On query result integrity over encrypted data

Ertem Esiner, Anwitaman Datta

Nanyang Technological University

Abstract

We leverage on authenticated data structures to guarantee correctness and completeness of query results over encrypted data. Our contribution is in bridging two independent lines of work (searchable encryption, and provable data possession) resulting in a general purpose technique, which does so without increasing the client storage overhead, while only a small token and a data structure is added to the server side (in comparison to a base searchable encryption without mechanisms for determining result integrity), where the data structure can simultaneously also be used for integrity checks on the stored data.

Keywords: searchable encryption, result completeness & correctness, integrity, byzantine servers.

1. Introduction

With the proliferation of data outsourcing in recent years, there has been immense associated security concerns. Intertwined with these security concerns are aspects of functionality. Broadly, the basic security concerns can be seen in the context of the CIA-triad (confidentiality, integrity and availability).

Confidentiality can arguably be achieved using encryption, but it interferes with functionality. This has triggered research on searchable encryption [1, 2, 3, 4], where the challenge is between the degree of confidentiality and flexibility. Consequently, research on fuzzy search [5, 6] to facilitate approximate queries has gained recent attention.

Another broad area of research revolves around the integrity of outsourced data [7, 8, 9]. The emphasis in these works is to verify whether the data that has been outsourced has been correctly retained at the third party.

This work is at the confluence of these two broad areas, and proposes a generic technique to determine the correctness and completeness of individual query results when (fuzzy) searchable encryption is deployed. Note that this is distinct from determining the integrity of the whole outsourced data. There are several new searchable encryption schemes, where verifiability of the query responses is considered [10, 11, 12, 13, 14], however these existing works are protocol specific and are not readily portable.

With respect to typical searchable encryption schemes, our approach introduces two additional storage overheads. First, the server needs to keep a hash value per keyword. Second, the server needs to keep a data structure constructed over the data blocks. The data structure however simultaneously amortizes integrity checks over the stored data [8, 15]. We use a simple searchable encryption scheme from Cash et al. [16] as a basis to explain our scheme.

There are other security (besides CIA) and functionality trade-offs with outsourced data, e.g., private information retrieval, differential privacy, etc. These are beyond the scope of this work and are not discussed any further.

2. Related Work

Being able to search over encrypted data is critical for preserving utility as well as confidentiality when storing data remotely. Song et al. showed a construction where the client generates a secure and efficient index (dictionary) to be stored along with the data [1]. To perform a search, a client provides necessary information to the server to deduce the indices of the searched keyword from the dictionary, and return those particular data parts. An index works as a mapping between a keyword and a list of document IDs in which the particular keyword appears. Generally the list of document IDs from the dictionary that are deduced by the server (as a response to a query by the client) are encrypted. Goh et al. present the first security definitions [2] (against Adaptive Chosen Keyword Attack). Later Chang et al. present stronger simulation based definitions [3]. Boneh et al. provide a public key encryption scheme to allow searchable encryption. Curtmola et al. provide improved definitions and new constructions [4]. There are also works which allow fuzzy keyword search that returns the results for the keywords which are close enough to the search key [5, 6, 17, 18]. Recently, dynamic solutions have also been proposed [16, 19, 20], where the techniques are more efficient than a naive solution of reconstructing the whole dictionary when a part of the data is updated.

There are some works that ensure correctness and completeness. Most of them consider (1) honest-but-curious (server) adversarial models where they do not consider the query results being wrong or incomplete [1, 3, 21, 22], others do not consider (2) dynamic data (updates on the data) [12, 23, 24]. For instance, Clarke et al. use a Merkle tree to authenticate the data and assume that the data structure is sorted depending on the queried keyword to provide completeness guarantees [12]. They do not consider dynamic data updates. Using a Merkle tree provides an efficient solution for static data, but it does not inherently support dynamic data. Kamara et. al. [24] provide a dynamic scheme yet the verification mechanism is for static data only. Likewise, some approaches are not readily portable, for instance Wang et al. [10] present a technique that uses the bloom filters and signatures and Chai et al. [11] propose a technique using Trie trees where the verification scheme is intertwined together with their query mechanism. Kurosawa et al.'s approach [13] supports dynamic updates where the verification complexity being linear over the data size can be considered as the downside of the scheme.

In table 1, we compare some of the recent works with this paper. The first threat model is a honest but curious (HbC) server where the server follows the protocol honestly, but only tries to learn information from the transactions. In this work we are interested in a harder problem, and consider byzantine servers that may

Property/Scheme	[1][21] [3][22]	[12]	[11]	[13]	this paper
Server Threat Model	HbC	Malicious	sHbC	sHbC	Malicious
Dynamic Data	-	No	No	Yes	Yes
Entangled Technique	-	No	Yes	Yes	No
Proof Complexity	-	log(n)	O(1)	O(n)	log(n)
Update Complexity	-	-	-	O(n)	log(n)

Table 1: Comparison with the previous works.

not return the search results as is. We divide them into 2; semi honest but curious (sHbC) and malicious. A sHbC server may execute the search partially and return an incomplete searching result honestly. On the other hand a malicious server additionally may alter the search result, trying to sneak non existing results in, or even can temper with the original data. We consider the proof and verification schemes in the mentioned works are dynamic if an authorized party can efficiently update the data while still being able to verify correctness and completeness of the future search results. In this paper, we are trying to achieve a technique that may work with all present searchable encryption techniques, thus we consider if a technique employed in any work is entangled with the particular scheme or can be employed separately in another scheme as well. Last but not least, we include the proof and update computational complexities with respect to the data size (n) to the comparison.

We provide a proof and verification mechanism that is simple, general and highly applicable to many (if not all) searchable encryption schemes (static and dynamic) present in the literature. Our scheme doesn't require a key, nor cryptographic functions processed by the server. The proof and verification complexities are logarithmic over the data size and it only requires the storage of expected 2 hash values per data block and a single hash value per keyword that may have been searched.

It is worth noting at this point that due to page limitation we have omitted the techniques for dynamicity and the proof-verify mechanism for an empty result set. We will make these two subsections available as online abstract and cite here upon publication.

3. Preliminaries: A Simple Searchable Encryption Primitive

We first describe a simple searchable encryptions scheme [16] to illustrate our proof mechanism. A dictionary is a list at the server that keeps keyword, block ID pairs. A search query should return the blocks with IDs that are listed with the keyword(s). The list should be masked and not leak (meta-)information about the keywords, such as the number of appearances of a keyword within the data, etc. We denote string concatenation as "||", e.g., "a"||"b" is "ab".

First, the client picks a secret key K and picks a pseudo-random function F (publicly available). Then she prepares the dictionary as follows. For each keyword w_i, she computes two keys $K1_i$ $(= F(K, 1||w_i))$ and $K2_i$ $(= F(K, 2||w_i))$ Then, for each block ID id related to w_i, she computes a pseudo random label l_{ic} $(= F(K1_i, c))$, where c is a counter on number of appearances of w_i and an encrypted identifier d_{ic} $(= ENC(K2_i, id))$. Note that $K1_i$ and $K2_i$ can only be computed by the client since she is the only one with the secret key K and knows the actual keyword w_i. At last, she stores all $< l, d >$ pairs to the server as the dictionary. The actual blocks' IDs in which the keyword appears, can be masked by adding extra random block IDs in the list.

To carry out a search, the client uses K and w_i and computes $K1_i$ and $K2_i$. She sends both of them to the server. Then, the server, iterating through c from 1 until it returns null, for each appearance of w_i, computes l_{ic} $(= F(K1, c))$ and collects the corresponding d_{ic} from the list and decrypts it with the key $K2_i$.

For the simplicity of elaboration, in the next section we represent the values in the dictionary openly instead of disguising them. Also, we substitute the computed search query by the searched keyword itself. In practice, the scheme itself allows the server to only decrypt the part of the dictionary (the relevant file IDs) about the searched items without knowing the actual query keyword.

4. Proof System Design

A response to a search query by the server is a part of the encrypted data. It may be a tuple from a database, a paragraph from a text, a slide from a presentation or a whole file. The below proof and verify mechanisms are described over a standard system model [1, 2, 4, 16, 19, 20]. The data is kept in blocks whose size may differ from file to file. We assume that a word is not divided across two blocks (this can be assured by the client's upload/update methods). The data is encrypted block-wise. A search query returns the corresponding blocks. As described in Section 3, the server keeps a dictionary which consists of keyword and block ID pairs.

In order for the server to prove to the client that the search result is correct and complete, the server needs to firstly prove that the returned data is a genuine part of the actual data that had been kept on behalf of the client, and secondly, that all the results associated with the search query are returned.

Left container in Figure 1 represents the server side of the system. The data structure we employ is RBASL [8] constructed over the data blocks. A node in RBASL consists of a rank, a level, a hash value (h(rank || level || below.hash || after.hash)) as shown in the Figure 1, where h is a collision resistant hash function. The rank value is the addition of the rank of the 'below' the 'after' nodes. For the leaf level nodes the 'below' rank is 1 if they are corresponding to a data block. The rightmost and leftmost nodes are sentinels and exists for convenience of the data structure/algorithms. If there is no 'after' link, the 'after' node's rank is considered to be 0. The hash value of the 'after' node is null if there is no 'after' link (may happen only on the leaf level) and the hash of the 'below' is the data block itself for the leaf level nodes.

Unlike DPDP [8], we do not employ the homomorphic tag mechanism from PDP [7], since for a search result, all the associated data blocks are returned. Therefore, there are no tag values (for blockless verification) stored at leaf level. The encrypted blocks are used to compute hash values of the leaf level nodes. This yields $1024 * n$ bits of saving, where n is the number of blocks and 1024 is the size of a tag employed in the DPDP scheme.

Our verification process proceeds in two layers.

4.1. Layer1: Correctness of the Response

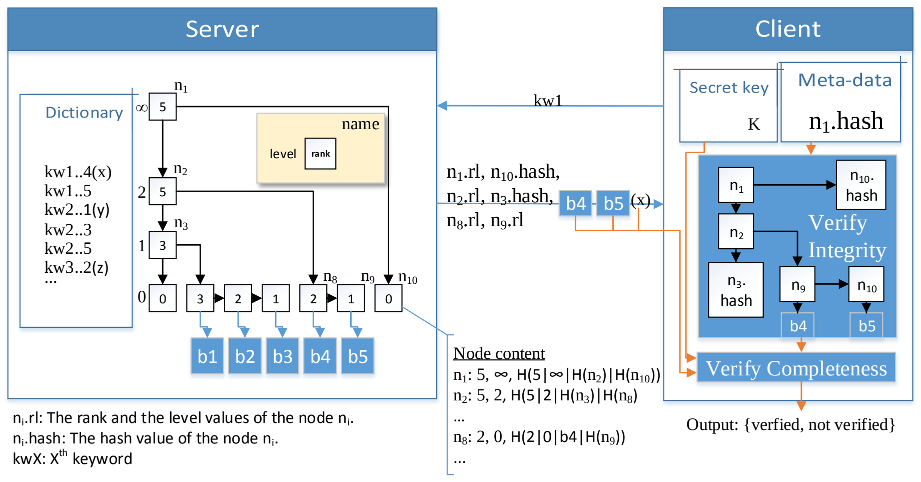


Figure 1: Server-Client interaction upon a search query.

The client first verifies the correctness of the blocks that the server sends back. Figure 1 presents a search query and its corresponding response by the server. The response includes the blocks that are listed in the dictionary for the searched keyword and the proof from the authenticated data structure. The proof is simply all the values that are necessary to compute the hash value of the root (n_1 in our example). As presented in Figure 1, one of the inputs to compute hashes of n_8 and n_9 are the blocks $b4$ and $b5$ (which are the results of the search query). The hash value of n_9 is an input for n_8's hash calculation, n_8's hash value is an input for n_2's and n_2's is an input for n_1's, as a daisy chain. After computing the hash chain, the client checks if the computed value is equal to the meta-data she keeps. If it is equal, then she is satisfied and ready to continue for the completeness verification.

The sketch of the security proof is as follows: Assuming there is a collision resistant hash function h [25], if the adversarial server, without having access to the actual blocks, sends a false proof that the client verifies (using a different set of values), it means another set of values create the same result at some step during hash calculations (for n_9, n_8, n_2, n_1 in our example), which would imply that we break the collision resistance of h.

So far we show the operations on one file for simplicity. An authenticated tree, such as a Merkle hash tree [26] can be employed as a file directory. This will generalize the idea over a whole file system.

4.2. Layer2: Completeness of the Response

After the correctness verification, the last operation by the client (as shown in Figure 1) is to verify the completeness. If the data could be sorted on the keywords, where all the blocks that include a keyword sit together in the leaf level of the data structure, then the solution is quite trivial. The sibling node to either ends of the leaf nodes that are returned are to be included in the proof, so that the verifying party can be satisfied that there are no other blocks that include the keyword [12]. While this solution may be effective for some cases such as health records since they are sorted based on the index/keyword to be queried, it is not readily applicable for all scenarios. A generic solution could be authenticating the dictionary as the data itself, so that we can check if the correct blocks are retrieved as listed in the dictionary. In this section, we present a more space, computation and communication efficient solution.

We divide verification of completeness into sub parts. Only if the first two are satisfied, the third proves completeness.

Requirement 1: All returned data parts are distinct.

Requirement 2: All returned data parts are from the latest version of the client's data.

Requirement 3: The number of keyword occurrences in the query response is equal to the actual number.

Requirement 1 is covered if (1.1) all blocks are genuine parts of the client data, and (1.2) all blocks are from different parts of the data. There may be two blocks that have the exact same content yet still are parts of the original file. While differentiating it, we still need to guarantee that one of the blocks is not replayed to fulfill Requirement 3. That can be done by ensuring that the location of the blocks are distinct. Our efficiency claim comes from the fact that Requirement 1 and 2 are covered by Layer 1 of the proof.

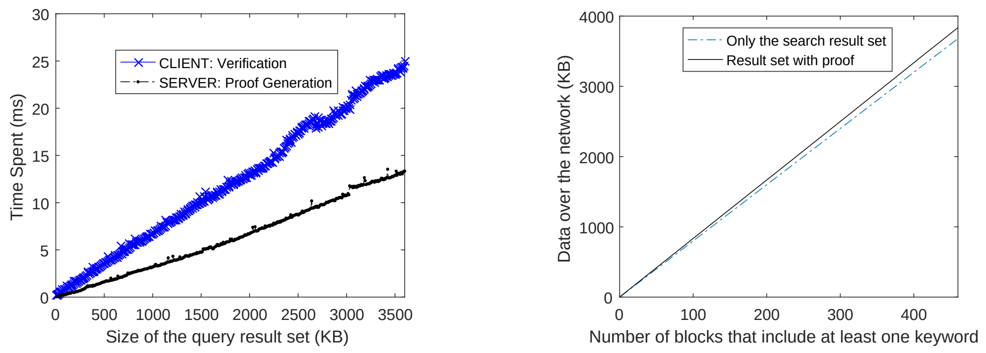
Requirement 1.1 is naturally covered by Layer 1 since the only set of values that can result with the meta data the client holds must include the genuine parts, as discussed in Section 4.1. Requirement 1.2 is covered since the correctness proof contains the rank values, thus the places of each received part are bound with the hash value of the root. Without the correct values of the rank information, the client can not be satisfied. Requirement 2 is covered by Layer 1 as well, given that the client's meta data is the hash value of the root of the latest version of the data structure. Therefore, a part from a different version will not satisfy the verification algorithm, if it is not the same data at the exact same location as the current version.

After the client is satisfied for the first 2 requirements, the only remaining control is the number of keyword appearances in the result set. To satisfy Requirement 3, the client should keep track of the number of occurrences of the keywords. The client can keep them in memory along with the meta data but while it increases the client storage needs, it also affects client portability negatively. Instead, we generate one extra value per keyword and upload them to the server along with the data. See in Figure 1, besides the first appearance of each keyword there is a completeness token (ct). In our example the ct values are x, y and z for keyword1, keyword2 and keyword3 respectively. A ct is a hash value computed using the corresponding keyword, number of appearances of that particular keyword in the data (not in the dictionary, since the dictionary may have some fake lines added by the client to hide information) and the secret key of the client as the inputs. For example, computation of ct for keyword1 is as follows: x = $hash$(keyword1 || 2 || K).

When the search results are received, the client decrypts and counts the appearances of the searched keyword. Then, she recreates the hash value by using the keyword, the count and the secret key, which all three are only known to the client. If the newly computed hash value is the same with the one received from the server, the client verifies the completeness of the search result.

5. Evaluation

We run our experiments on a 64-bit computer with a 3.2GHz Intel Xeon E5-1650 Processor with one active core, Dell Perc H310 (for high density, entry level servers) disk drive, 15.6GB main memory and 12MB L2 cache, running Ubuntu 12.10. As security parameters, for a 80-bits expected security we use 80-bit random numbers, and SHA-1 hash function. Reported results are the averages of 10 iteration of experiments. Next we show the overheads introduced by our proposed scheme and evaluate the advantages of our adaption of DPDP to our correctness proofs.

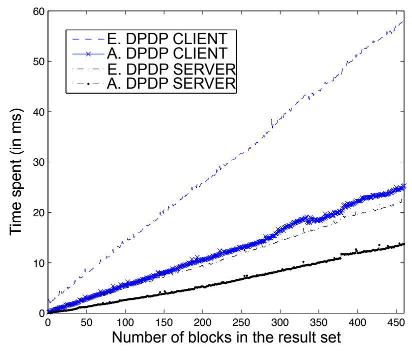


(a) Process overheads introduced for proof generation and verification.

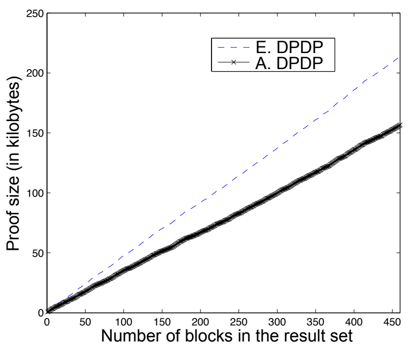
(b) Bandwidth usage for searchable encryption with/out correctness & completeness guarantees.

Figure 2: Overheads introduced.

Overheads introduced: A block is typically of size between 2KB to 16KB. We assume an average block is 8KB and the overall data stored at the servers is around 1GB. The time spent at the client side (to verify the proof) and at the server side (to generate the proof) are shown in Figure 2a. The server collects the values from a readily accessible data structure and sends them to the client. On the other side, the client needs to calculate the hash values from leaf to root for verifying the proof. Therefore, the client process time doubles the time spent by the server. The total latency added to a full round query-response of 3.6MB (result set size) is around 37ms which is imperceivable by a user. The communication cost comparison among the usual response and the one with the proof is shown in Figure 2b. Again, for 3.6MB of data returned, the overhead of the proof is around 150KB, which is barely perceivable. We made our experiments considering the worst case where the client queries don't follow a pattern. In fact, the closer the returned blocks are to each other in the data structure, the smaller the proofs and the faster it is to calculate them.



(a) Time comparison of employed DPDP and adapted DPDP.



(b) Employing DPDP and Adapted DPDP communication cost comparison.

Figure 3: Comparison with the original DPDP scheme.

Adapted data structure: In Figure 3a, we show the time spent at the server side for generating a proof for correctness of the returned blocks and time that the client spends to verify the proof. The main difference of our requirement compared to a DPDP system is that for a search query, the blocks that are included in the search result set needs to be returned to the client. As discussed in Section 4.1, the homomorphic tag verification mechanism can be removed and the blocks themselves should be used instead. Thus, the server doesn't combine the blocks together and the client doesn't need to multiply the tag values to compare the result with the sum of the blocks. Compared to DPDP, the proposed adaptions yield a considerable reduction of required time at both the server (∼40%) and the client (∼60%). There is also natural communication benefits from it. The tag mechanism in the DPDP scheme is put to reduce the communication cost by not sending the blocks. Instead, the tags are sent for each block, where each tag costs 1024 bits. Our approach does not even require the tags. In Figure 3b, we accordingly see a 30% communication gain compared to the DPDP solution.

Acknowledgements

Ertem Esiner's research was financially supported by the Singapore International Graduate Award (SINGA), grant no. TU-2013-R13145800000000.

References

[1] D. X. Song, D. Wagner, A. Perrig, Practical techniques for searches on encrypted data, in: IEEE S&P, 2000.

[2] E.-J. Goh, et al., Secure indexes, IACR Cryptology ePrint Archive (2003) 216.

[3] Y.-C. Chang, M. Mitzenmacher, Privacy preserving keyword searches on remote encrypted data, in: ACNS, Springer, 2005.

[4] R. Curtmola, J. Garay, S. Kamara, R. Ostrovsky, Searchable symmetric encryption: improved definitions and efficient constructions, in: ACM CCS, 2006.

[5] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, W. Lou, Fuzzy keyword search over encrypted data in cloud computing, in: IEEE INFOCOM, 2010.

[6] B. Wang, S. Yu, W. Lou, Y. T. Hou, Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud, in: IEEE INFOCOM, 2014.

[7] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, D. Song, Provable data possession at untrusted stores, in: ACM CCS, 2007.

[8] C. Erway, A. Küpçü, C. Papamanthou, R. Tamassia, Dynamic provable data possession, in: ACM CCS, 2009.

[9] A. Juels, B. S. Kaliski Jr, Pors: Proofs of retrievability for large files, in: ACM CCS, 2007.

[10] J. Wang, H. Ma, Q. Tang, J. Li, H. Zhu, S. Ma, X. Chen, A new efficient verifiable fuzzy keyword search scheme, JoWUA.

[11] Q. Chai, G. Gong, Verifiable symmetric searchable encryption for semi-honest-but-curious cloud servers, in: IEEE ICC, 2012.

[12] A. Clarke, R. Steele, Secure and reliable distributed health records: achieving query assurance across repositories of encrypted health data, in: IEEE HICSS, 2012.

[13] K. Kurosawa, Y. Ohtaki, How to update documents verifiably in searchable symmetric encryption, in: CANS, 2013.

[14] W. A. Drazen, E. Ekwedike, R. Gennaro, Highly scalable verifiable encrypted search, in: IEEE CNS, 2015.

[15] E. Esiner, A. Kachkeev, S. Braunfeld, A. Küpçü, O. Özkasap, Flexdpdp: Flexlist-based optimized dynamic provable data possession, ACM TOS (2016).

[16] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, M. Steiner, Dynamic searchable encryption in very large databases: Data structures and implementation, in: NDSS, 2014.

[17] A. Boldyreva, N. Chenette, Efficient fuzzy search on encrypted data, IACR Cryptology ePrint Archive (2014) 235.

[18] C. Liu, L. Zhu, L. Li, Y. Tan, Fuzzy keyword search on encrypted cloud storage data with small index, in: IEEE CCIS, 2011.

[19] E. Stefanov, C. Papamanthou, E. Shi, Practical dynamic searchable encryption with small leakage, IACR Cryptology ePrint Archive (2013) 832.

[20] M. Naveed, M. Prabhakaran, C. A. Gunter, Dynamic searchable encryption via blind storage, IACR Cryptology ePrint Archive (2014) 219.

[21] S. M. Bellovin, W. R. Cheswick, Privacy-enhanced searches using encrypted bloom filters, IACR Cryptology ePrint Archive (2004) 22.

[22] R. Curtmola, J. Garay, S. Kamara, R. Ostrovsky, Searchable symmetric encryption: improved definitions and efficient constructions, IOS Press JCS (2011).

[23] K. Kurosawa, Y. Ohtaki, Uc-secure searchable symmetric encryption, in: IACR FC, 2012.

[24] S. Kamara, C. Papamanthou, T. Roeder, Dynamic searchable symmetric encryption, in: ACM CCS, 2012.

[25] J. Katz, Y. Lindell, Introduction to modern cryptography: principles and protocols, CRC Press, 2007.

[26] R. C. Merkle, A certified digital signature, in: CRYPTO, 1990.