THE PUBLISHING HOUSE
OF THE ROMANIAN ACADEMY

PROCEEDINGS OF THE ROMANIAN ACADEMY, Series A,
Volume 14, Special Issue 2013, pp. 357–364

MAINTAINING THE CONFIDENTIALITY OF ARCHIVED DATA

Catalin LEORDEANU[*], Dragos CIOCAN[*], Valentin CRISTEA[*]

[*] Faculty of Automatic Control and Computers, University 'Politehnica' of Bucharest, Romania
Corresponding author: Catalin LEORDEANU, E-mail: catalin.leordeanu@cs.pub.ro

This paper proposes a novel data protection method, whether we are talking about data transmitted over the internet, or just archived and stored on a disk. The proposed method allows the owner of the data to store secret information inside and archive and also enables him a degree of deniability regarding its existence. This approach can be useful in situations when the owner of the data is constrained to reveal secret information. The proposed method enables the encryption of two different files which are then added to the same archive. Considering that one of the files contains secret information the user has the option of decrypting one of the files, while plausibly denying the existence of any other data inside the archive. This goal is achieved by adding random padding data which cannot be distinguished from encrypted data. As an additional protection method, after the encryption step, the data is split into multiple sections, which are scrambled according to a given key.

Key words: Confidentiality, Privacy, Cryptography, Plausible deniability.

1. INTRODUCTION

Protecting data against attacks is a very important subject which has been the focus of numerous research efforts. It is still a very active field, as researchers continue to improve data security. This goal can be seen as a combination of confidentiality, integrity, availability and authenticity.

The continual evolution of communication technologies, as well as the increase in the amount of data transmitted over the internet, require increased security. In this situation the main solution to ensure this degree of security involves cryptography techniques. This is a good solution to ensure the fact that an external attacker is not able to access the secure data. The choice of the encryption algorithm is also very important and we need to ensure that it is difficult to break. There are however situations when this is not sufficient, either from the fact that an attacker with sufficient time and computing power might break the encryption, or the owner might be constrained through other means to reveal the secret key. There is therefore a need for a mechanism which will enable the owner of the data to be able to plausibly deny the existence of any secret data in the archive. Without such a mechanism any encryption becomes useless if the holder of the secret key can be forced to reveal it.

One of the approaches used to achieve this goal is steganography. While encryption can be used to transform the data into something which cannot be understood, steganography can obscure the actual existence of the secret data. For example, one of the most common methods in steganography is called “LSB insertion” and it is used to hide secret messages into images. The priciple of this method is based on the modification of the least significant bit from the 24 bit reprezentation of each pixel, to form a secret message.Since only the least signinficant bit is modified the difference in the image will be minimal and the secret message will only be read by someone who knows of its existence.

The proposed solution combines steganography methods with encryption in order to hide a secret message inside and encrypted archive. Using this solution an attacker may decrypt the data inside the archive, without realizing that there is another hidden message.

This paper is organized as follows. Section 2 describes similar research efforts, with an emphasis on deniable encryption solutions. Section 3 describes the proposed solution and the structure of the resulting archive., Section 4 presents the algorithms which were used, as well as the operation which are performed in

order to ensure data confidentiality. Section 5 contains experimental results using the proposed solution and Section 6 concludes this paper and outlines directions for future research.

2. RELATED WORK

There are many research efforts which handle data confidentiality. One such solution is covered by the Deniable File Systems (DFS)[1]. This refers to file systems which can hide a small part of itself. It differs from classic Encrypted File Systems through the fact that the folders and files are not visible. [2] However, a disadvantage of the Deniable File Systems is the fact that it covers the entire partition or file system. It is not very useful if the user only wishes to hide a small secret message inside another file. StegFS [4] is also a file system which is able to hide the existence of different sets of data. It is actually an extenion of the Ext2fs file system for linux and the data can be encrypted as part of different security levels. An attacker would not be able to know if he has discovered the secret keys to all the security levels.

Another solution to ensure data confidentiality is TrueCrypt [3]. It falls into the category of Deniable File Systems, which we mentioned earlier. It is able to use different cryptographic algorithms, such as AES, Serpent, Twofish and others to encrypt in real-time an entire disk or a partition. It can also offer plausible deniability through the creation of a hidden disk partition, which is kept inside a visible partition. It has however a number of security vulnerabilities [3] and it is also difficult to configure.

A basic solution for deniable encryption is presented in [5]. It is however difficult to implement and use. Most other solutions have used the term “plausible deniability”, which brings it closer to steganography through the insertion of secret data inside encrypted files.

3. OVERVIEW OF THE CONFIDENTIALITY SOLUTION

The goal of the solution is to provide a way to maintain the confidentiality of archived data. The use of random encrypted data blocks is not new. In most algorithms the encrypted information is divided intro blocks of fixed size. For example, in the case of the AES algorithm, these blocks of data have 128 or 256 bits. Since the data that we need to encrypt is rarely of a size multiple of the block size, we are forced to use padding. The solution presented in this paper is based on the insertion of hidden data inside this padding when creating an encrypted archive.

The proposed solution is able to encrypt two different files, each with its own encryption key, and place them inside a single archive. One of the files can contain sensitive secret information, which the user is trying to hide. The other one may contain less sensitive information, which the user may reveal in order to obscure the existence of the first file. There is a compression phase when the two files are combined so as not to arrouse suspicions regarding the size of the encrypted output. Padding containing random data is also added to further obscure the real information. The difference in size of the final archive may therefore be blamed on the padding.

Let us consider the following scenario. A user following this solution may encrypt a file *f1* of 800KB, and another file *f2* of 200KB containing sensitive information. Considering a padding of approximately 20% the resulting encrypted file will have a size of 1200KB, without compression. If the user is forced to reveal the secret key to decrypt the data, he may only reveal the key which decrypts file *f1*, while blaming the difference in size on a larger padding than the one which was used. The existence of the second file, *f2*, therefore remains hidden.

The structure of the output file is shown in Fig.1. Upon analysis the archive will look like a single block of encrypted data. The choice of the AES cypher[6], using 128b or 256b will be sufficient to discourage brute force attacks.

The internal structure of the file can be divided into 3 useful parts: the header(1) containing identification data, offsets and encryption parameters, the padding(2) and the encrypted files(3). Another goal of the proposed solution is that the output should present itself as a single block of data, making it difficult for an attacker to distinguish its components from the padding.

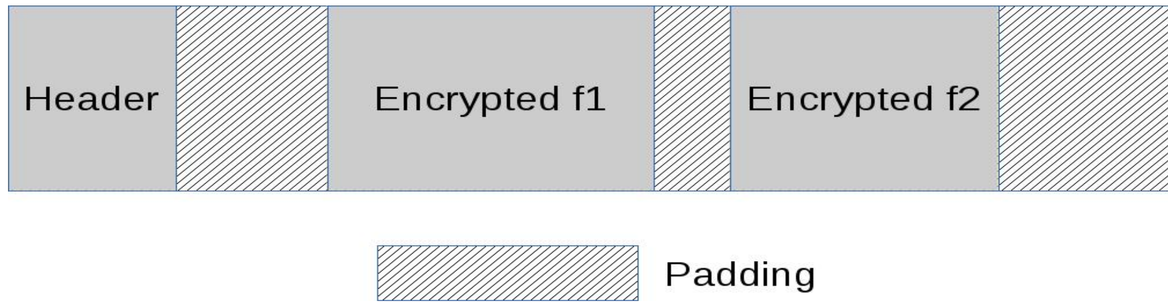

Fig. 1 – Structure of the output file.

3.1. The header

The header is a very important part of the archive, since it contains information describing the way that the rest of the data should be interpreted. The header also contains some padding, as to make it difficult for an attacker to identify its elements. The total length of the header is 256B.

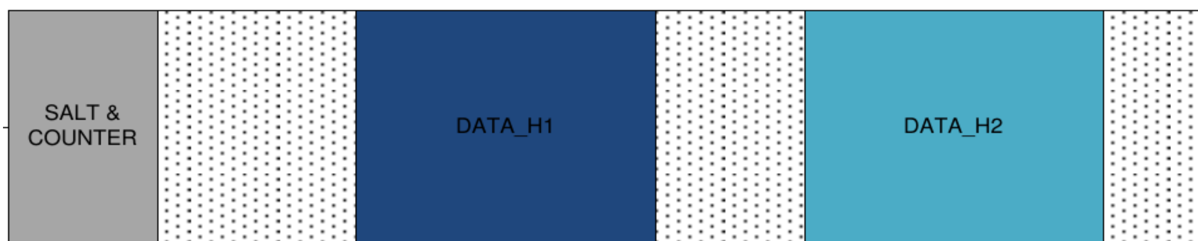

Fig. 2 – Structure of the header.

The header contains the following information:

– Salt & counter. The salt is a random generated number which will be added to the encryption key, to increase the difficulty of dictionary-based attacks. The counter, or Iteration counter also helps improve the security level of the encryption. These values are visible to an attacker and they are placed at the beginning of the header.

– Two data blocks Data_H1 and Data_H2. Each of these contains the offset and the size of the two encrypted files. Their positions inside the header are dependent on the encryption passwords by using and MD5 or SHA-1 hash functions. This blocks are placed at positions determined based on the passwords for each file.

Considering that we are dealing with integer values for the above information, the useful data in the header is 32B. The rest is padding, made up of random bytes. An attacker should have no way of distinguishing between the useful information inside the header and the padding.

3.2. The encrypted data

As seen in Fig.1, the archive should contain the two encrypted files. To increase the security of the solution the encrypted files will be split into blocks of fixed size and shuffled. Since a fixed shuffle scheme would have been trivial to undo, we used a key-based shuffle. The secret keys used for encryption are also used to guide the shuffle operations.

This approach also means that there is no need for any other secret keys, besides the two ones used for the encryption of the two files inside the archive.

4. STAGES FOR INFORMATION STORAGE AND RETRIEVAL

Fig. 3 shows the order of the operations which the user needs to perform during the data storage and retrieval phases. In the first row of the figure we have the stages needed to create the archive containing the

two encrypted files. On the bottom row we have the same stages in reverse order performed to retrieve the original data. Based on which of the two passwords the user decides to use, he will obtain the data for the first file or the second one, containing more sensitive data.

The actual compression step is simple but necessary in order to prevent the attacker from drawing any conclusions based on the size of the output file. Based on empirical evidence we decided to place the compression step at the beginning of the operation flow. This way the initial data has less entropy than the encrypted one and therefore the compression ratio is also slightly larger in most cases[7].

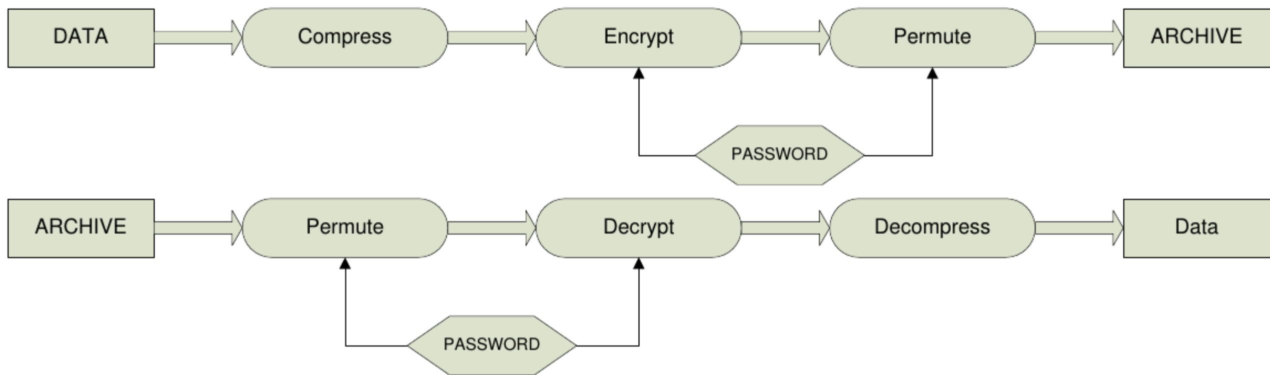

Fig. 3 – Operation flow for information storage and retrieval.

The actual compression step is simple but necessary in order to prevent the attacker from drawing any conclusions based on the size of the output file. Based on empirical evidence we decided to place the compression step at the beginning of the operation flow. This way the initial data has less entropy than the encrypted one and therefore the compression ratio is also slightly larger in most cases[7].

The encryption is also a very important step to ensure the security of the stored data. Since encrypted data is indistinguishable from sets of random bytes, this step also means that the attacker will have difficulty distinguishing between the encrypted files and the padding. However, this is also dependent on the ability of the random number generator to create strings with high entropy. Here the degree of security depends mostly on the algorithm used. We chose to use for our implementation the AES encryption algorithm, using a block size and key size of 128b, which is currently considered to be secure[6].

4.1. Key-based permutation

As an additional security measure, the entire data block is divided into blocks of the same size, which are then shuffled. The actual permutation is based on a key derived from the passwords used for the encryption and the number of blocks that the data has been divided into. The permutation we used is based on the algorithm presented in [8].

This permutation can be divided into three steps:

1) *Initialization of the permutation key.* Considering that the data we need to permute has been divided into N blocks we need to create a vector of the same size which will be the permutation key. Each element of the permutation key will be filled according to the ASCII value of a character from the password. Considering that the password which we use has a number of characters $n<N$ we are left with a number of $N-n$ positions in the key which we need to fill. This is done by adding two adjacent values and inserting the result on the first position. Afterwards, the values of the permutation key are brought into the interval [0,N). Considering that the character of the password has an ASCII value of X, the value of the element of the permuation key will be $X \bmod N$. This step ends when all the N positions of the permutation key have been filled.

2) *Elimination of duplicates.* If there are any duplicates inside the permutation key then the first value will be kept unchanged and the rest will be replaced by the value 0.

3) *Filling the blanks.* All the values of the permutation key must be brought into the interval *[1,N]*. All the values of 0 will be replaced with values which are not already present in the permutation key, starting from both ends of the key. This step ends when all the elements of the permutation key belong to the interval *[1,N]*.

An example for the creation of the permutation key, based on the password string "parola", for a permutation of 10 blocks, can be seen in Fig. 4.

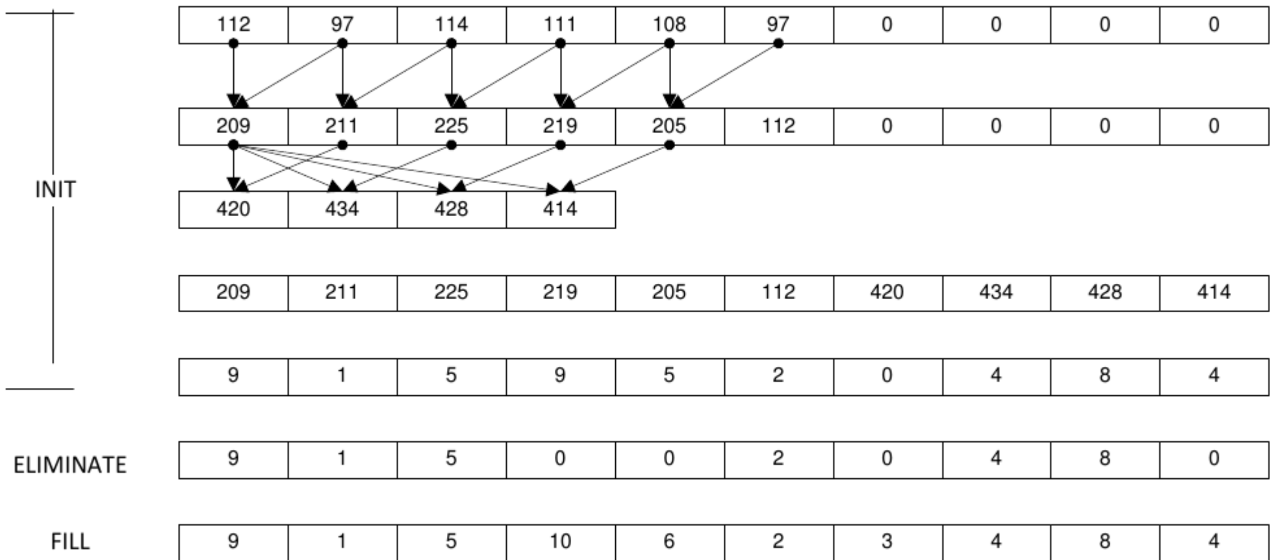

Fig. 4 – Example of the creation process of a permutation key.

The permutation is useful because it adds another level of protection for the data. An attacker would not be able to analyze the cyphertext until he has reversed the permutation. Since the permutation scheme is based on the passwords for the files it also does not require any other secret key from the user. After the end of the permutation stage, the result is the final form of the file containing the secret information. The process of retrieving the stored information is identical with the storage operations which we described in this section, but the necessary stages are executed in reverse order.

5. EXPERIMENTAL RESULTS

We tested the proposed solution using both text and binary files of different sizes. Our objectives were to validate the proposed solution and to evaluate it based on its performance and the size of the resulting archive. We were able to embed the two encrypted files of sizes between 100kB and 150MB into the archive, with padding between 10% and 30%, and also recover them successfully using the two passwords.

Regarding the performance of the proposed solution, we tested the required time for the each of the three main stages: the compression, encryption and permutation.

Fig. 5 and Fig. 6 show the dependency between the size of the files and the time needed for the data storage stages. Since there was a very large difference in the order of magnitude between the storage of very small files (1kB–100kB) and larger files, we chose to split this graph. Fig. 5 shows the performance for smaller files, upto 1MB, while Fig. 6 shows the same performance measurements, but for the experiments in which we used files between 1MB and 150MB.

We can see in these experiments the expected increase in processing time. However, considering that even for files of 150MB, we can see that each phase lasts less than 15 seconds, it is definitely slower than a simple compression or encryption. In total, for the largest experiment, the application would need 30 seconds to complete all the stages described in the previous sections. This should not be a problem for the user, since our solution is intended for the archival of sensitive data, not for data which is needed very often or data which is constantly updated.

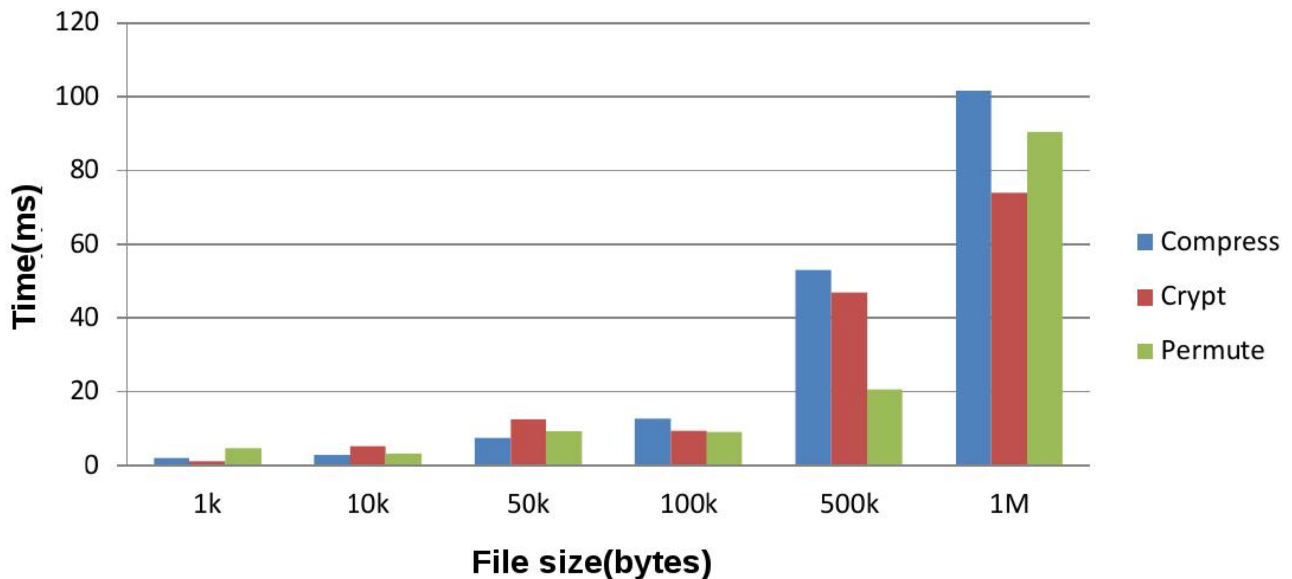

Fig. 5 – Performance of the storage operations for small files.

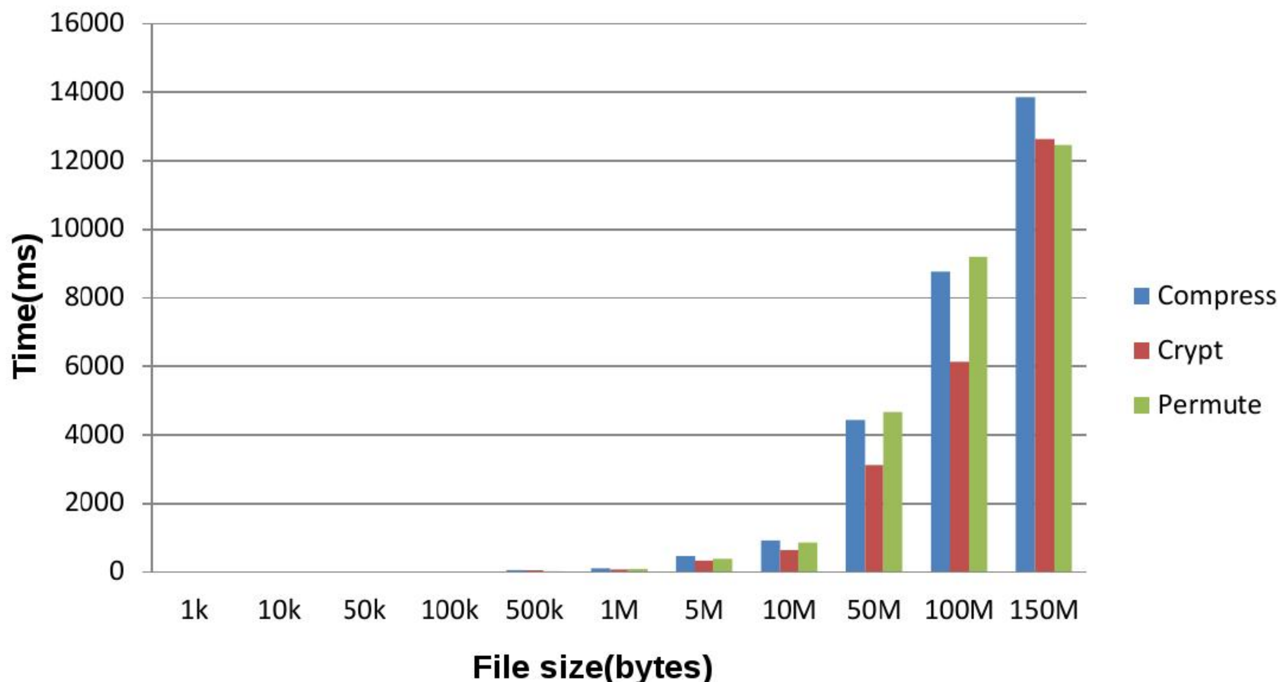

Fig. 6 – Performance of the storage operations for large files.

During our experiments we also found that the performance measurements were symmetrical. The measured performance was the same when we stored a set of files as when we retrieved them. This makes sense because they are basically the same operations, only executed in reverse order.

In Fig. 7 we represented the relative time needed for each stage of the proposed solution. It is evident that the permutation and the compression stages take the most time, while the encryption is highly variable, as opposed to the other stages.

Another important factor in our experiments was the size of the resulting archive. We needed to be sure that an attacker would not be able to discover the fact that there is any hidden data based on this information. We considered the following scenario. We inserted into the archive two files of different sizes, based on the algorithm described in the previous sections. In this case, the smaller file containing sensitive information is much easier to hide inside the archive, without raising any suspicions from an attacker. The difference in size could be attributed to the padding.

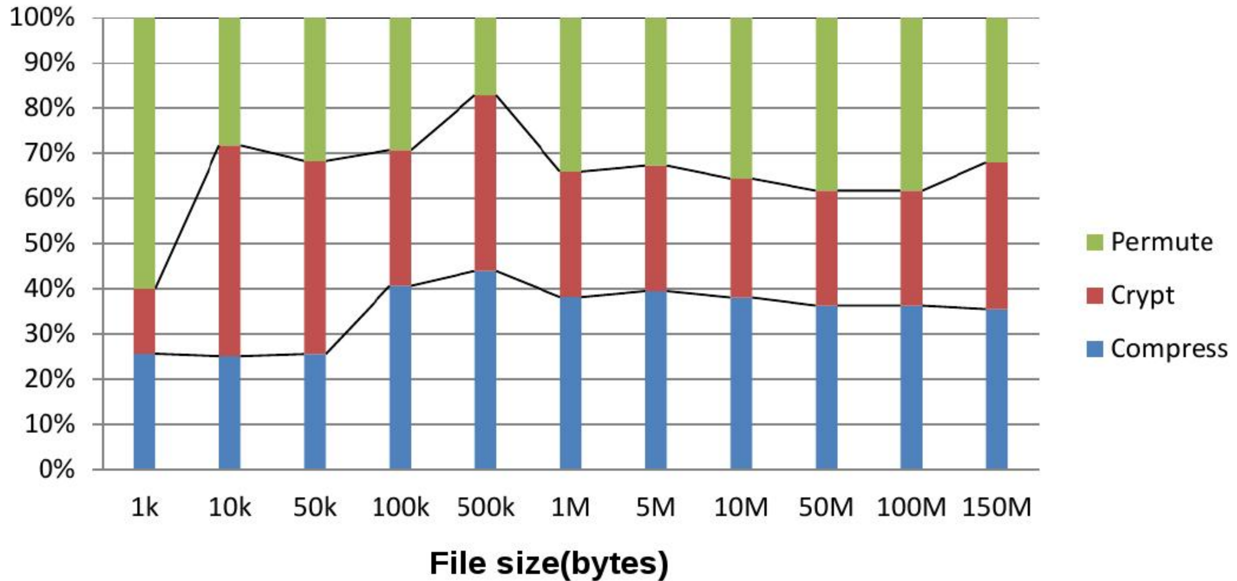

Fig. 7 – Relative performance for each stage.

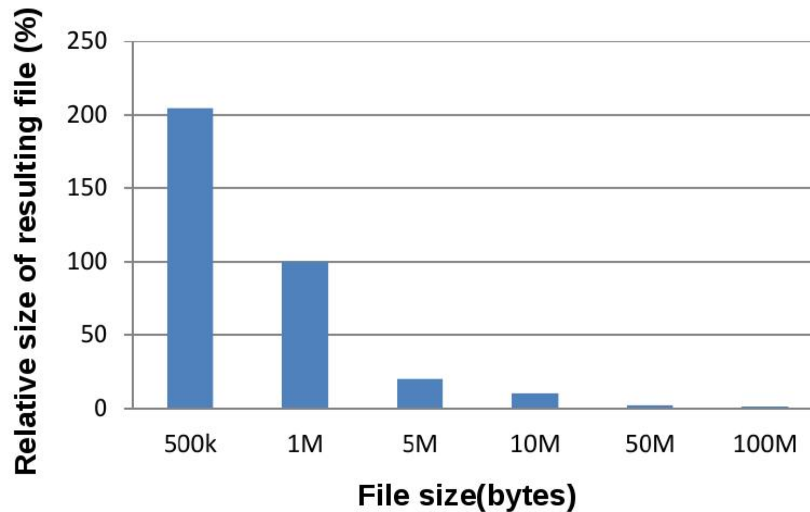

Fig. 8 – Relative increase in size for the storage of a 1MB file.

Fig. 8 shows the results of such a scenario. We considered a 1MB file which we needed to hide. The purpose of this set of experiments would be to find which would be the appropriate size of the second file inside the archive, so that the attacker would not be aware of the existence of the secret information. For this experiment, if we hide a 1MB file containing sensitive information, along with 5MB of data for the second file, we can see that the difference in size would be under 20%. We consider this to be the limit granting the user plausible deniability to the existence of the sensitive information. Any difference in the size of the archive below this limit could be considered to be a part of the padding.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a solution to increase data confidentiality by creating an archive which contains an encrypted file, as well as a second hidden file, encrypted with a different key. The user is therefore granted plausible deniability regarding the existence of the second file. The security of the solution is directly dependent on the security of the encryption algorithm. For this solution we used the AES cypher which is currently considered secure.

We also added an additional level of security through a key-based permutation of the resulting archive after the encryption step. The role of this step is to further increase the difficulty for an attacker to analyze the archive and find the hidden data. Therefore, if the user is forced to reveal the password he can just decrypt the first file and an attacker would be unaware of the existence of any other useful data inside the encrypted archive. Any difference in size due to the existence of the second encrypted file could be blamed on the padding.

As future work we intend to further improve this solution. One direction would be to increase the number of tests and to see if we can increase the security of the hidden data, either by improvements for the storage and retrieval algorithms, or through the use of different methods for the random number generator, permutation or cryptographic algorithms. Another direction for future work could be the investigation of the actual storage methods for such confidential data. Since Cloud storage services are becoming widespread, there is a need to improve the security of such data and to make sure that the users can trust that their data is not accessed by any unauthorized entitites, even if they do not have direct control over the storage environment. Ensuring data confidentiality is a very challenging subject, which has the potential to improve many security systems, as well as the usability of existing storage solutions.

REFERENCES

1. Anderson, Ross, Roger Needham, and Adi Shamir, *The steganographic file system*, Information Hiding. Springer Berlin Heidelberg, 1998.
2. A. Czeskis, D. Hilaire, K. Koscher, S. Gribble, T. Kohno, B. Schneier, *Defeating Encrypted and Deniable File Systems*, HOTSEC'08, 2008.
3. Czeskis, Alexei, *et al.*, *Defeating encrypted and deniable file systems: TrueCrypt v5. 1a and the case of the tattling OS and applications*, 3rd USENIX HotSec (2008).
4. McDonald, Andrew D., and Markus G. Kuhn, Stegfs, *A steganographic file system for linux*, Information Hiding, Springer Berlin Heidelberg, 2000.
5. Canetti, Rein, *et al.*, *Deniable encryption*, Advances in Cryptology—CRYPTO'97, Springer Berlin Heidelberg, 1997. 90–104.
6. Akkar, Mehdi-Laurent, and Christophe Giraud, *An implementation of DES and AES, secure against some attacks*, Cryptographic Hardware and Embedded Systems–CHES 2001. Springer Berlin Heidelberg, 2001.
7. Balakrishnan, Kedarnath J., and Nur A. Touba, *Relating entropy theory to test data compression*, Proceedings IEEE European Test Symposium. 2004.
8. S.M. Hussain, N.M. Ajlouni, *Key Based Random Permutation*, Journal of Computer Science 2 (5): 419–421, 2006

Received July 15, 2013