast overwrite file(s) with random data (AES256 CTR) and close file (forcing flush to disk), 1 and 2 times respectively.
Medium, slow, and very slow modes
  1) Overwrite file content with all 0 once, all 1 once, and finally with random data, respectively 1, 2 and 3 times
  2) Replace file content with random content of fake random size, (1B..4KB)*n, flush file each time
  3) Rename file or folder with random name, flush file, n times
	MEDIUM

Space separated list of files and folders to be deleted, full qualified names	"c:\testdir\testfile.txt"

SANITIZE

Command line parameters	Example
Executable name	pea
Feature called: **sanitize free space**	SANITIZE
Level of security: ZERO, ONE, VERY_FAST, FAST, MEDIUM, SLOW, VERY_SLOW Level zero and one overwrite free space with all 0 and all 1 respectively, and are intended for improving compressibility of the filesystem (i.e. disk image backup, or virtual machine optimization) rather than for secure deletion of free space. Very fast and fast overwrite free space with random data (AES256 CTR) 1 and 2 times respectively. Medium, slow, and very slow modes overwrite free space with all 0 once, all 1 once, and with random data, respectively 1, 2 and 3 times.	ZERO
Input: unit name	C:\

ENVSTR

Command line parameters	Example
Executable name	pea
Feature called: **display host system's environment variables' strings**	ENVSTR

COMPARE

Command line parameters	Example
Executable name	pea
Feature called: **compare files**	COMPARE
First input file: full qualified name	"c:\testdir\testfile_a.txt"
Second input file: full qualified name	"c:\testdir\testfile_b.txt"

CHECK

Command line parameters	Example
Executable name	pea
Feature called: **checksum and hash files, show 32 byte sample of file header and EOF**	CHECK
Output coding: write checksum/hash results as HEX: hexadecimal LSBHEX: hexadecimal, Least Significant Byte first BASE64: Base64 coding	HEX
List of algorithms to be used, enter one or more by name: ADLER32, CRC16, CRC24, CRC32, CRC64, ED2K, MD4, MD5, RIPEMD160, SHA1, SHA224, SHA256, SHA384, SHA512, SHA3_256, SHA3_512, WHIRLPOOL Otherwise specify one of following options: ALL use all supported algorithms LIST listing files and folders; folders displays statistics about content ANALYZE same as list, but displays first 32 byte at file header and last 32 bytes at EOF SHA256SAVE save SHA256 hash values and filenames to a sha256sum compatible text file, then close SHA256G search the SHA256 hash value on Google in the default web browser, then close SHA256V search the SHA256 hash value on Virustotal in the default web browser, then close	ALL
Stop parsing for checking algorithms/options and start parsing list of files/folders to be checked	ON
Space separated list of files and folders to be checked, full qualified names	"c:\testdir\testfile1.txt" "c:\testdir\testfile2.txt" "c:\testdir\testfile3.txt"

BENCH

Command line parameters	Example
Executable name	pea

Feature called: **perform synthetic CPU benchmark**
The benchmark runs a single core performances test over a single process, and a multi core performances tests over 16 processes launched in parallel.
Each process uses up to 6 threads:
  • one CPU-intensive thread performing multiple iterations of arithmetic and logic calculations over an array of 64K 64 bit integers and 64K double precision floating
  • up to 5 non intensive threads to update the GUI
Please note that alongside pure integer performances OS system scheduler efficiency, and possibly extra cores (to create and destroy processes, handle multiple GUIs' non intensive threads) impacts positively multi core benchmark results, while thermal throttling has a negative impact on performances.
Benchmark result unit, which is inversely proportional to completion time, is an arbitrary unit where 100/515 (100 as single core result, 515 as multi core result) is roughly equivalent to 2020 MacBook Air M1 chip, and does not represent directly any other measurement unit. Benchmark results are purely meant to be self-consistent and allow comparison between different machines.
Ratio value, for multi core benchmark, shows how faster was the calculation of the same math workload compared to single core.
	BENCH

LIST (superseded by CHECK HEX LIST ON)

Command line parameters	Example
Executable name	pea
Feature called: **list files and folders**	LIST
Mode of operation INFO: list files/filders and calculate information: total size, smaller/larger object, total objects, older/newer object LIST: list files/folders only (slightly faster)	INFO
input directory or file name: full qualified name	"c:\testdir\"

HEXPREVIEW

Command line parameters	Example
Executable name	pea
Feature called: **hexadecimal preview** shows offset, hexadecimal representation of bytes and possible utf8 translation of each string of 16 bytes per row; limited to 64 MB files max.	HEXPREVIEW
input file name (only files, dirs are not allowed): full qualified name	"c:\testfile1.txt"

TEXTPREVIEW

Command line parameters	Example
Executable name	pea
Feature called: **try to preview file as text** provides search, count lines/words/chars, text wrap, and change encoding functions limited to 256 MB files max.	TEXTPREVIEW
input file name (only files, dirs are not allowed): full qualified name	"c:\testfile1.txt"

CRC32SAVE, MD5SAVE, SHA1SAVE, SHA256SAVE, BLAKE2BSAVE

Command line parameters	Example
Executable name	pea
Feature called: **save crc or hash values in a file compatible with GNU Coreutils hashing utilities**	CRC32SAVE
Space separated list of files and folders to be checked, full qualified names	"c:\testdir\"

SHA256G, SHA256J, SHA256V

Command line parameters	Example
Executable name	pea
Feature called: **search SHA256 hash online for possible matches with known malware**	SHA256J
Single input file, full qualified name	"c:\testfile.exe"

MOTW, MOTWFULL

Command line parameters	Example
Executable name	pea
Feature called: **check if Mark of The Web is found in input files**	MOTW
Space separated list of files and folders to be checked, full qualified names	"c:\testdir\"

Command line switches (like AUTONAME, NOALGO etc…) are here conventionally written in uppercase but are not case sensitive since are internally converted in uppercase.

The max length allowed for command line and for each single parameter may vary from system to system, this may be a practical problem only for invoking PEA feature with a very long list of files passed as command line parameters (FROMCL switch), however it's possible using dir names as input objects allow to pass all dir's and subdir's content to PEA as a single parameter.

The problem may be avoided using FROMFILE switch and composing a plain text file listing one input object for each line, in that case a single parameter is needed for passing to PEA all input objects.

If PEA is used within a third part application as a library, the input objects will be passed directly as a dynamic array of ansistrings, being the size of the input array memory limited.

For any of the three approaches should be however noted that in the actual implementation the list of input objects (passed as parameters or from the file list or from input filename array) is internally parsed to recursively listing dir's content and discarding non accessible objects, and all object's names are saved into a dynamic array, whose size is memory limited; that may matter if a very huge number of input objects is internally listed to be archived, even if they are externally passed as a single filelist or under a single dir name.

As default policy PEA overwrites filetypes created by itself and don't overwrite any other kind of file, without asking for user interaction in those cases (to don't stop batch operations): in example when creating an archive the application will silently overwrite an archive with same name in same path, but on the other side it will not overwrite files extracted from an archive if the extraction operation is repeated multiple times, it will rather create a new output with progressive number appended to the name.

Pea executable automatically closes if no errors needs to be reported to the user, except if the operation is meant to generate human readable output (i.e. calculate checksum, display environment variables etc).

Exit codes

All functions follows the same exit code model for reporting successful termination or error:

> 1 abnormal termination, not an internal error, the process was externally forced to terminate (in MOTW command, Mark of The Web metadata is found)

0 success

-1 this exit code is set while the process is incomplete

-2 the process completed with non-blocking errors

      pea: some files cannot be archived (not found, not readable)

      unpea: errors detected in the archive

      wipe: some files cannot be deleted (locked, not found)

-3 the process encountered a blocking error

-4 process cancelled by user (cancel button is provided to allow stopping lengthy processes)

Report logs

In addition to information provided by the exit code it is possible to log in details the results of the activity, adding _REPORT modifier to "mode of operation" switch, for functions supporting that switch.

INTERACTIVE mode will become INTERACTIVE_REPORT mode, and will interactively ask to user where to save the log file.

BATCH and HIDDEN modes will become BATCH_REPORT and HIDDEN_REPORT modes, and will automatically save the log file in the same directory set for the output; no further user interaction is required so those modes are suitable for use in batch scripts for unattended operation.

Benchmarks

How does PEA implementation compares with existing state of art archivers?

Comparing different software that offers different features, saves different metadata, offers different interaction and reporting to the user, performs different runtime error checks etc… it's like comparing apples and oranges.

However, most users would like to know what to expect from an archiver software, at least for most evident and immediate factors: speed and compression ratio.

Settings

Hardware: notebook with Intel Core i7-8565U CPU, 4 physical cores with hyper-threading (8 logical cores), 8 GB RAM, system disk 512 GB PCIe NVMe SSD with NTFS filesystem

System: Windows 10 64 bit

Software: 64 bit versions of 7-Zip 19.00, PeaZip 8.0.0, WinRar 6.0

Data: benchmark input contains 42 files in 4 directories for total 303 MB (318.000.857 bytes), composed by

- Calgary, Canterbury and Silesia corpora
- enwik8, 100 MB English Wikipedia dump

Results

Test	Compression (sec)	Archive size (MB)	Extraction (sec)
WinRar, ZIP default	3,0	100,00	1,0
WinRar, RAR default	13,8	80,40	1,0
7-Zip, 7Z ZIP default	11,9	97,70	1,2
7-Zip, 7Z LZMA2 default	38,0	73,60	0,9
PeaZip, PEA fast	6,1	109,00	4,0
PeaZip, PEA default	10,5	101,00	3,9
PeaZip, PEA default - volume check (SHA256)	10,0	101,00	3,6
PeaZip, PEA default + encryption	10,8	101,00	4,3
PeaZip, PEA maximum	18,9	101,00	4,1

As expected, PEA, which uses Deflate based compression, provides compression ratio similar to ZIP format, closer to WinRar implementation, as 7-Zip uses a slower but more powerful optimized implementation of Deflate.

In terms of compression, both 7Z and RAR formats offers a consistent upgrade in performances, but at expense of being more computing intensive.

Compressed size, MB
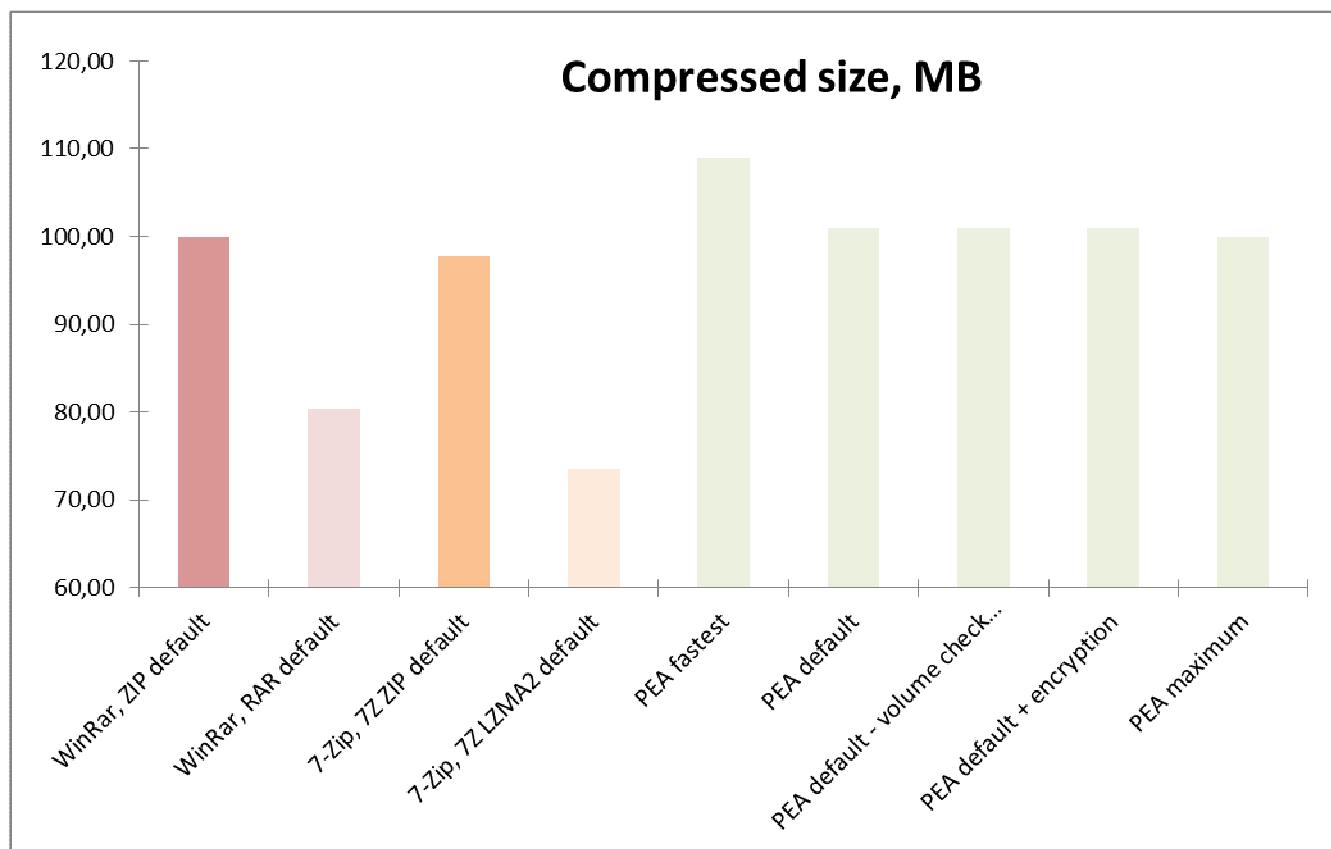
Fast PEA compression offers better performances (but not as fast as WinRar's ZIP implementation) at cost of worse compression ratio, maximum PEA compression does not offer noticeable advantages over standard compression in this test even if it nearly doubles the compression time.

7Z and RAR formats at default compression settings, even if efficiently using the multicore architecture, requires higher compression times than default Deflate-based PEA and ZIP at default compression settings, with 7Z being the slowest format.

It is interesting to notice the impact of additional algorithms in PEA format.
The 10,5 sec compression time can be reduced to 10 seconds (-0,5 sec) turning off SHA256 hash –based integrity check and providing only CRC32 checksum – comparable to what is offered by most of the competing formats.
Vice-versa, using authenticated encryption AES256 in EAX mode, increases the compression time to 10,8 sec (+0,3 sec).

Compression time, sec
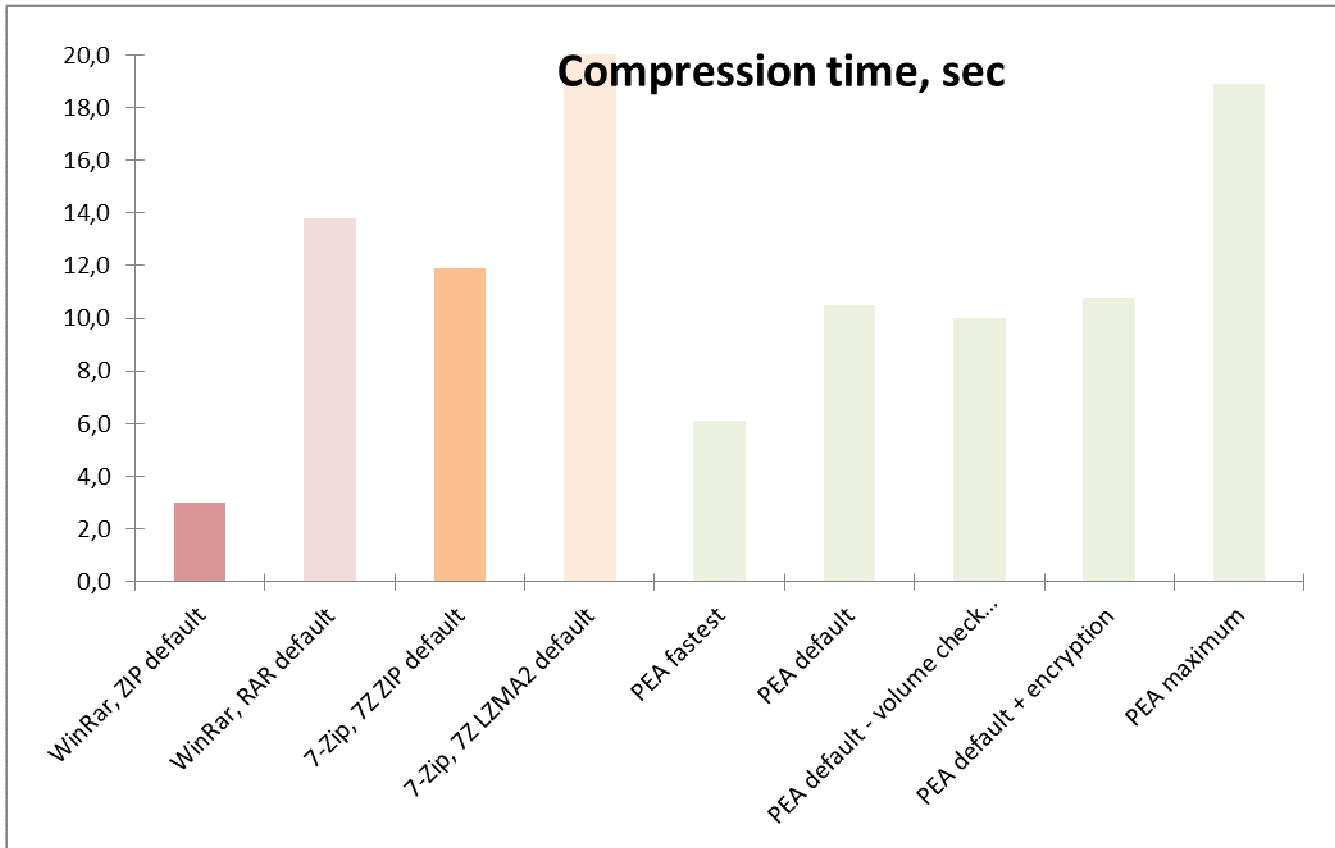
As for extraction times, PEA format is slower than competitors.

It is worth noticing that PEA compression level affects negligibly the de-compression speed, while employing additional algorithms have a noticeable impact: -0,3 sec turning off SHA256 integrity check, +0,4 sec using EAX mode AES256 authenticated encryption.

Extraction time, sec
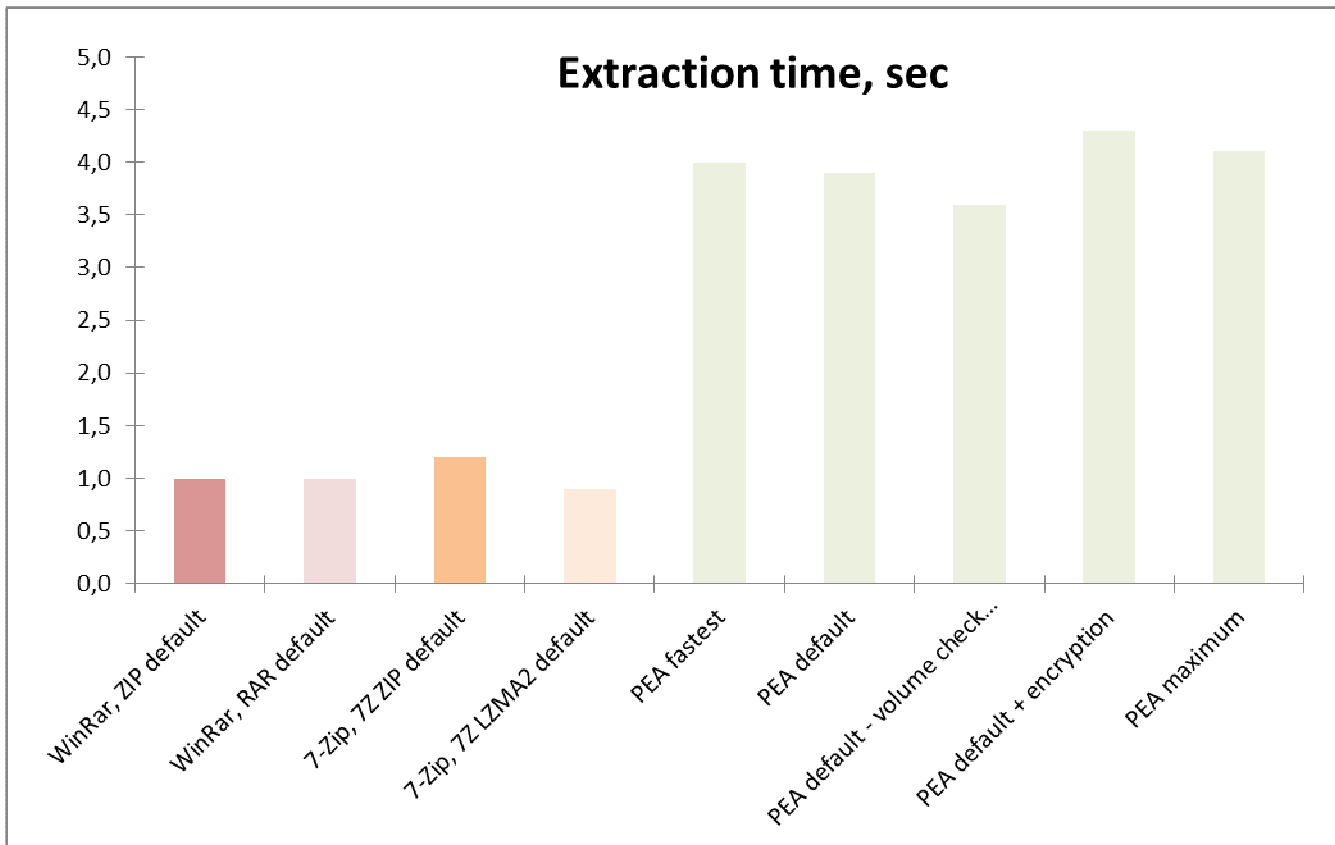
Conclusions

As expected PEA compression ratio is quite similar to other Deflate-based compression formats as ZIP (and Gzip).
Compression speed is comparable with slower 7-Zip's Deflate implementation, de-compression speed is slower than better optimized competitors.

Increasing compression settings to maximum provides little improvements for the entity of performance drawbacks, and it is not recommended for general usage – but may be beneficial on low entropy highly compressible data, which is out of the scope of this benchmark.

Being PEA compression Deflate-based and single threaded, resource usage is low both in terms of computing power (except, by purpose, for key initialization schedule if encryption is involved) and of memory usage, making PEA format suitable for being used in environments where multiple concurring instances are running, as servers.

Speed is slightly impacted by the strong multilevel integrity check and strong authenticated encryption offered by PEA format, but it remains in the range of Deflate-based class, the performance trade-off is reasonable for general use.

Implementation notes

(global):
- When a directory is provided as input object, the content of the dir and all subdirs is listed and added to the archive; arbitrary level of nesting is supported.
- When a hardlink is provided as input object, the pointed data is archived; when a symlink file (i.e., a Windows link) is provided as input object, only the link file, containing references to the actual data, is archived.
- Empty files and empty dirs are supported.
- There is no separate thread controlling the application and allowing aborting operation while running, the only way to stop a PEA instance is to kill the process.
- Unattended operations (BATCH and HIDDEN operation modes) are interrupted due to need of operator's feedback when one of the following conditions is encountered:
    - PEA output path is full;
    - output path seem to not have enough free space for requested operation;
    - parsing parameters, a critical error is encountered such that operation cannot continue;
    - elaborating the input data (UnPEA), a critical error is encountered such that operation cannot continue: the user will be prompted an error message; if the error type is unexpected a job log will be autosaved.
- Esteem of requested space (and number of volumes involved) for a given operation may be not accurate when: compression/decompression is involved (PEA, UnPEA); some objects get deleted or moved by other processes while PEA is archiving them; not all volumes are accessible in the same path (UnPEA, RFJ), this generally doesn't impact on other operations but may not allow to PEA to give accurate report of job progress.
- INTERACTIVE operation mode cannot be used when calling *_lib_procedure procedures (not calling from command line the pea application generated compiling project_pea as a standalone executable), al parameters must be passed from the main application;
- HIDDEN mode can be used even calling pea standalone executable from command line but the process will remain hidden and will not display a graphic tool for terminating it; that may be an undesirable behaviour under most circumstances.

PEA:
- If from command line, file list or procedure parameters are passed names with wrong cases as objects to save, it will work on case-insensitive systems (the objects with wrong cases in names will be found), but it will not work on case-sensitive systems (objects will not be found and will be listed as skipped in job log). It may lead to problems, on a case-sensitive system, using objects restored from an archive created giving wrong cases in names on a case-insensitive system;
- the application will silently switch to a volume size of at least 10 byte + volume tag size

RawFileSplit:
- .check file is saved to disk as a whole object, not taking in account the given volume size, that usually has no impact unless volumes space constrains are meaningful for the order of magnitude of .check file.

RawFileJoin:
- The application needs all the volumes to be accessible and in the same path, unlike UnPEA that can understand what volumes are needed and can ask the user for providing them if they are not found.

Known issues

UnPEA:
- Restore file attributes supported only on Win32.
- Doesn't restore file last modification date.

Notes for development

It's possible to compile pea_unit to obtain a standalone PEA executable, as in PEA executable implementation.

When launching PEA executable, form create parse paramstr(1) to determine what call_* procedure is being requested; call_* procedures allow interaction for entering passphrase/keyfile (if needed) and let time to the application to draw the appropriate form.

If no interaction is needed, or after pressing passphrase/keyfile confirmation button, "interacting" boolean variable is set to false, allowing a timer to call parse_action procedure that calls the procedure * (i.e. pea, unpea, rfs, frj) as requested in paramstr(1); after that, another boolean variable is set to true in order to prevent the timer to re-trigger the action again.

When the procedure * is called, it parses and error check the command line and, if no error is found, it calls the operative procedure *_procedure, passing appropriate parameters as translated from command line arguments.

When pea_procedure is called, the input list will be "expanded" by an internal procedure that recursively list dirs content and trim not found object from the input list; a detailed report is done listing found and not found objects (either missing while expanding input list or at any further point in pea_procedure invoking the object).

To use PEA features as library from another application, you can call *_lib_procedure procedures; refer to the code for parameter description, however those procedures aims to offer a syntax as close as possible to command line usage.

You could also directly call *_procedure procedures, that points directly to execution of requested features, however *_lib_procedure procedures aims to offer a more developer-friendly interface performing checks on passed parameters and autocompleting some of the parameters (most of the less intuitive ones) required by *_procedure procedures.

Mode of operation is declared as opmode in *_lib_procedure, then passed to *_procedure as pw_param; INTERACTIVE* modes of use cannot be used from *_lib_procedure, keying material (passphrase and, if used, keyfile) must be provided as parameters from the main application calling PEA or UnPEA features.

Section 3: Question & Answers

1) Why should I use PEA?

PEA format supports advanced security features:

- cryptographically strong hash functions for integrity checking, both for single objects in archive and for volumes.
- authenticated encryption, and triple cascaded encryption

PEA format specifications, and implementation, are open source

PEA implementation is script-friendly, totally usable from command line, and provides detailed task logging

2) Why should I NOT use PEA?

PEA is presently a minor format and possibly unknown to many of your intended recipients.

PEA does not provide novel compression algorithms capable to compete with most powerful ones (LZMA2, Zpaq), nor with fastest ones (Brotli, Zstandard)

PEA does not provide an error correction mechanism as RAR

PEA only offers support to basic functionalities over archive, add and extract (with built in verification), so there is no separate list/test command, or options to add, modify, and delete files in existing PEA archives.

3) How to decide if I should use or not use PEA?

PEA format is focused on security, basically create an archive, protect it with authenticated encryption, and thoroughly verify it was not tampered in any part on extraction.

The user requiring security and integrity, and possibly easy integration with command line scripts, fits the user case of PEA format.

Advanced usage as browsing archive content, and update existing archives, is not implemented, and consequently such needs rules out using PEA as archive format.

Also, PEA Deflate-based compression offers no advantage over other Deflate-based formats like ZIP and GZip, so it is not suited for users searching for faster than Deflate (as Brotli, Zpaq) or more powerful than Deflate compressors (as in RAR, 7Z, Zpaq formats).

4) Why is so difficult obtaining a good passphrase rating?

Passphrase's rating given by PEA is based on entropy evaluation according criteria and observations in [3] [4] [5]; no dictionary test is featured in the entropy calculation; entropy evaluation is restarted for each word.

It's quite established that is not easy introducing great amount of entropy typing easy to remember strings of characters, even using mixed cases, numbers and special characters.

Moreover, the entropy rating is not a strength rating, since it's not easy (if feasible at all) to automate an algorithm to understand if a passphrase is easily guessable or not i.e. can be guessed using social engineering, factor that can decrease by orders of magnitude the effort in guessing the passphrase.

As a general role of thumb, an entropy poor passphrase will generally be weak, but an entropy rich passphrase will not automatically be strong.

In order to fully exploit the keyspace of a 256 or even a 128 bit cipher is more suited a randomly generated keyfile containing sufficient entropy, sampled from more entropy-rich sources than written sentences; however that leads to entirely different kinds of problem, focused in managing the key file in a secure way to neither have it exposed nor lose it (that will make the encrypted data no longer accessible).

References

[1] Tadayoshi Kohno - Analysis of the WinZip Encryption Method - 11th ACM Conference on Computer and Communications Security, October 25-29, 2004.
[2] Mihir Bellare, Phillip Rogaway, David Wagner: The EAX Mode of Operation. FSE 2004: 389-407
[3] Matt Mahoney, Florida Institute of Technology: Refining the Estimated Entropy of English by Shannon Game Simulation.
[4] NIST SP 800-63 Electronic Authentication Guideline
[5] ECRYPT Yearly Report on Algorithms and Key Lengths (2005) revision 1.0, 26 January 2006
[6] http://home.netsurf.de/wolfgang.ehrhardt/index.html [no longer active]
[7] http://www.eskimo.com/~weidai/benchmarks.html
[8] http://rfc.net/rfc4086.html
[9] http://www.cs.fit.edu/~mmahoney/compression/text.html

Special thanks goes to:
- Wolfgang Ehrhardt who written and maintains a very good Delphi/Pascal crypto library, used in PeaZip project, you can find his works on [6]; please refer to Wolfgang Ehrhardt for all what concerns those units, (collected in "we" path in pea-peach source package);
- All people on sci.crypt.* and comp.compression, for the useful answers and information provided;
- Wikipedia, for the quality of the articles and links provided;
- FreePascal and Lazarus teams, for developing and maintaining such a good development environment and related documentation.