File Encryption and Hiding Application Based on Advanced Encryption Standard (AES) and Append Insertion Steganography Method

Gotfried C. Prasetyadi, Achmad Benny Mutiara, Rina Refianti
Faculty of Computer Science and Information Technology
Gunadarma University
Depok 16424, Indonesia
e-mail: amutiara@staff.gunadarma.ac.id

Abstract—**Steganography is a method of hiding secret message in other innocuous looking object so that its presence is not revealed. In this paper, a message, which is a computer file of any type, is hidden in a selected cover or carrier file, which is a computer file of certain types. The steganography method used in this paper is called append insertion steganography method. This method of steganography is chosen as an attempt to remove the limitation of message format, which appears in many popular steganography methods. To scramble the meaning of the hidden message, AES-256 (Rijndael algorithm) is used to encrypt the message with a secret passphrase. A special block of bytes is used to identify and verify the original message so it can be recovered while retaining its integrity. The C# programming language and .NET Framework are chosen to implement the algorithm into a Windows application. In this paper, one cover file contains exactly one message file. In the testing, five random files are used as secret message. Their integrity is calculated using SHA-256 before hiding and after recovery. In the testing, they all retain their integrity, proven by exact hash values. Thus, the application as the implementation of proposed algorithm is proven feasible but only for personal use as some improvements still has to be implemented.**

Keywords— file hiding, append insertion steganography, symmetric key encryption, personal security

I. INTRODUCTION

Steganography is the art of concealed communication, which keeps the existence of a message secret. It is an alternative tool for privacy and security, which may suppress and prevent unnecessary attention of third parties, as encrypted messages are obvious even though they are in nonsense form [1]. Thus, steganography can be applied to avoid unwanted attention or in places where the usage of encryption is illegal. Encryption and steganography can be used together to ensure data security. Any possible eavesdropper should not realize any attempt of secret communication at all, and when the presence of the message is compromised, it is still not in meaningful form.

Appending data to the end of file is one of the most flexible forms of digital steganography, as many file types allow data to be appended into them, without causing error, and it removes limitations such as size and type of data that could be hidden. Many types of file have end-of-file (eof)

marker inside it, and most file readers will simply ignore any data after the eof [2].

A. Related Works

Many articles are reviewed not only to study the algorithm, but also to find similarity in them that can be considered as limitation, which solution can be proposed.

In [3], grayscale secret images with bit depth of 8-bits are hidden in cover images with bit depth of 24-bits. Cover images have size of 800x800 pixels, while secret images have 100x100 pixels. The resulting stego images have 99% conformity with the original images and it can withstand both intentional and unintentional attacks to some extent.

In [4-5], images are processed using binary mode. Transform domain method such as DWT and DCT are used. In addition, the size of the images is often too small.

[6] uses pixel differencing with n-bit Least Significant Bit (LSB) substitution to provide higher embedding capacity without sacrificing the imperceptibility of the host data. Some images used in the testing have up to 359 kilobytes hiding capacity. In [7], a steganography technique using LSB substitution and PVD method is presented as an adaptive scheme in the spatial domain. The method partitions the grayscale image into several non-overlapping blocks with three consecutive pixels. This can also increase hiding capacity. Both [6] and [7] have not yet implemented encryption.

B. Typical Limitation of Various Steganography Methods

As mentioned previously, many steganography methods limit the type and size of message that can be hidden inside cover/carrier files. At best, the hiding capacity of cover files can be optimized using methods that are relatively new and they were tested against various attacks with certains parameters.

This study aims to design and implement an algorithm based on append insertion steganography method that is flexible for the user to choose what type and size of secret file he/she wants to hide, instead of focusing to limit the message in some way to meet the requirements of a steganography method.

II. Research Method

A. Limitation of Cover File Types

While the message file can be practically any file, the carrier file must have certain characteristic so it can be opened after being embedded with encrypted data. To avoid losing the embedded data, the best types of carrier file are the files intended for sharing and viewing only. The file format must also have some kind of stop marker, found either at the end of file or at other structure, which is used as a point for their default handler application to stop reading the stream of data. This way, the resulted stego file can still be opened normally. Some file formats which have that characteristic, are:

- Portable Document Format (pdf),
- Bitmap image (bmp),
- Joint Photographic Expert Group (jpg/jpeg), and
- Portable network graphics (png).

B. Proposed Algorithm for Identifying Secret Message

A special block of bytes, called Identification (ID) Block, is needed for the algorithm to find out the following information of a secret message:

1. Hash value. This is used as secondary measure for authentication and integrity. The original message hash value is kept and later compared when the message is extracted. Forty digits long SHA1 hexadecimal value is chosen.

2. Type (extension) of file. Even though the file has no extension at all, it will pose no problem because empty string value ("") is supported.

3. Key index. A 0-based index value is needed to tell the application at which point the message sequence start in the stego file, so it can be extracted completely. This value is equal to the length of cover file.

Vertical line/pipe (|) character is used to separate those three values. The key index is generated last and always padded so the total length of ID block is 64 bytes. Therefore, the structure of this vector is:

[40 bytes hash] + "|" + [x bytes extension] + "|" + [y bytes key index],

where

$$40 + x + y + 2 = 64.$$

This ID block is encrypted and then appended to the end of encrypted message.

C. AES-256 Encryption

Advanced Encryption Standard (AES) with Rijndael algorithm, which replaced the aging DES, is used to enhance the security and confidentality of data [8]. In this study, 256-bits key length with block size of 128 bits is used.

The core of Rijndael algorithm is the round function, which consists of four steps; each has its own particular function [9]:

1. Bytesub (BSB),
2. Shift Row (SR),
3. Mix Column (MC), and
4. Round Key Addition (RKA)

1) Cipher Block Chaining

CBC is a block cipher mode of operation to define how block ciphers (such as in DES and AES) may be applied correctly depending on the specification [9]. CBC eliminates the near obvious data pattern weakness found in the older Electronic Codebook (ECB) mode.

In CBC mode, each block of plaintext (128 bit for AES) is XORed with the previous ciphertext block before being encrypted using chosen algorithm, hence the name 'chaining'. For the first plaintext, an initialization vector (IV), which best randomly and securely generated, is used to XOR the first plaintext.

For the decryption, previous ciphertext block is used to XOR the decrypted ciphertext block to generate the plaintext. The same IV used in the encryption is used to XOR the first decrypted ciphertext. Figure 1 shows the diagram of CBC operations.
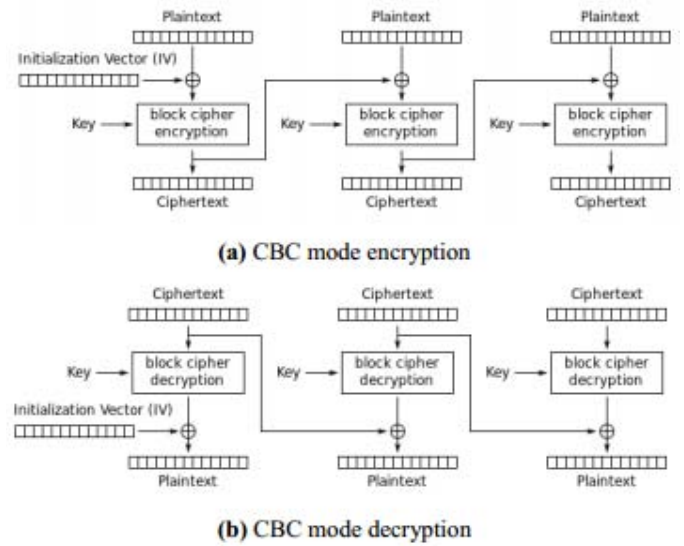

(a) CBC mode encryption

(b) CBC mode decryption

Fig. 1. CBC mode of operation

2) PKCS7 Byte Padding

During encryption process, the message chosen by user may come in various sizes, and the plaintext block cannot be empty and must be filled completely with the help of padding. Thus, PKCS7 padding is used in this study for its simplicity. In PKCS7, the value of each added byte is the number of bytes that are added. For example, if 3 bytes of message is put into the 8-bytes block, additional 5 bytes are needed to complete the block. Using PKCS7, the value of those bytes is 05. Even if the message size in bytes divided by block size results in

integer value, padding is still used for consistency. PKCS7 is thoroughly described in [10].

D. Proposed Algorithm for File Hiding and Recovery

1) File Hiding

The steganography method used to hide the confidential message file is appending insertion method. The primary reason is to remove the file type and size limitation found in many other methods. The activity diagram of the proposed file hiding scheme is illustrated in Figure 2. The algorithm is as follows:

1. Choose a carrier file and a message file.
2. Read all contents of those files as binary into array of byte.
3. Generate 64 bytes identification (ID) block.
4. Encrypt the of the message byte array.
5. Append the ID block to the encrypted message byte array.
6. Append the message bytes with ID array to the cover byte array.
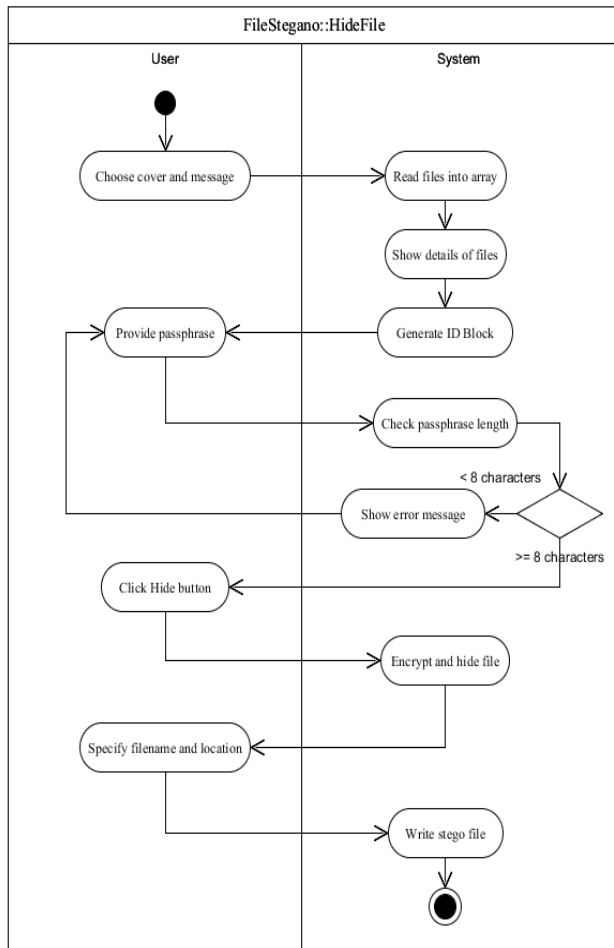7. Write the resulting stego byte array to a stego file.

Fig. 2. File Hiding activity diagram

The AES-256 with 256-bit key and 128-bit block size is chosen as encryption standard. Cipher Block Chaining (CBC) mode of operation is used to handle the blocks. The encryption algorithm is as follows:

1. Randomly generate 128-bit (16 bytes) salt and initialization vector (IV).
2. Use the user given passphrase and salt to generate the real 256-bit secret key.
3. Put the message bytes into blocks of 128-bit with the help PKCS7 padding.
4. For $i = 0$; $i <$ number of blocks; i++:
   - If $i = 0$, XOR the message block with generated IV, encrypt the resulting block using the secret key to generate ciphertext block.
   - Else, XOR the message block with previous ciphertext, encrypt the resulting block using the secret key to generate ciphertext block.
5. Prepend the salt and the IV to the generated ciphertext.

The functions to generate salt, IV, and secret key must be secure. The resulting stego file will have this structure:

[x bytes cover] + [[16 bytes salt] + [16 bytes IV] + [y bytes encrypted] + [64 bytes ID]]

2) File Recovery

To recover the hidden message file, the ID block has to be retrieved first before encrypted message bytes can be extracted and decrypted with secret passphrase. The algorithm is as follows:

1. Choose a stego file.
2. Read all contents as binary into a byte array.
3. Get the 64 bytes ID block at the end of stego byte array.
4. Extract and read the encrypted message bytes into a byte array.
5. Decrypt the encrypted byte array
6. Write the resulting message byte array to a file with certain type.

The activity diagram of the proposed file recovery scheme is shown in Figure 3.

The decryption algorithm is as follows:

1. Get the 16 bytes salt and IV.
2. Generate the original secret key using correct passphrase and the salt.
3. Put the ciphertext bytes into blocks of 128-bit.
4. For $i = 0$; $i <$ number of blocks; i++:
   - If $i = 0$, decrypt the ciphertext block using the key, and XOR it with IV to generate the message block.
   - Else, decrypt the ciphertext block using the key, and XOR it with previous ciphertext block to generate the message block.
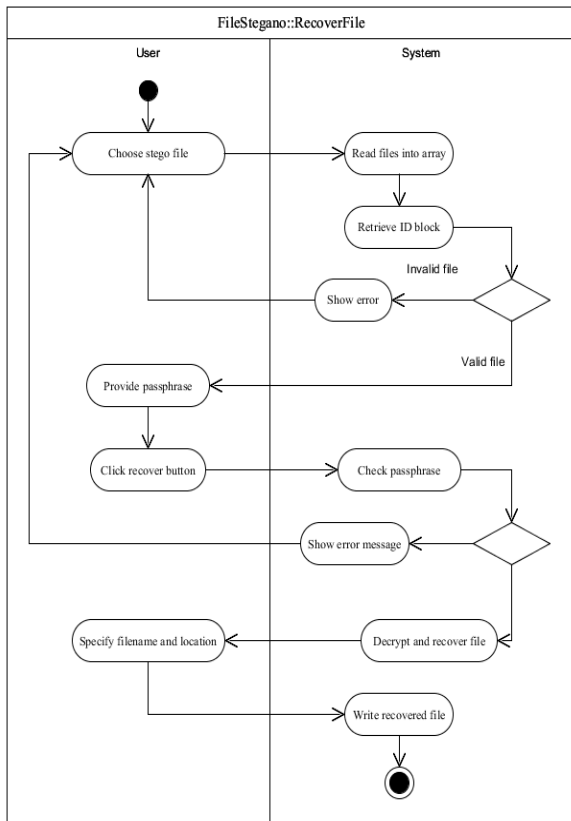
Fig. 3. File Recovery activity diagram

III. RESULT AND DISCUSSION

The application was made using C# programming language and .NET Framework optimized for Windows operating system. The size of final assembly is approximately 58.5 KB.
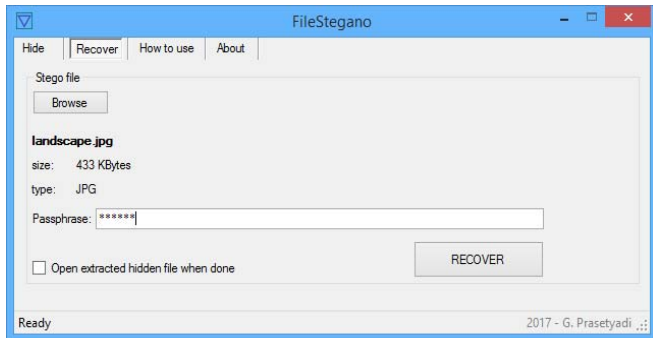

Fig. 4. User interface of application

The integrity testing examined whether the message file, which is the file to be hidden in the carrier/cover file, retained its integrity before being encrypted and embedded into the cover file, and after being successfully extracted and decrypted. SHA-256 hash function was used to produce hash value (also called signature or checksum). The same message file would always produce the same hash value.

In this testing, five randomly generated files, each with different size, got their hash value computed. Those files was

encrypted and hidden in a carrier file and then retrieved back, using proposed algorithm explained previously. Table I shows the details of those files. A random 2 MB portable document format (PDF) file was used as cover. Note that the characteristics of cover files would not matter most of the times, as long as they comply with the requirements explained previously. The hash value of the retrieved files was to be computed again, and the value should be the same. The result is shown in Table II.

TABLE I. DETAILS OF RANDOMLY GENERATED FILES

No.	Filename	Size (MB)	Size (bytes)	Hash value
1	dummy09MB	9.53	10000000	E0BD9F3AB7B7868B...
2	dummy19MB	19.0	20000000	4F6D706F771F58A3...
3	dummy28MB	28.6	30000000	D129C56BB17001E1...
4	dummy38MB	38.1	40000000	AAFE10A8DD14DD04...
5	dummy47MB	47.6	50000000	D7EA6522CDA07B68...

TABLE II. DATA INTEGRITY TESTING RESULT

No.	Filename	Size (MB)	Size (bytes)	Hash value match?	Conclusion
1	dummy09MB	9.53	10000000	Match	VALID
2	dummy19MB	19.0	20000000	Match	VALID
3	dummy28MB	28.6	30000000	Match	VALID
4	dummy38MB	38.1	40000000	Match	VALID
5	dummy47MB	47.6	50000000	Match	VALID

To measure how the usage of this application affects system performance, additional test was run. Dummy files that had been used in data integrity test were used again, this time to analyze memory usage and time needed to complete two main operations: data hiding and encryption.

The test is run on the local machine with following specifications: CPU: 4 Cores, up to 3.6 GHz, RAM: 8 (2x4) GB, 1600 Mhz, Hard drive: 466 GB capacity, Read 74.45 MB/s, Write 64.18 MB/s, Operating system: Windows 8.1 Pro, 64-bit

Table III shows the result,

TABLE III. PERFORMANCE TESTING RESULTS

Filename	Size (bytes)	Encryption time (ms)	Hiding time (ms)	Peak memory (MB)
dummy09MB	10000000	405	168	144.8
dummy19MB	20000000	808	472	255.1
dummy28MB	30000000	1181	1433	267.5
dummy38MB	40000000	1612	2710	624
dummy47MB	50000000	2005	1272	690.4

During the testing, CPU, memory, and hard drive were on idle and used to process only necessary programs and services,

including the operating system, the IDE and its debugger, and the application itself.

Based on the result, obviously, the bigger is the message file, the longer is the encryption and hiding time. Exception was found at the last test, which the hiding time for a file with 50000000 bytes in size only took 1272 milliseconds. This could be caused by the .NET Framework optimization when operating with large data.

IV. CONCLUSION

The proposed file hiding and file recovery algorithm involving identification (ID) block was proven usable for hiding any kind of data (file) using append insertion method in the selected cover file types. In addition, this method allows the usage of current standard AES to scramble the message/hidden files. The application as the implementation of proposed concept could be used to hide and later recover a message file up to 47 MB in roughly 2005 milliseconds. This study is an initial attempt to remove the limitation of message type and size in steganography method, thus giving users freedom to choose any file format. Some improvements have to be done, such as to increase its robustness to detection and various attacks by making the cover and the message seem inseparable through implementation of special file formatting and handling.

REFERENCES

[1] I.J. Cox, M.L. Miller, J.A. Bloom, J. Fridrich, T. Kalker, Digital Watermarking and Steganography, Burlington: Morgan Kaufmann Publishers, 2008.

[2] M. Raggo, C.Hosmer, Data Hiding, Massachusetts: Elsevier. 2013, pp.43-46.

[3] M.R. Pour Arian, A. Hanani, "Blind Steganography in Color Images by Double Wavelet Transform and Improved Arnold Transform, Indonesian Journal of Electrical Engineering and Computer Science, 2016, Vol. 3 (3), pp.586-600.

[4] K.R. Kakkirala, et al., "Block based robust blind image watermarking using discrete wavelet transform", Signal Processing & its Applications, IEEE 10th International Colloquium on Kuala Lumpur, 2014, pp.58-61.

[5] T.M. Thanh, et al., "A proposal of novel q-DWT for blind and robust image watermarking," IEEE 25th Annual International Symposium on Personal, Indoor, and Mobile Radio Communication (PIMRC). Washington DC, 2014, pp.2061-2065.

[6] M.S. Arya, M. Rani, C.S. Bedi, "Improved Capacity Image Steganography Algorithm using 16-Pixel Differencing with n-bit LSB Substitution for RGB Images," International Journal of Electrical and Computer Engineering (IJECE), 2016, Vol.6 (6), pp.2735-2741.

[7] M.B. Jahromi, K. Faez, "An Adaptive Steganography Scheme Based on Visual Quality and Embedding Capacity Improvement," International Journal of Electrical and Computer Engineering (IJECE), 2016, Vol. 4 (4), pp.573-584.

[8] H. Delfs, H. Knebl, Introduction to Cryptography, New York: Springer, 2007.

[9] R.A. Mollin, An Introduction to Cryptography, Boca Raton: Taylor & Francis Group, LLC, 2007.

[10] R. Housley, RFC 5652 - Cryptographic Message Syntax (CMS). September 2009. [Online]. Available: https://tools.ietf.org/html/rfc5652#section-6.3. [Accessed 3 May 2017].