Double Fully Homomorphic Encryption for Tamper Detection in Incremental Documents

Vishnu Kumar, Bala Buksh and Iti Sharma

Abstract The famous scheme of Van Dijk, Gentry, Halevi and Vaikuntanathan (DGHV) style of Fully Homomorphic Encryption (FHE) is simple to implement. This paper proposes how this scheme can be modified and used for double encryption to secure files stored at third party like Cloud service provider (CSP). The files belong to an incremental text model which is followed by majority of confidential documents. Encryption is similar to a homomorphic hash function. A protocol is also proposed to detect if files have been tampered, and the tampering can be located up to single file level or up to a word of file level.

Keywords Incremental documents · Homomorphic encryption · Tamper detection

1 Introduction

Confidential documents mostly follow an incremental structure, where files arrive like chunks of a large file. Say files have arrived in sequence $f_1,f_2,f_3,\ldots,f_k$, then each of f_i is a file in its own, and any sequence from beginning $f_1 f_2 f_3\ldots f_k$ is also a file at the moment file f_k arrives. Tamper detection commonly relies on hashing and message authenticators (MAC) code computations. In case, these incremental documents are stored with a third party, like a cloud service provider, the files need to be stored in encrypted form. This encryption secures the crucial information stored in files from being leaked. At the same time, now files cannot be used, not even concatenated without decrypting them back. Considering the amount of data,

V. Kumar (✉) · B. Buksh
R.N. Modi Engineering College, Kota, Rajasthan, India
e-mail: vishnumkota@gmail.com

B. Buksh
e-mail: balabuksh@gmail.com

I. Sharma
Rajasthan Technical University, Kota, Rajasthan, India
e-mail: itisharma.uce@gmail.com

S.C. Satapathy and S. Das (eds.), *Proceedings of First International Conference on Information and Communication Technology for Intelligent Systems: Volume 2*, Smart Innovation, Systems and Technologies 51, DOI 10.1007/978-3-319-30927-9_17

downloading the files back from cloud to the user site, decrypting and then concatenating, then uploading to the cloud is an impractical solution. If this stupendous task is outsourced to the cloud, there will be a need to share the encryption key; losing all the point of encryption in the first place. Here we assume that cloud might be honest but curious. So, the first requirement is that CSP should be able to concatenate files in encrypted form only. This can be simply achieved if the files are encrypted at word level, that is, each word of the file is encrypted separately and concatenated to form a file. Though it can be taken to bit-level, it would be very expensive. Once this has been done, one can implement any message digest or hash computation function homomorphically over the files for the purpose of tamper detection. The real problem starts when the files are a part of an incremental document. The message digest or hash has to be now computed for f_1,f_1f_2,$f_1f_2f_3$ and so on. This requires a homomorphic hash as has been suggested in this paper.

2 Incremental Text Model

Documents like appeals, applications, project reports, annual reports etc. are crucial and large sized. They are incremental in both use and storage. Though so common in practice, the similar model for the soft copies of incremental documents has yet not been formalized. We present here an elementary idea how such documents can be stored and viewed as soft copies. Let every individual part of the document be called a file, denoted by f. At any point, the document is a sequential concatenation of files generated till now, $D_k = \langle f_1 f_2 .. f_k \rangle$. Since, the files are generated and maintained chronologically, for any file f_i and file f_j, $i < j$ suggests that file f_i has been generated before f_j. Files either have independent existence or the entire document has. No other sequence or combination of file can be retrieved or has any meaningful interpretation. Thus, at any point of time k, D_k is sequence of k files beginning at f_1 till f_k.

3 DGHV Style FHE Scheme

Homomorphic encryption is a technique which allows operations to be performed over encrypted data, and the results when decrypted give the same output as would have been produced by actual desired operation on plaintexts. Thus, the operation 'g' performed over cipher texts is homomorphic to desired function 'f' over plaintexts. If the operations 'f' contain any arbitrary circuit and homomorphic 'g' exist for any such 'f', the scheme is called fully homomorphic. The idea was asked for by Rivest et al. [1], and the first solution arrived in 2009 through Gentry [2]. The idea has been developed from bits to integers, based on several hard problems. The recent works based on ring learning with errors [3, 4] are efficient and secure enough. Yet, practical issues like number and size of keys involved, noise growth etc. make implementing FHE debatable.

In this section we first discuss the FHE scheme to be used, based on those suggested in [5], which in turn are symmetric versions of DGHV [6] style FHE systems. Any file is assumed to be divided into equal sized words, which may be 1, 2 or 4 bytes long or more as per the computation power available at the data owner. The size of word affects the runtime cost of encryption. File is to be encrypted before storage to ensure security. The key used remains secret with the data owner. Let the word size be w (256, 65,536 or 4,294,967,296), and length of key η bits. The file to be encrypted consists of 'B' words, and each word $b_1, b_2, \ldots, b_B$ is encrypted as

$$x_j = b_j + p * r_j \tag{1}$$

where p is the secret key, an odd number of length η bits. Each r_i is a random odd number of length η bits, $r_j \neq p, \forall j$. Thus, file $\langle b_1, b_2, \ldots, b_B \rangle$ is now encrypted and stored in Cloud as $\langle x_1, x_2, \ldots, x_B \rangle$. The length of each encrypted word, now onwards called cipherword, is of $O(\eta)$.

Each file is associated with a hash value, which is equivalent to bitwise XOR of all words of the file, producing a hash word of length $O(\lambda)$. Since the encryption is fully homomorphic, instead of computing XOR of all words and then encrypting it under key p, the encrypted hash value, y, can be directly computed as modulo sum of all cipher words. Here, we require a homomorphic hash, so the encrypted hash is re-encrypted under a new key q,

$$y = (\Sigma x_j)w \tag{2}$$

$$z = y + qs \tag{3}$$

where, q is a public key, an odd number of λ bits and s is a random odd number of λ bits, $s \neq q$. This produces the final hash value of a file to be stored as z, a number of length $O(\lambda)$ bits.

The fully homomorphic properties of the suggested scheme can be observed directly, as proved in [5].

Note on parameter selection: The word size, w, makes it mandatory that $\eta > \log_2 w$, so that modular arithmetic remains correct. Moreover, $\lambda > \eta$, for hash computations. Our experiments suggest that minimum difference of 2 be maintained.

4 Proposed Protocol

Let the incremental document be a chronological sequence of n files, $D = \langle f_1 f_2 \ldots f_n \rangle$. The user generates any file f_i, encrypts it into a sequence of cipherwords under key p using Eq. (1), and uploads it to the third-party storage (TPS). TPS computes the hash value of file as z_i using Eqs. (2) and (3) under key q. At a subsequent instant, file f_{i+1} is generated, and similar procedure is followed. At any arbitrary instant k, the document D_k is concatenation of k files $\langle f_1 f_2 . . . f_k \rangle$, it is required to have hash values associated with documents too. The hash values of documents are computed as

$$Hash(D_1) = Hash(f_1) = z_1$$

$$Hash(D_2) = Hash(f_1 f_2) = z_1 + z_2$$

Thus,

$$Hash(D_k) = Hash(f_1, f_2 . . . f_k) = \sum z_k.$$

We proceed to discuss how hash value of a document D_k can be verified for correctness.

```
ALGORITHM Verification
INPUT: Document $D_k$
OUTPUT: YES/NO
Step 1: For each file $f_i$ in $D_k$, compute the sum of
cipherwords as
            $A = \sum_{i=1}^{k} \left(\sum x_j\right)_{\forall x_j \, in f_i}$
Step 2: Let hash value of $D_k$ be $z_k$, then
            $B = z_k \mod q$
Step 3: If $A \equiv B \mod w$, return YES else return NO
```

The steps 1 till 4 of the protocol are repeated many times at arbitrary instances. Files are concatenated in chronological sequence. At any point k, the command is D_k and user may want to perform a tamper check. The request of tamper check is handled by sending hash value of current document. It is verified using verification algorithm. If verification is positive, protocol stops In case verification is negative, tamper is detected and user now proceeds to locate the tamper. This is done by running verification algorithm over all documents beginning at D_1. But now the hash values have to be recomputed by TPS under a new key sent by user. Let the point at which verification fails is t. This implies that file f_t is tampered. The error can further be located by checking file f_t. The cipher word which does not decrypt correct is tampered.

When the verification process returns a negative reply, a tamper in one or more files is expected. The protocol for tamper detection can be summarized as shown in Fig. 1.

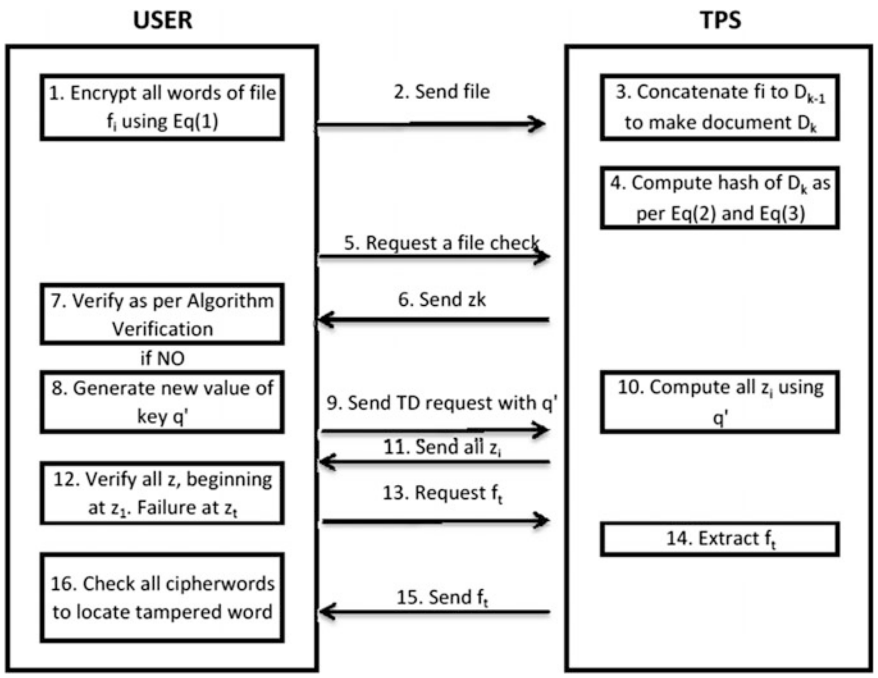
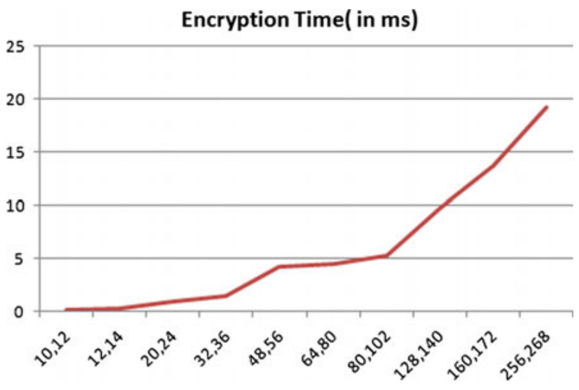

USER
TPS
1. Encrypt all words of file f_i using Eq(1)
2. Send file
3. Concatenate fi to D_k-1 to make document D_k
4. Compute hash of D_k as per Eq(2) and Eq(3)
5. Request a file check
6. Send zk
7. Verify as per Algorithm Verification
if NO
8. Generate new value of key q'
9. Send TD request with q'
10. Compute all z_i using q'
11. Send all z_i
12. Verify all z, beginning at z_1. Failure at z_t
13. Request f_t
14. Extract f_t
15. Send f_t
16. Check all cipherwords to locate tampered word

Fig. 1 Tamper detection protocol

5 Analysis of Proposal

The proposal was implemented as a Java program and growth of runtime recorded for different values of η, λ and w. The extensive results cannot be reported due to limitation of space. Figure 2 is a plot of encryption time with different combinations of η and λ, keeping $w = 256$. It can be seen that growth is quadratic as the value of parameters increase. This is apt even for a lightweight client.

Fig. 2 Growth of encryption time with increasing values of parameters



5.1 Performance Analysis

Formal analysis can be performed for each step involved. To encrypt a word of size $\log_2 w$ bits, under key of η bits, major cost is incurred during multiplication of p and r, making it $O(\eta^2)$ operation, since $\log_2 w \ll \eta$. Computing hash involves addition of $O(\eta)$, and multiplication of $O(\lambda^2)$. Time complexity of verification process for any document D_k is $O(k\lambda)$, thus depending on number of files in the document D_k. Tamper detection process, once the tampered file has been detected, involves checking all cipherwords of suspected file. This amounts to cost of encryption of all words of file that depends on both n and η^2.

5.2 Security Analysis

Brute-force method to guess a b-bit odd number is of effort $O(2^{b-1})$. Since the secret key 'p' is never shared, it can only be guessed, requiring $O(2^{\eta-1})$ effort. Forging a valid hash requires picking a correct number of length $(2\eta + 1)$, since 'q' may be leaked. This amounts to naive effort of $O(2\eta + 1)$.

At present, an effort of 2^{80} cycles is considered secure. This means that suggested scheme and protocol is secure at parameters $\eta = 40$, $\lambda = 42$. Considering other kind of attacks, a key of length 128 bits is considered secure enough. The parameter values $\eta = 128$, $\lambda = 140$, when experimented incur encryption time of less than 10 ms.

6 Conclusion

The security and tamper detection problem for incremental documents has not been considered till now. We present a simple solution based on DGHV style FHE. The protocol for tamper detection and location is simple yet powerful. Keeping the efficiency issue in mind, a lightweight scheme is suggested, which is fast enough even for very large keys.

7 Future Scope

Better candidates of FHE can be explored for a similar application. The overall protocol is still under implementation. We plan to analyze growth of verification time and tamper detection time with increasing size of file. Security analysis in context of known plaintext-ciphertext pairs of the suggested scheme is yet to be done. Formal ontology related to incremental documents can be a further research area.

References

1. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Found. Secure Comput. **4**(11), 169–180 (1978)
2. Gentry, C.: Fully homomorphic encryption scheme. In: Diss. Stanford University (2009)
3. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-LWE and security for key dependent messages. In: Advances in Cryptology–CRYPTO, pp 505–524. Springer, Berlin (2011)
4. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. In: Cryptology ePrint Archive, Report 2011/277
5. Aggarwal, N., Gupta, C., Sharma, I.: Fully homomorphic symmetric scheme without bootstrapping. In: International Conference on Cloud Computing and Internet of Things (CCIOT), Chengdu, China (2014)
6. Van-Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Advances in Cryptology–EUROCRYPT, pp 24–43. Springer, Berlin (2010)