L-006

複数ユーザ間の安全なエンドツーエンドデータ共有
End-to-End Secure Multi-Party Data Sharing

並木 優祐[†]　　中村 章人[†]
Yusuke Namiki　Akihito Nakamura

1. Introduction

The advance of IT and the Internet makes it easy to transmit and share data with friends and colleagues using intermediate systems, such as USB flash drive, email, and file hosting service. Because they involve security risks, people often secure data with access control mechanisms to counteract threats. There are two approaches: file-based and server-based. In the former case, a file is encrypted or password-protected. It requires a secure way to deliver secret information, including encryption key or password, to the parties. In the latter case, users are highly dependent upon the service and administration, and can trust it completely. Public-key cryptography solves these problems. It is an asymmetric cryptography system assuring the confidentiality, authenticity, and non-repudiability. However, it is limited to one-to-one encryption between two users and unsuitable for sharing the same secret information among multiple people.

This paper presents a system, called ShEnc, for end-to-end secure multi-party data sharing. End-to-end encryption provides secure transmission of message from one end to the other while the intermediate systems may not be especially trustworthy. The system depends neither on a prior secret sharing nor a dedicated server. Instead, we utilize the public key encryption technique; only the public keys of the parties are disseminated beforehand, and robust confidentiality of shared data and authenticity of the sender are possible.

Eu-Jin Goh, et al., proposed a secure file system on remote untrusted storage [1]. The approach is similar in that the file system uses public-key cryptography. The target device of the system is only remote storage like network-attached storage, especially NFS. Our system is applicable any types of intermediate, including remote storage, portable storage, email attachment, and file hosting service.

2. ShEnc system

Here, we present the design and implementation of the proposed system.

2.1 Assumption

The system is responsible for the secure transmission and storage of files. That is, it ensures the confidentiality and integrity of the names and contents of the files. We suppose that the intermediate systems may not secure; for example, the threat actors may gain unauthorized access to the file hosting server or conduct man-in-the-middle attacks, the owner may lose a USB flash drive or have it stolen, and so on.

† 会津大学　The University of Aizu

We also suppose that the private key is kept secure by its owner. In addition, users can obtain public keys correctly so that we don't care about key exchange problem. Our system uses the SHA-2 hash function [2], AES cryptography [3], and RSA cryptography [4]. We suppose they are the secure algorithms.

2.2 File format

Figure 1 shows the file format to pack the encrypted message, i.e. a file of data, and the related information. The file of this format is referred to as ShEnc file. The file is composed of two parts: header and data.

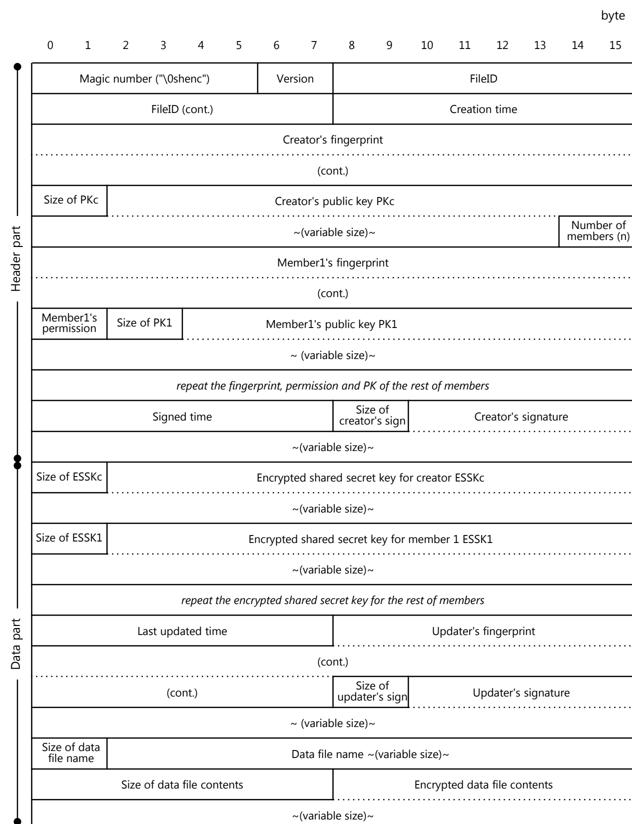

Figure 1: ShEnc file format

The header part is composed of the following main elements. These values are generated by the creator, i.e. the initial sender, of this ShEnc file. This part is not encrypted and available for inspection by anyone.

- Magic number: file format indicator.
- Version: version of the file format (currently 1).

Copyright © 2020 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

- FileID: ID of ShEnc files to track change history of the data file.
- Fingerprint: public key fingerprint to identify a public key; also used to find user's encrypted shared secret key.
- Permission: user’s access mode: read or write the data file.
- Signature and time: the base for verification of ShEnc file.

The main elements of the data part are as follows.

- Encrypted shared secret key (ESSK): the shared secret key to decrypt the data file. It is encrypted with each member’s public key.
- Updater fingerprint and signature: the base of verification of ShEnc file.
- Data file name and contents: the file to be shared securely; encrypted by the shared secret key.

The data file can be modified not only by creator but also by sharing members who have write permission. After the modification, the ShEnc file is signed by the member who modified the data.

2.3 Encryption and description algorithms

2.3.1 Encryption

Figure 2 shows the encryption procedure. The inputs are the member list and the data file. We assume that the creator has completed an acquisition of the members' public keys in advance.

(1) The creator specifies the permissions of members and generates the header information.
(2) The header information is encoded into a sequence of bytes. It is hashed with cryptographic hash function SHA-2 and signed with the creator's private key SKc.
(3) The creator generates a shared secret key SSK.
(4) For each sharing member m, ESSKm is generated by encrypting SSK with the member's public key PKm and enclosed in the data part. This step is also applied with the creator and PKc.
(5) The contents of the data file are encrypted with SSK.
(6) The header information and the encrypted data file are put all together.
(7) Finally, the whole information is encoded into a sequence of bytes. The hash value is produced and the ShEnc file is signed with the creator’s private key SKc.

2.3.2 Decryption

Figure 3 shows the decryption procedure at the recipient side. The input is a ShEnc file only.

(1) First, the recipient verifies the creator and its signature with the creator’s public key PKc.
(2) Then, the recipient also verifies the last updater and its signature, if it exists and has write permission, with the updater’s public key PKc.
(3) The recipient searches the header part for its fingerprint check the permission.
(4) If the recipient has read permission, the recipient decrypts SSK with its paired private key SKr.
(5) Finally, the secret message is decrypted with SSK. Now the recipient gets the original data file.

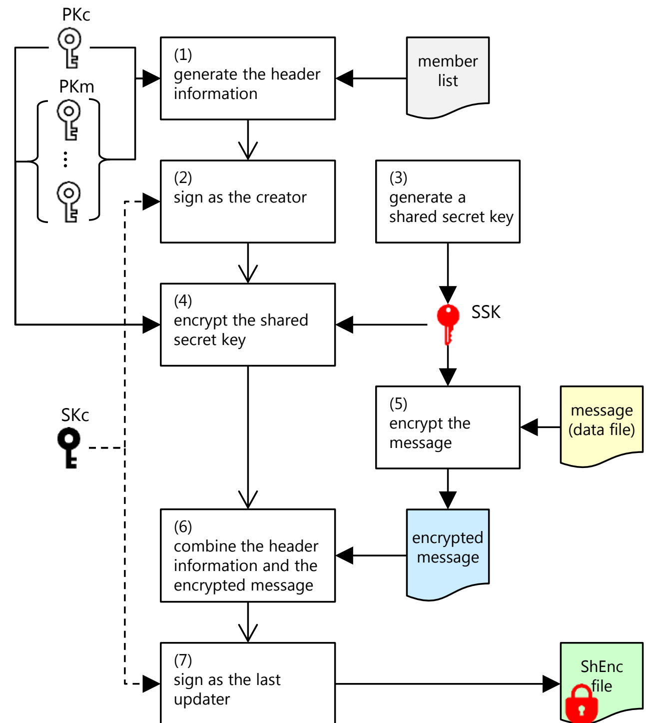

Figure 2: Encryption

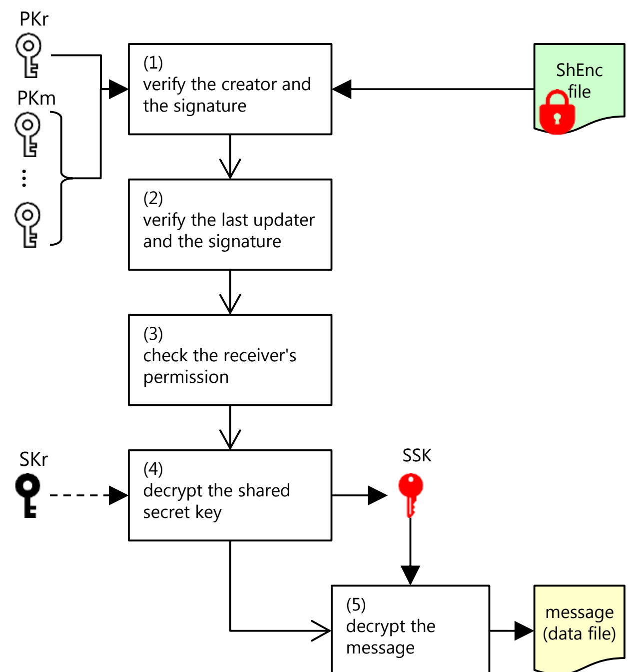

Figure 3: Decryption

Copyright © 2020 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

2.3.3 Security of ShEnc

Here, we consider a case that a malicious actor obtains a ShEnc file and tries to read the data file. The contents of data file are encrypted with the shared secret key SSK so that the key is necessary to read it. The ShEnc file only contains the encrypted form of SSK with each member's public key, i.e. ESSKm. To obtain SSK, the private key corresponding to the member's public key is required. However, every private key is kept secure by its owner. Hence, the malicious actor cannot obtain the private key corresponding to the public keys so that the actor cannot read the contents of the data file.

Let's consider another case that a malicious actor obtains a ShEnc file and sends falsify data. The file has two signatures to keep the integrity of contents. If the actor modifies the partial data like the member information, the hash value of the file become different and it doesn't match the signature. If the malicious actor modifies the whole data including creator information and signature, the data has consistency; however, the recipient can know the creator is different. The recipient doesn't have the malicious actor's public key and the forged ShEnc file is can't be trusted.

2.4 Implementation

We have developed a tool to pack and unpack a data file into a ShEnc file. The tool has the encryption and decryption functions with the verification following the algorithm we described above. In addition, it has public key management function for groups of members. The tool also provides the method to verify the public key using a fingerprint when registering a new member's public key. The update feature enables us to track the update history of a ShEnc file while the permitted members modify the data part.

Go language is used for CLI-based implementation [5]. Go is a modern open source programming language which has various standard libraries including cryptography. Using the innate cross compile construct, we can expect to generate high performance native execution code for many platforms.

3. Performance evaluation

In this section, we show the evaluation results of the proposed method and its implementations. The performance measurement employed the platform and components shown in Table 1.

Table 1: Performance evaluation environment

Platform	Apple iMac 21.5-inch, 2017 CPU: Intel Core i5, 2.8 GHz RAM: 8 GB
OS	macOS Mojave, 10.14
Go	version 1.12.7

We used /dev/urandom for random number generation. The processing time was measured using time command and bash built-in command in real time and CPU time, respectively.

3.1 File size

According to the file format shown in Figure 1, we can estimate the size S of a ShEnc file where n is the number of sharing members and s is a size function to compute the size of a given element. S is the sum of the sizes: header part S_H, shared secret keys and updater S_D, and file name and contents S_F.

$$S = S_H + S_D + S_F$$
$$S_H = 78 + s(\mathrm{PKc}) * (n + 1) + (s(\mathrm{PKm}) + 36) * n + s(\mathrm{SIGc})$$
$$S_D = 44 + s(\mathrm{ESSKc}) + (s(\mathrm{ESSKm}) + 2) * n + s(\mathrm{SIGu})$$
$$S_F = 10 + s(\text{data file name}) + s(\text{data file contents})$$

s(file name) and s(file contents) should be multiple of AES256 block size, i.e. 16 byte.

For example, if n members share a file and they use 4096-bit RSA keys, the resulted ShEnc file size is as follows.

$$S_H = 78 + 526 + (526 + 36) * n + 512 \quad = 1116 + 562 * n$$
$$S_D = 44 + 512 + (512 + 2) * n + 512 \quad = 1068 + 514 * n$$
$$S_F = 10 + s(\text{data file name}) + s(\text{data file contents})$$
$$S = 1116 + 562 * n + 1068 + 514 * n + 10 + s(\text{data file name}) + s(\text{data file contents})$$
$$= 2194 + 1076 * n + s(\text{data file name}) + s(\text{data file contents})$$

According to the equation, the overhead is about 2 kibibyte (KiB) at least and the size of a ShEnc file increases by about 1 KiB per one member. The binary prefix kibi means 2^{10} or 1024; therefore, 1 kibibyte is 1024 bytes [6].

3.2 Processing time

3.2.1 Data file size vs. processing time

We measured the processing time of encryption/decryption with changing the size of the data file. The size of the data file we tested were range from 1 KiB to 1 gibibyte (GiB) and the number of sharing members was fixed to 4 people. The Figure 4 shows the results of processing time.

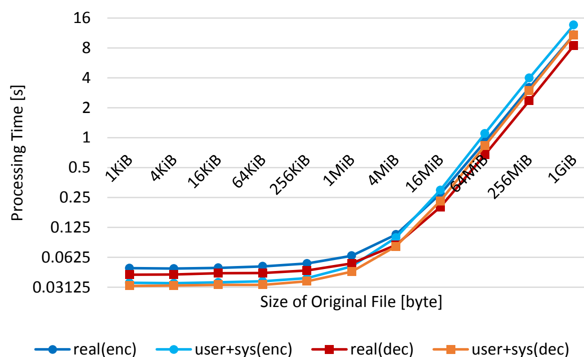

Figure 4: Data file size vs. processing time

Copyright © 2020 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

According to Figure 4, the processing time is almost the same when encrypting and decrypting data file which is smaller than 1 mebibyte (MiB). It increases linearly with data file size larger than 4MiB. When the data file is smaller than 1MiB, it takes more time to prepare file header, encrypt shared secret key, and sign the file, while the data file is larger than 4MiB, it takes more time to encrypt the data file contents. We can also see the encryption time is slightly longer than decryption time. The encryption has more process to encrypt the shared secret key with all the public keys.

3.2.2 The Number of members vs. processing time

Next, we evaluated the processing time with changing the number of sharing members. The number of members we tested was range from 1 to 4096 people and the data file size was 128 MiB. Figure 5 show the results.

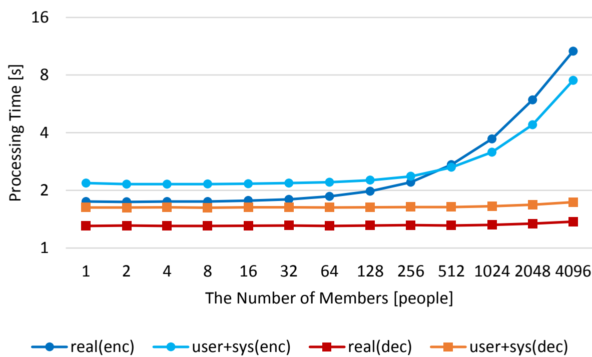

Figure 5: Number of members vs. processing time

According to Figure 5, the encryption time is almost the same when the number of members is smaller than 256 people. When the number of members is larger than 256, the encryption time increases linearly with the number of members. We can consider that the number of members is more than 256, the encryption includes more computation to encrypt the shared secret keys with more members' public keys and the processing time of the shared secret key encryption becomes longer than the processing time of others. Regardless of the number of members, the decryption time is almost the same. In decryption, it doesn't take much time because of only two decryption steps: shared secret key and data file decryption.

4. Conclusion

We designed and developed a system for end-to-end secure multi-party data sharing. The system has a unique format that includes a creator and sharing member information, i.e. fingerprints and public keys, signatures to verify integrity, and multiple encrypted forms of a single shared secret key. Each of which is encrypted by a member's public key so that only the members can decrypt one of them. The system supports to update the data to track the change history of the data file contents.

We developed an original tool to encrypt and decrypt data files. It is implemented in Go language and runs on many platforms, including macOS, Linux, and Windows. We performed the performance evaluation of the system. The results show that performance is good under the condition where the file size is smaller than 64 MiB and the number of sharing members is less than 256.

参考文献

[1] Goh, Eu-Jin, Shacham, Hovav, Modadugu, Nagendra, Boneh, Dan, “SiRiUS: Securing Remote Untrusted Storage”, *Network and Distributed System Security Symposium (NDSS)*, Vol.3, pp.131-145, 2003.

[2] US NIST, “Secure hash standard (SHS)”, *FIPS PUB 180-4*, Aug 2015.

[3] US NIST, “Announcing the advanced encryption standard (AES)”, *Federal Information Processing Standards Publication 197*, 2001.

[4] Rivest, Ronald L., Shamir, Adi, Adleman, Leonard, “A method for obtaining digital signatures and public-key cryptosystems”, *Communications of the ACM*, Vol.21, No.2, 1978.

[5] The Go Programming Language. https://golang.org

[6] Wikipedia, “Kibibyte”, https://en.wikipedia.org/wiki/Kibibyte

Copyright © 2020 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.