DOC⋆: Access Control for Information-Theoretically Secure Key-Document Stores

Yin Li,[1] Sharad Mehrota[2], Shantanu Sharma[3], and Komal Kumari[3]
[1]Dongguan University of Technology, China. [2]UC, Irvine, USA. [3]New Jersey Institute of Technology, USA.

ABSTRACT

This paper presents a novel key-based access control technique for secure outsourcing key-value stores where values correspond to documents that are indexed and accessed using keys. The proposed approach adopts Shamir's secret-sharing that offers unconditional or information-theoretic security. It supports keyword-based document retrieval while preventing leakage of the data, access rights of users, or the size (*i.e.*, volume of the output that satisfies a query). The proposed approach allows servers to detect (and abort) malicious clients from gaining unauthorized access to data, and prevents malicious servers from altering data undetected while ensuring efficient access – it takes 231.5ms over 5,000 keywords across 500,000 files.

1 INTRODUCTION

While secure data outsourcing and query over encrypted data has been widely studied over the past two decades [1, 2], the problem supporting access control over ciphertext has received little attention. This paper focuses on access control in the context of key-document (KD) stores — a type of KV store wherein the value corresponds to a document. Such a document may contain additional keys that may also be used to index the document, *e.g.*, a document (value) may consist of a doctor's note, and it may be indexed based on tags extracted from the text in the note. Such KD stores often store data in the form of an inverted index of doc-ids of documents associated with the key. As in regular KV stores, access to documents is based on keys. Popular KV stores such as Redis [3] allow storage of KD stores in addition to regular KV pairs.

We consider the problem of outsourcing when the inverted index of doc-ids/file-ids based on keywords, the set of documents/files, and the access control rights are outsourced in ciphertext to the cloud. A query for a keyword k over the outsourced database would retrieve all documents containing k for which the user has been granted access rights by the database owner (DBO).

In our model, neither the cloud nor the clients are trusted. Clients access only data to which they have access rights. The technique ensures that the cloud does not learn cleartext data, the client queries, or which clients have access rights to which data.

Access Control in Key-Value Stores. Access control in regular KV stores can be specified at either the ***record-level*** or at the ***key-level***. At the record-level, a DBO specifies who can have what type of access to which record (*i.e.*, a KV pair). In contrast, at the key-level, if a client is allowed/denied access to a key k, then the client is allowed/denied access to all KV pairs for the key k. In key document (KD) stores, as in regular KV database, access control can be specified at the ***key-level*** (a client is allowed/denied access to documents containing a given key) or at the ***document-level*** when users are explicitly allowed/denied access to certain documents.

While record/document-level access control is more popular, systems such as Redis support key-level access control as well [4], since key-based access can often be much easier to specify, easier to implement, and scales well with a large number of documents and a large number of clients. Consider, for example, a DBO with hundreds of thousands of documents. It is often easy for DBO to define access policies based on a limited set of keywords, *e.g.*, permit Alice to access all documents containing the keyword "Urgent" but not those containing "Finance." Also, for the system perspective, it becomes easier to check user's access rights for a key compared to checking access-rights of many files during query execution. Specifying and managing policies at the document-level, especially, when there are many clients, becomes difficult.

Key-level access control, nonetheless, adds complexity since now if a client has access to a keyword k_1 and not k_2, then a document d containing both k_1 and k_2 must not be returned, while documents containing k_1 but not k_2 should be returned.[1]

At first glance, one could convert a key-level access control specification to a document-level representation for which solutions have been explored in the literature [6, 7]. However, such a conversion is challenging when there are a large number of documents and clients, and, furthermore, the access control policies may dynamically change. For instance, say a DBO wishes to change Alice's policy to allow temporary access to documents containing the keyword "Finance" in addition to those containing "Urgent." If access control implementation was achieved through a document-level control, then DBO will need to determine a set of outsourced documents that contain the keyword (possibly tens of thousands), update the access control of these documents individually, and have to change the policies again when the temporary access is to be revoked. In contrast, with key-level access control, DBO would only need to update the policy for the relevant keywords, making the process significantly more efficient.

DOC⋆: A Key-based Access Control Mechanism. This paper focuses on the key-level access control in the context of key-document (KD) stores. Other types of access control, *e.g.*, document-level access control in KD stores, or record-level access control in KV stores are relatively simpler and have also been studied in the previous literature [6, 8, 9] and the (simpler-version of the) solution we develop can also be applied to the key-level access control in KV stores.

We develop an access control mechanism, entitled DOC⋆ (where ⋆ refers to STorage with Access control and secuRity) when secret-sharing is used to outsource data. Secret-sharing, unlike encryption mechanisms which are computationally secure, is unconditionally or *information-theoretically secure*, regardless of the adversary's

[1] DOC⋆ uses a policy model that permits only explicitly allowed actions, due to its stronger security guarantees as discussed in [5], ensuring that if DBO accidentally omits an access permission, the default is set to be denial, thereby preventing unauthorized access. Although such a denial might be inconvenient for the client, it does not compromise the system's security.

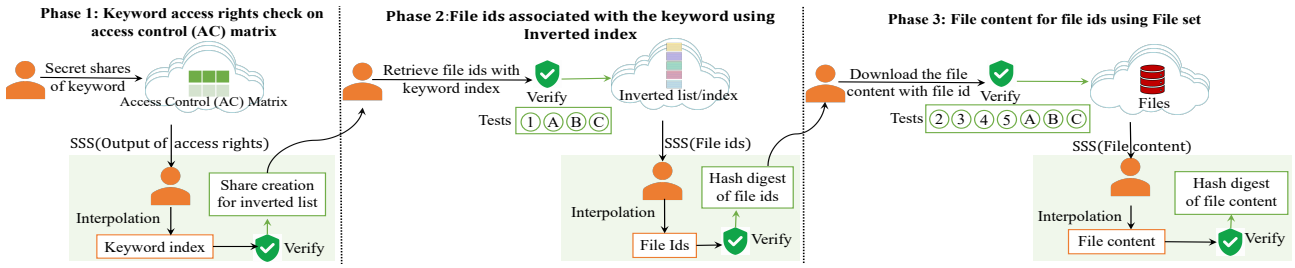


Figure 1: Execution of three phases of DOC*.

computational capabilities. Several additional benefits of secret-sharing (over encryption-based techniques) are well recognized in the literature: *distributed trust* (moving trust from a single server to multiple servers) to avoid insider attacks [10, 11]; *computational efficiency* to perform addition and multiplication on ciphertext; avoiding the risk of a single point of failure [12]; avoiding the risk of data theft [13, 14]; and tolerating malicious servers. Secret-sharing has gained popularity for data processing [15–19] and multi-party computation [20–24] by both academia and industries. While we develop our access control mechanism using secret-sharing for outsourcing, homomorphic encryption [25], which also offers addition and multiplication over ciphertexts, can also be used.

Code of the paper is available in [26]. Appendices provide: (*i*) dynamic operations–adding/deleting new files with new/existing keywords, (*ii*) access grant/revocation, (*iii*) methods for enhancing Phase 3, (*iv*) client-side result verification methods, (*v*) formal security proofs with proofs of theorems given in this paper, (*vi*) degree reduction method, and (*vii*) six additional experiments.

2 PRELIMINARY

This section overviews our model, secret-sharing, and the desired security requirements.

2.1 Model

Our model consists of three entities:

(*i*) **Database Owner (DBO)**: outsources ciphertext data along with access rights in ciphertext form to the cloud. DBO defines access rights for different users based on keywords and outsources them in the form of an ***Access Control (AC) matrix*** of $\alpha \times \beta$, where α is the number of clients and β is the number of keywords. A row of AC matrix represents a capability list for a client. Additionally, DBO outsources an ***inverted index/list*** with β rows, one corresponding to each keyword k_i containing doc-ids/file-ids for documents/files containing k_i. AC matrix, inverted index/list, and documents/files are outsourced using secret-sharing to a (set of) servers.

(*ii*) **Servers**. A set of $c > 1$ servers stores data and executes queries. Particularly, an i^{th} server stores the i^{th} shares of *AC matrix, inverted index, and documents*. Upon receiving a query keyword k from a client, the servers return (a subset of) documents in which k appears that do not contain any keyword k_j for which the client does not have access rights. A minority (see next subsection for the details on this) of the servers can also be malicious and may interfere with the execution of the protocol to prevent correct execution. Malicious servers may collude amongst themselves and/or with the clients.

(*iii*) **Clients**: send queries to retrieve all documents/files associated with a specified keyword. Clients can be honest or malicious. An honest client follows the protocol correctly. A malicious client can try to download documents/files/data to which they may not have access. Malicious clients can collude with malicious servers.

2.2 Shamir's Secret-Sharing (SSS)

We use SSS [27]. Let S be a secret. Let p be a prime number. Let $\mathbb{F}_p$ be a finite field of order p. SSS requires a secret owner to distribute S into $c{>}c'$ shares by randomly selecting a polynomial of degree c' with c' random coefficients, s.t. $f(x){=}S{+}a_1x{+}\cdots{+}a_{c'}x^{c'}$, where $f(x){\in}\mathbb{F}_p[x]$ and $a_i{\in}\mathbb{N}$ $(1{\leq}i{\leq}c')$. Secret S is distributed into $c{>}c'$ shares, by computing $f(x)$ for $x{=}1, 2, \ldots, c$. An i^{th} share is placed at the i^{th} server. S can be reconstructed using Lagrange interpolation [28] over any $c'{+}1$ shares. An adversary can reconstruct S, iff they collude with $c'{+}1$ servers. Thus, by choosing c' as the degree of the polynomial, the scheme remains secure even if c' servers can collude. Further, $3c'{+}1$ shares are needed to tolerate c' malicious servers and ensure error detection and correction (as proved in [29]).

SSS is *additive homomorphic* [30]: the sum of shares at servers produces the sum of cleartext value after interpolation. SSS is, also, *multiplicative homomorphic* [30]: servers can locally multiply shares, and the result can be constructed at the *owner* if having enough shares, as each multiplication increases the polynomials' degree. The multiplication result reconstruction, on a different entity that does not possess the original secret shares, requires an additional degree reduction step at the server, which comes with additional cost. We use terms '*multiplicative secret-sharing/ multiplicative shares*' and '*Shamir's secret-sharing/ secret-shares*' interchangeably.[2]

2.3 Security Requirements

We list the security requirements for our problem setting below:
(*i*) ***Confidentiality:*** Servers (even if servers collude) must never learn cleartext data outsourced by DBO, access control rights given by DBO to a client, or the query keyword the client wishes to retrieve.

(*ii*) ***Read obliviousness.*** Servers must not be able to distinguish between two or more queries based on which data is returned to clients, *i.e.*, access-patterns/identity of the object is hidden from servers.

(*iii*) ***Restricted access to the client.*** The client cannot fetch any documents that contain a keyword to which they do not have access.

[2]While SSS requires more than one server, this assumption is supported by various factors, including economic incentivization (as collusion goes against their financial interests), legal regulations, and jurisdictional boundaries. Such servers can be selected on different clouds, making the assumption more realistic. The proliferation of independent cloud vendors over the years has led organizations to adopt multi-cloud solutions to mitigate risks, *e.g.*, vendor lock-in, and to enhance fault-tolerance [31–35]. Leveraging multi-cloud environments enables organizations to outsource shares more effectively.

Note that such documents may contain other keywords for which a client has access. Nonetheless, the presence of a restrictive keyword should prevent client from gaining access to such a document.

3 HIGH-LEVEL OVERVIEW OF DOC*

A client's query is processed in three phases. In the first phase, the servers, given a query for a keyword, check AC matrix and return to the client a vector of size β with one number for each keyword. The returned vector contains a zero (in SSS form) if the client has access to the keyword, else a random value. In the second phase, the client can then use the index of the returned value zero to pose a query to retrieve all file-ids containing the keyword. Finally, based on the file-ids, the client fetches the files in the third phase. We illustrate DOC* strategy using an example.

Example of DOC*. Consider the following files/documents:

1: The King of Torts is a suspense novel written by John Grisham.
2: Stephen King is known as the King of Horror.
3: . . .

A DBO first constructs ***access control (AC) matrix*** by creating columns with *searchable keywords* – the keywords using which a client can search for files (words such as "a," "is," and "hello" are removed since clients do not search based on such words). AC matrix in DOC* is a $\alpha \times \beta$ matrix, where α corresponds to the number of clients/groups in an organization and β is the number of keywords. A row of AC matrix represents a capability list for a client. Suppose, two (searchable) keywords are King and Horror. DBO creates capability lists for a client, Alice, see below, where $\mathbb{M}(*)$ denotes a secret-share. This shows Alice is allowed to search for the keyword King (indicated by $\mathbb{M}(0)$), not Horror (indicated by a random number). All values except the client name in AC matrix are secured using SSS, (the method is explained below §5). Second, DBO outsources an ***inverted list/index*** using SSS, see below, where the gray part is not outsourced to the cloud and is written to show the row corresponding to the keyword. Finally, DBO outsources all files using SSS.

Keywords →	$\mathbb{M}$(King)	$\mathbb{M}$(Horror)	. . .
Alice	$\mathbb{M}(0)$	$\mathbb{M}(19)$	. . .
⋮	⋮	⋮	⋮

Access control (AC) matrix.

Positions/Row-id	File-ids
1 (King)	$\mathbb{M}(1)$, $\mathbb{M}(2)$
2 (Horror)	$\mathbb{M}(2)$
⋮	⋮

Inverted index.

Suppose, Alice searches for the keyword Horror. DOC* will execute the following three phases/rounds (as depicted in Figure 1).

***Phase 1*:** Alice will send to servers, the keyword Horror using SSS, servers will perform a computation *obliviously* (*i.e.*, without knowing access-patterns – memory address/identity of the object) over the capability row of Alice in AC matrix and return a vector of size equal to the number of keywords in AC matrix. Alice interpolates the values and obtains all random numbers, showing either she does not have access to the keyword Horror or the keyword Horror is not present, and then, she terminates the protocol. Now, suppose, Alice searches for the keyword King, then she will obtain a vector as $\langle 0,$ random number, $\ldots\rangle$, after interpolation, where the position of zero refers to the row/index of the inverted index, containing all file-ids associated with the keyword King.

***Phase 2*:** Alice, to fetch the desired row of the inverted list, sends a vector $\langle 1, 0, \ldots, 0\rangle$ of size β (the number of rows in the inverted list) with all positions zeros and only the desired position with one using SSS. Servers verify the correctness of the vector (*i.e.*, containing all zeros except a single one at the desired position), then perform a dot product between the vector and the inverted index and return the result. Then, Alice learns file-ids 1 and 2, after interpolation.

***Phase 3*:** Alice tries to fetch files 1 and 2. The servers will obliviously return only file 1, not file 2, due to having the keyword Horror, which she is not allowed to search. At the end of Phase 3, Alice learns file 1 in cleartext after interpolation, not file 2, indicating that file 2 contains at least one keyword without search permission to her. ■

Information leakage. In an ideal setting, a client should be able to retrieve a file containing the queried keyword only if that file does not include any keyword to which the client is denied search access. Also, the client should not learn any extra information, such as the presence or absence of keywords in AC matrix, the queries made by other clients, or the number of retrievable files relative to the total number of files that match the queried keyword. DOC* prevents all such information leakage, with one exception during Phase 3.

In Phase 3, if a file is not retrieved, the client can infer that it contains query keyword (since its file-ids were retrieved in Phase 2) and includes at least one keyword to which client is denied access (since the file itself was not returned). Although the client cannot access file's content, they may attempt multiple queries across all keywords to infer the set of keywords associated with that file. If the client knows and has access to all the keywords, except one k_j, this will reveal that file contains k_j. In all other scenarios, client cannot determine which keywords with disallowed access appear in the file.

Such a multi-query leakage, when the client knows and has access to all keywords but one, can be prevented by randomizing the file-ids returned in Phase 2, for example, by adding to file-ids a random number derived from a query-specific seed or by padding with fake file-ids (which are excluded in Phase 3). These techniques can help prevent the client from inferring the presence of restricted keywords in the file.

4 A BASELINE & CHALLENGES

We establish a **baseline solution** for secure key-based access control over secret-shared data to identify the specific challenges that DOC* needs to address and to compare performance against DOC*. The baseline offers weaker security, as it assumes the clients to be honest that DOC* does not. As such, this baseline can be viewed as being unfair to DOC* given DOC* stronger security. Nonetheless, it suffices since, as the experiments will show, DOC* significantly outperforms the baseline. We choose such a baseline since it is easy to implement using state-of-the-art secret-sharing tools MP-SPDZ [36].

In the baseline solution, we store AC matrix using Shamir's secret-sharing (SSS) and execute state-of-the-art secret-sharing-based MP-SPDZ [36] to check the client's access rights for searching a query keyword. On success, the servers return all the doc-ids associated with the query keyword to the client using the inverted index. In the baseline: (*i*) AC matrix is created for a single client where columns correspond to one of 5,000 keywords and cell values correspond to the access rights of the client, and (*ii*) inverted index contains

Notations	Meaning
α	# clients or the group allowed to perform search operation
β	# unique keywords across all files/documents
γ	The maximum number of files associated with a keyword (known to every entity)
δ	# files/documents ($\delta \geq \gamma$)
p	A prime number used as modulo in secret-sharing
$\mathbb{M}(x)$	Multiplicative shares or Shamir's secret-share (SSS) of x
sw	A keyword at a server in AC matrix
uw	A keyword at a client
$A \odot B$	Dot product between A and B, where $A \odot B = \sum_{0<i<n+1} A[i] \times B[i]$
$\mathbb{M}(AC)[i]$	The i^{th} value of access control matrix, denoted by AC
ACT_pos	The third row having hash digest of the positions in AC matrix

Table 1: Frequently used notations in this paper.

5,000 rows, one for every 5,000 keywords in the same order as they appear in AC matrix, and each row contains all doc-ids associated with the keyword. While DOC* will fetch secret-shared files also, we leave this step aside from the baseline solution. This baseline solution to answer a client request took 24.23 seconds. The reasons of this inefficiency will be discussed in Challenge 1 below.

To understand the summary of operations supported by DOC*, we classify them in terms of the challenges we addressed. Later sections will provide details of all these operations.

Challenge 1: Efficient query processing. The baseline solution took 24.23s, where the offline/preprocessing and online/computation phases of multi-party computation (MPC)-based processing took 19.77s and 4.46s. Let us focus only on the online phase of MPC. For checking access rights, MP-SPDZ [36] uses the equality test of [37]. The online phase took 2×8 rounds of communication among servers, where 8 rounds were used to check the keyword and another 8 rounds were used to check the access right. It incurred ≈50MB dataflow among servers, while the size of ACT matrix was only 157KB. The reason of the overwhelming dataflow and the number of rounds is as follows: In each online round, 81ℓ bits flow among servers for the multiplication protocol involved in the equality check, where ℓ is the number of bits. Thus, to check access right over 5,000 keywords that are represented as 64 bits each, it requires 2×8×81×64×5000 ≈51.84MB. Finally, a dot product between the output of the access check and the inverted index is performed, resulting in all the doc-ids containing the keyword.

Our solution (§6.2). DOC* develops an **efficient** protocol for obliviously (*i.e.*, without revealing the identity of the keyword in AC matrix) checking the access rights of the client and returning the doc-ids. To bring efficiency for the *same operation*, our protocols *do not need* communication among servers if clients are assumed to be trusted or use two rounds of communication, each transmitting a few integers, when clients are not assumed to be trusted, to obliviously verify clients' behavior at the server. This makes our protocol at least 45 times faster than MP-SPDZ — while the online phase in the baseline took 4.46s, our new method took only 95ms (16ms for checking the access rights and 79ms for performing the dot product).

Challenge 2: Obliviously returning only files having no keywords for which access is denied. On receiving the file-ids, the client retrieves the file content. But now, the client may behave maliciously and try to fetch any file. Further, the file may contain a keyword for which the client does not have search access, and such a file should not be returned by the server. Of course, MP-SPDZ can address this; however, it will incur significant computational overhead.

Our solution (§6.4). DOC* develops a **verifiable, oblivious** file retrieval protocol. To fetch a file, the client creates a vector containing all zeros except for one at the desired position. We develop oblivious algorithms to verify the correctness of the vector at the server and oblivious algorithms to ensure the file has no keyword to which access is disallowed. The entire process does not reveal to *servers which file the client is fetching and whether the file contains a keyword to which the client does not have access rights or not* by always returning a file (real/dummy).

Challenge 3: Randomization of polynomials after multiplication — producing irreducible polynomials. When servers perform multiplication on the secret-shared data and the query, the resulting polynomial becomes reducible (*i.e.*, not fully random, because such a polynomial can be factorized and potentially found by exhaustive search), which may, hence, reveal some information about the secret-shared data to the client. The famous BGW protocol [29] can make the polynomial irreducible, with a high cost among the servers due to the communications required by interpolation and resharing.
Our solution (§6.1 and §6.2.1). DOC* develops a secure and computationally efficient method compared to BGW to make polynomials irreducible and fully randomized by eliminating the need for interpolation and resharing among servers for randomization. For this, distributed secret-shared random numbers are generated at the server before the query execution and used to randomize a polynomial.

Challenge 4: Dealing with malicious clients colluding with a minority of the servers. MP-SPDZ, when using SSS, does not deal with a case when malicious clients can collude with a minority of the (malicious) servers. This collusion can reveal additional information to the client, *e.g.*, nullifying the impact of making the polynomial irreducible, as discussed above.

Our solution. DOC* protocols deal with malicious clients colluding with a minority of the servers, making it impossible for the client to deduce any information about the secret. For example, the *presence of a minority of the servers colluding with malicious clients does not impact the process of randomization of polynomials*. In DOC*, the client sends a one-hot bit vector in shared form to the servers. However, a malicious client may create incorrect vectors by either placing one at the wrong places or having non-binary values. To detect such malicious behavior of clients by the server, DOC* develops three Tests: A, B, and C (§6.5).

5 DATA OUTSOURCING IN DOC*

This section develops a method for DBO to outsource a set of files using SSS to servers, by first, creating three data structures: an access control matrix, an inverted index/list, and a file set, and then, creating shares. Below, we explain the three data structures:

(1) ***Access-control (AC) matrix**:* contains access control information for each keyword and each client/group of clients (*e.g.*, groups can be CS, EE, and ME departments). Let α be the number of clients/groups. Let β be the total number of (searchable) keywords in all the documents. AC matrix contains α+2 rows and β+2 columns (see Table 3). Each row corresponds to a client/group, and each column corresponds to a keyword. Each row contains zero or a random number in each column, where a zero in the $(i, j)^{th}$ cell

File-ids	File content
1	How are you
2	Are you Ana
3	Fig is a fruit

Table 2: Cleartext files.

Keywords →	Are	Ana	Fig	Fake
Starting address in inverted index	1	2	3	0
Hash digest →	H(1)	H(2)	H(3)	H(0)
Clients' information ↓				
Lisa	**0**	**1**	**2**	**0**
Ava	**3**	**0**	**0**	**0**

Table 3: Access control (AC) matrix.

Positions	File-ids
1 (are)	1, 2, 0, H(2, H(1, H(are)))
2 (Ana)	2, 0, 0, H(0, H(2, H(ana)))
3 (Fig)	3, 0, 0, H(3, H(Fig))
0 (Fake)	0, 0, 0, H(0, H(Fake))

Table 4: Inverted index.

file_ids	AP list	File content with digest
1	1,0,H(1)	How are you, H(how are you, H(1))
2	1,2,H(1)+H(2)	Are you Ana, H(are you Ana, H(2))
3	3,0,H(3)	Fig is a fruit, H(Fig is a fruit, H(3))
0	0,0,H(0)	Dummy, H(Dummy, H(0))

Table 5: Files.

Keywords	112816	112412	161918
Hash	$\mathbb{M}(H(1))$	$\mathbb{M}(H(2))$	$\mathbb{M}(H(3))$
Lisa	**1**	**2**	**3**
Ava	**4**	**1**	**1**

Table 6: Share1 of AC matrix.

Keywords	112817	112413	161919
Hash	$\mathbb{M}(H(1))$	$\mathbb{M}(H(2))$	$\mathbb{M}(H(3))$
Lisa	**2**	**3**	**4**
Ava	**5**	**2**	**2**

Table 7: Share2 of AC matrix.

Keywords	112818	112414	161920
Hash	$\mathbb{M}(H(1))$	$\mathbb{M}(H(2))$	$\mathbb{M}(H(3))$
Lisa	**3**	**4**	**5**
Ava	**6**	**3**	**3**

Table 8: Share3 of AC matrix.

File-ids
$2, 3, \mathbb{M}(x_1)$
$3, 1, \mathbb{M}(x_2)$
$4, 1\ \mathbb{M}(x_3)$

File-ids
$3, 4, \mathbb{M}(y_1)$
$4, 2, \mathbb{M}(y_2)$
$5, 2\ \mathbb{M}(y_3)$

File-ids
$4, 5, \mathbb{M}(z_1)$
$5, 3, \mathbb{M}(z_2)$
$6, 3, \mathbb{M}(z_3)$

Table 9: Three shares of inverted index.

indicates that the client i is allowed to search for the j^{th} keyword, while a random number indicates otherwise. Random numbers are selected carefully, see the end of this subsection.
One of the additional rows contains keywords, and another row contains the hash of the row-id corresponding to the keyword in the inverted index.[3] One of the additional columns contains clients/groups name, (or the client's provable identity that is sent with a query) in cleartext, and another column contains a fake keyword with allowed access to conceal volume (*i.e.*, the count of the files) during Phase 3.

(2) ***Inverted list/index:*** contains, for each keyword, doc-ids/file-ids in which the keyword appears. For enabling *verification by clients*, each row of the index includes a hash digest over the doc-ids associated with the keyword.[4] To make each entry of the inverted index of an identical length, fake file-ids (say zero/random numbers greater than real doc-ids) are added (to reduce space and computation overheads by adding fake doc-ids, §7 develops a method). One row is added with a fake keyword and fake doc-ids to hide allowed/ denied access from servers; see the last paragraph on this issue in §5.2.

(3) ***File (set)/Documents.*** Each file contains a file-id, the file content, a hash digest over the file-id and the content (enabling verification of the file content by the client), and an AP (absence/presence of keywords) list. The AP list prevents servers from sending a file containing at least one keyword to which the client has no access. Two possibilities for storing AP lists are: (*i*) AP list is of size either β, each value with 0 or 1, showing the absence or presence of each of the β keywords in the file, where 0 refers absence; otherwise, 1. (*ii*) AP list contains the column number of AC matrix corresponding to the keywords that appear in the file, along with the *sum of the hash digest* of the positions (see Table 5). The first alternative of AP list offers full security, while the second reveals the number of keywords appearing in a file to the client – we will discuss this in §6.4.1. Finally, DBO adds a dummy/fake file that is used to hide the volume from servers during Phase 3 of file retrieval.

Creating and outsourcing shares. DBO generates ***four multiplicative shares*** of the AC matrix, inverted list, and files using random ***polynomials of degree one***.[5] DBO *does not create shares of the client's name in the AC matrix*. Shares of number are created straightforwardly using SSS. To create shares of English keywords, they are converted to numbers (using letter positions or ASCII codes), equalized in length by adding a random number, and then multiplicative shares are generated. Each i^{th} share is outsourced to the $\mathcal{S}_i$ server.

Selecting random numbers for AC matrix. Suppose two keywords in AC matrix are 'AB' and 'ABC.' Their numerical representations could be '0102' and '010203,' respectively. Suppose only 26 numbers are needed. We can add any number greater than 26 and smaller than 100 to AB (*e.g.*, 010299), making its length identical to ABC's length. Random numbers in the cells of AC matrix for denied access are always greater than the numerical representation of any string; *e.g.*, if the largest string in numerical form is 2727 (*i.e.*, ZZ), random numbers in any cell must be greater than 2727 (the reason will be clear in STEP 2 of Phase 1 in §6.2).

5.1 Example of Data Outsourcing

Cleartext files and AC matrix. Table 2 shows three files in cleartext. Suppose, the three files have three keywords: `are`, `ana`, `fig`. Based on these keywords, an AC matrix (see Table 3) is created for two clients, `Lisa` and `Ava`. The first row of AC matrix keeps the keywords. The second row (gray-colored) keeps the row-id of the inverted list in which the keywords appear. The third row (blue-colored) is for hash digest for those row-ids. Yellow-colored part shows the capability list of clients. $\langle 0, 1, 2, 0\rangle$ indicates that `Lisa` has access to those files containing `are` and `fake` keywords, while `Ava` can access files containing `ana`, `fig`, and `fake` keywords.[6] ***For the purpose of simplicity, we put 1,2,3, as random numbers, in AC matrix*** and do not show shares of the fake keyword in Tables 6,7,8. The gray part in the AC matrix indicates a connection with the inverted list and is not outsourced.

Inverted index/list. Table 4 shows the inverted index for the three keywords with the hash digest over the file-ids. A fake file-id 0 has been added to the keywords `Ana` and `Fig`, thereby all keywords have an identical number of file-ids. Another zero is added to all the keywords, and this zero shows an empty space for handling the insertion of the new files. DBO can add such zeros multiple times, depending on the insertion workload. A row with fake file-ids is also added. In the inverted list, the gray-colored part, which is written for the purpose of explanation, is ***not*** outsourced.

File data structure. Table 5 shows the file-ids, the content of files, the hash digest over the file content and its id, and AP list. The

[3] The hash digest will be used in Phase 3 to ensure that the client does not retrieve a file if it contain a keyword to which access is denied.

[4] Hash digest can be computed by chaining the digests of the doc-ids and the keyword; *e.g.*, the hash digest for a keyword appearing in files f_1 and f_2 would be: $H(f_2, (H(f_1, H(\text{keyword}))))$, where H is a secure hash function [38]. This chaining process allows the addition/deletion of the doc-ids to/from the inverted index, without recomputing the hash digest for all file-ids associated with a keyword (explained in detail in Appendix B.1.1 in [26]).

[5] We selected polynomials of degree one under the assumption that no cloud server will collude, hence two shares are enough. However, we perform one multiplication at the server, so three shares are needed. The fourth share is used for client-side result verification purposes.

[6] DOC* can handle multiple keywords connected by conjunctions and disjunctions by considering them a single composite keyword.

last row shows a dummy file-id with its dummy content. The fake keyword, fake file-id, and fake file do not need to be at the end of the data structures. They can be placed at any place.

Share creation. DBO creates SSS of AC matrix, inverted list, and the files. DBO represents keywords as: `are` as `11,28,15`, `ana` as `11,24,11`, and `fig` as `16,19,17`, and creates shares of such numbers. ***For the purpose of simplicity, we select a single polynomial ($f(x) = (x+s) \bmod p$, where $p = 500009$, s is a secret, and $x \in \{1, 2, 3\}$) to show the shares of AC matrix and inverted list.*** However, in real deployments and in our experiments, DOC* selects different random polynomials for every secret. Tables 6, 7, and 8 show three shares of AC matrix of Table 3. Three shares of the inverted list of Table 4 are shown in Table 9, where $\mathbb{M}(*)$, where $* \in \{x_i, y_i, z_i\}$ (i=1,2,3) indicates multiplicative shares/SSS of the hash digest. We do not show shares of fake items, and the remaining shares of AC matrix, inverted list, and files (Table 5) can be created similarly, but are not shown here due to space limitations.

5.2 Discussion

Information leakages from the secret-shared data. Since DBO selects random polynomials to create shares, a keyword/file-id appearing at multiple places (either inverted list, AC matrix, or files) will look different in ciphertext; preventing an adversary from learning information by looking at AC matrix, inverted list, and files. Leakage that occurs from AC matrix is the number of keywords and the length of the keyword. The inverted list may reveal the maximum number of files with a keyword. The files and AP list may reveal their size. These leakages can also be prevented by padding, *e.g.*, padding keywords to the same length, padding dummy keywords to AC matrix, padding dummy file-ids to the inverted list, or adding dummy content to the files. Padding strategy increases the data size, and so, the processing time. After interpolation, the client discards dummies. A common way is to use a predefined dummy value, *e.g.*, -99, known to all participating entities. As we are working on English letters, a dummy value of -99 or 27 will work; for a larger character set like Unicode, we need to select a larger dummy, *e.g.*, -9999999999.

Why padding is secure? Padding makes two keywords "Jo" and "John" appear as "12,15,dummy,dummy" and "12,15,08,14" in cleartext, and DBO will create shares of such numbers. Since shares are generated with random polynomials, repeated appearances of the same number appear different, preventing adversaries from distinguishing between real and fake keywords or deducing their lengths.

Why keywords are not encoded in AC matrix? We represent keywords in letter positions/ASCII codes and pad each to the maximum length—increasing the size of AC matrix. Using encoding, *e.g.*, a hash map, to allocate numbers to keywords, we could reduce keyword size but introduce issues: how to avoid false positives, how clients know the position in the map, how to handle collisions, and how to insert new keywords. Thus, we did not use encoding methods.

Different access rights for keywords. Servers may learn the query, if they can distinguish keywords. Suppose, there are only two keywords, $k \neq k'$, a client has access to search only k, and *all such are known to servers*. Now, if servers return files after checking access rights, they will learn that the query is for k. DOC* can also avoid this by returning fake files for k' (*i.e.*, executing Phase 2 and Phase 3, even if access is not allowed). To handle this, DBO inserts two fake entries with allowed/disallowed access rights in all data structures, (we show one of them with allowed access in Tables 3,4,5).

6 QUERY PROCESSING IN DOC*

DOC* has three interrelated phases to fetch the file containing the desired keyword. In the following, we develop protocols where the clients do not verify the results obtained from the server, while the server verifies the client queries in Phase 2 and Phase 3. Below, for simplicity, we use three servers and assume one of them is malicious. Table 1 shows frequently used notations.

6.1 A Building Block: Distributed Secret-Shared Random Number Generation

This section develops a method of generating secret-shared random numbers that will be used in different steps in DOC*. Moreover, such random numbers can also be used for other purposes, such as hiding secrets by adding random number during degree reduction [29].

Objectives of using distributed secret-shared random numbers. The primary purpose of using random numbers in DOC* query processing is to preserve data confidentiality throughout various computational steps. For instance, in Phase 1 (Step 2), the servers obliviously evaluate whether a client has access to a queried keyword; if access is denied, servers return a random number. Another example is Phase 3 (Step 8), where servers obliviously add random numbers to the file content if it contains a keyword to which the client has no search access, while the client tries to retrieve the file.

These random numbers also serve a second critical function: constructing the constant term $\mathbb{M}(0)$ ***of degree two*** that is used to randomize the shared polynomial after multiplication. Particularly, servers perform a single multiplication between a query q and one of their data structures $\mathbb{M}(a)$, say $\mathbb{M}(c) \leftarrow \mathbb{M}(a) \times \mathbb{M}(q)$. Here, we observe that a single multiplication can obscure the true value $\mathbb{M}(a)$. However, the polynomial corresponding to $\mathbb{M}(c)$ is reducible, which may potentially lead to information leakage. To mitigate this, we propose the addition of a quadratic term $\mathbb{M}(0)$ to $\mathbb{M}(c)$, which introduces uncertainty in the reducibility of the resulting polynomial while preserving the polynomial's constants (the secrets).[7]

Design objectives. Generating these random numbers is independent of a query and should be done before query execution for efficiency. The naïve way of generating random numbers using a common seed at all the servers is not viable, as colluding servers could reveal the seed, defeating randomness's purpose. Instead, random numbers are generated in a distributed manner, so servers will have only their

[7] A well-known method for polynomial randomization is BGW [29]. However, BGW method has certain limitations that preclude its direct application in our scenario. Firstly, it necessitates that all servers select random polynomials with zero constants and aggregate these polynomials to the original polynomial without any form of verification, thereby failing to prevent malicious behavior by servers. In contrast, our method allows the servers to verify $\mathbb{M}(0)$; see details below on verification. Secondly, BGW method can only randomize the polynomial during the query processing, which introduces additional communication and computational overhead. In contrast, our method generates $\mathbb{M}(0)$ before query execution and uses it during query processing.

share of the random number at the end of the computation, but do not know the actual random number.

Distributed random number generation introduces another challenge: a malicious server can potentially compromise the correctness of the output, *i.e.*, the shares of the random numbers at each server. To address this, non-malicious servers, first, verify the generated random numbers before using them in further computations.

Aside. Reusing the same random number or $\mathbb{M}(0)$ across multiple queries can leak information to the client. Thus, *for each query*, new random number and $\mathbb{M}(0)$ are generated in advance.

Method. We assume that each server $\mathcal{S}_{z\in\{1,2,3\}}$ generates, say q, non-zero random numbers, creates shares of those using *polynomial of* ***degree one***, and distributes appropriate shares to the other servers. On receiving shares from the other servers, $\mathcal{S}_z$ aggregates them. $\mathbb{M}(\text{RN})[]$ denotes such random numbers in share form at each server. The value of q could be the number of keywords in AC matrix if the random numbers are used in Phase 1 or the size of a file if they are used in Phase 3.

Meanwhile, $\mathcal{S}_z$ generates $\mathbb{M}(0)$ ***of degree two*** by multiplying $\mathbb{M}(\text{RN})$ with c_z locally (where c_z is the input for SSS, *i.e.*, $f(x{=}c_z) = (ac_z{+}b)c_z$ and $f(x) = ax{+}b$ corresponds to the polynomial of $\mathbb{M}(\text{RN})$).

Verification of $\mathbb{M}$(RN)[]. Before using $\mathbb{M}(\text{RN})[]$ in query execution, non-malicious servers verify them, since malicious servers can distribute any number as share, contaminating $\mathbb{M}(\text{RN})[]$ and $\mathbb{M}(0)$. To verify, $\mathcal{S}_z$ does: $a_z \leftarrow \sum_{1\le i\le q}(PRG(seed)\times\mathbb{M}(\text{RN})[i])$ mod p.

In other words, each server first generates q new different random numbers using a common seed and then performs a dot product between the new random number and $\mathbb{M}(\text{RN})$. Then, each two servers interpolate $\langle a_z, a_{z+1}\rangle$, $\langle a_{z+1}, a_{z+2}\rangle$, and $\langle a_{z+2}, a_z\rangle$. This approach leverages the observation that legal random number shares correspond to a polynomial of degree one, whereas illegal ones correspond to a polynomial of any other degree. Thus, a polynomial of degree one leads to consistent interpolation results between any two servers, a property that does not hold for a polynomial of any other degree.

Theorem 1. ***If a server creates shares of random numbers incorrectly,*** $LI(a_z, a_{z+1})$***,*** $LI(a_{z+1}, a_{z+2})$***, and*** $LI(a_{z+2}, a_z)$ ***at*** $\mathcal{S}_z$ ***will produce different results, where*** $LI(*)$ ***means Lagrange interpolation.***

6.2 Phase 1: Access Control Check over AC Matrix

High-level idea. Phase 1 works over AC matrix (Table 3) and enables the client to determine their search access rights for queried keywords. If the client has access to a queried keyword, they learn the row-id of the inverted list, which contain all file-ids associated with the keyword. If access is denied, the client learns nothing — specifically, they cannot distinguish whether the keyword is not present in AC matrix or whether access is denied. E.g., Phase 1 allows the client Lisa to learn whether she is allowed to search for a keyword or not, but she gains no further information. Even in the presence of collusion between Lisa and a minority of servers, she remains unable to infer whether keywords such as Ana or Fig are present in the AC matrix or if access to them is denied; see Table 3.

A client sends keywords in SSS form. Servers operate obliviously over AC matrix to find the keyword and the access right, and then, return a vector of SSS form to the client. After interpolation at the client, the vector contains *a zero, if the client is allowed to search the keyword*; otherwise, all random numbers.

Algorithm design objectives. We need an algorithm in Phase 1 to:
(*i*) *Check access rights over ciphertext.* Since both access rights and the client's query are in ciphertext, the server needs to find the access rights without knowing the query keyword in cleartext and the access decision in cleartext.
(*ii*) *Prevent information leakages.* The server must not learn additional information, such as which keyword the client is searching for, the client's access right, and the content of the access control matrix. The client must not learn additional information, such as access rights of other clients and queries executed by other clients.

6.2.1 *Algorithms in Phase 1:* work as follows:

Step 1: ***Client:*** sends their (provable) identity (*e.g.*, name) and a keyword (uw) in SSS form to three servers $\mathcal{S}_{z\in\{1,2,3\}}$. Secret-shares of the keyword are created using the same strategy as used by DBO to create shares of keywords (mentioned in §5). Note that the client can select any polynomial to create shares.

Step 2: ***Servers:*** performs the following:

1. Obliviously checking access rights and polynomial randomization. Servers find a desired row corresponding to the client in AC matrix based on the client's identity and perform the following:

$$\mathbb{M}(ans\mathcal{S}_z)[i] \leftarrow [((\mathbb{M}(sw)[i]{-}\mathbb{M}(uw)+\mathbb{M}(AC)[i])\times\mathbb{M}(\text{RN})[i]) + \mathbb{M}(0)]\,\text{mod}\,p, \forall i \in \{1,\ldots,\beta\}$$

$\mathcal{S}_{z\in\{1,2,3\}}$ subtracts each keyword ($\mathbb{M}(sw)[i]$ — the first row of Table 3) of AC matrix with the keyword $\mathbb{M}(uw)$, received from the client, and adds the corresponding access control value $\mathbb{M}(AC[i])$.

Note that $\mathbb{M}(sw)[i]{-}\mathbb{M}(uw)+\mathbb{M}(AC)[i]$ may provide additional information (*e.g.*, random numbers associated with denied access or which keywords are present) to the client, $\mathcal{S}_z$ multiplies $\mathbb{M}(\text{RN})[]$ to each output (to make the client unable to distinguish keywords). Further, the servers need to ***randomize*** the whole result by adding $\mathbb{M}(0)$ (as produced in §6.1) with different secret polynomials of degree two for each query to avoid factorization of the polynomial at clients ($\mathbb{M}(0)$ with polynomial degree one can be recovered by a single server). We discuss the security analysis later.

2. Sending results. $\mathcal{S}_z$ sends a vector, having β numbers in SSS form (where AC matrix contains β keywords) to the client. *If the client is allowed to search for the queried keyword, the above equation will produce zero in SSS form; otherwise, a random number.*

Step 3a: ***Client:*** performs Lagrange interpolation (denoted as LI) over the vectors received from servers, *i.e.*,

$$vecR1[i] \leftarrow LI(\mathbb{M}(ans\mathcal{S}_1)[i], \mathbb{M}(ans\mathcal{S}_2)[i], \mathbb{M}(ans\mathcal{S}_3)[i]).$$

If $vecR1[]$ contains only random numbers, it means that the client is *not* allowed to search for the keyword. Otherwise, $vecR1[]$ contains *only one zero at the position corresponding to the keyword's position in AC matrix or inverted list.*

6.2.2 ***Example of Phase 1:*** is given in Table 10. *For simplicity, we do not add* $\mathbb{M}(0)$ *in the example.*

6.2.3 ***Correctness.*** SSS is somewhat homomorphic and does not affect the final result of the expression. According to the formulation of $ans\mathcal{S}_z$ presented in STEP 2, only if the query keyword matches the i^{th} keyword in AC matrix and the corresponding related access control number in the capability list of the client $AC[i]$ is zero, $ans\mathcal{S}_z$ will be zero. Otherwise, the client always obtains nonzero numbers, indicating that the keyword is not in AC matrix or the client has no right to search it. Further, the non-zero random number of $\mathbb{M}(AC)$ (allocated using the method of §5) prevent false positives, *i.e.*, $\mathbb{M}(sw)[i]-\mathbb{M}(uw) \neq \mathbb{M}(AC)[i]$.

6.2.4 ***Security Discussion.*** ***Servers:*** (*i*) receive from a client a keyword uw of SSS form; thus, servers cannot learn the keyword in cleartext; (*ii*) perform identical operations on each keyword of AC matrix, making it impossible for them to learn anything from the operations they perform; thereby access-patterns are hidden from servers; (*iii*) always return a vector of length β; thus, volume of the answer is also hidden from servers.
Clients: cannot learn information about the random numbers used to indicate denied access for a keyword, as servers multiply random numbers $\mathbb{M}(\mathsf{RN})[]$ to obfuscate the subtraction results. Even if a minority of the servers collude with a malicious client, they cannot learn $\mathbb{M}(AC)[i]$ of a keyword with denied access. Also, a malicious client colluding with malicious servers cannot learn the access right information of other clients or queries by them in cleartext due to not having enough shares. Furthermore, a client that does not knowing a keyword $\mathbb{M}(sw)[i]$ in AC matrix cannot distinguish between the absence or presence of the keyword or disallowed access to keywords, due to multiplying $\mathbb{M}(\mathsf{RN})[]$; see the following theorems.[8]

Theorem 2. ***If a malicious client colludes with a minority of malicious servers, such malicious entities cannot deduce*** RN ***to obtain additional information, except for*** $ans\mathcal{S}_z$***, in Phase 1.***

Multiplication between two shares may lead to related secret polynomial reducible. Also, the client may create $\mathbb{M}(uw)$ twice or more using the same polynomial, then deduce $sw[i]+AC[i]$ by finding the common divisor between two secret polynomials related to $(\mathbb{M}(sw)[i]-\mathbb{M}(uw)+\mathbb{M}(AC)[i])\times\mathbb{M}(\mathsf{RN})[i]$ for two queries. Randomization of $\mathbb{M}(uw)$, *i.e.*, adding $\mathbb{M}(0)$, avoids this.

Theorem 3. ***Randomization of*** $\mathbb{M}(uw)$ ***prevents malicious clients from deducing*** $sw[i] + AC[i]$ ***over multiple queries, even colludes with a minority of malicious servers.***

6.2.5 ***Cost Analysis.*** Computation cost at a server and the client is $O(\beta)$. Communication cost between a server and a client is $O(\beta)$.

6.3 Phase 2: Finding File-Ids from Inverted List

High-level idea. Phase 2 works on the inverted list to allow clients to retrieve the file-ids associated with the keyword they queried in Phase 1, provided that the client has search access to the keyword.

[8] This is important when revealing the presence/absence of highly sensitive keywords (*e.g.*, nuclear).

`Lisa` wants to fetch files containing a keyword `are`, represented as 112815. Note that `Lisa` is allowed to search for `are`; see Table 3. Suppose $\mathbb{M}(\mathsf{RN}_1) = [4, 5, 6]$, $\mathbb{M}(\mathsf{RN}_2) = [5, 6, 7]$, $\mathbb{M}(\mathsf{RN}_3) = [6, 7, 8]$. *For simplicity, we do not add* $\mathbb{M}(0)$ *here.*
STEP 1: ***Client.*** `Lisa` creates SSS of `are`, using a polynomial $f(x) = (5x+s) \bmod p$, where $p = 500,009$: for x=1 : 112820, which is sent $\mathcal{S}_1$, for x=2 : 112825, which is sent $\mathcal{S}_2$, and for x=3 : 112830, which is sent $\mathcal{S}_3$.
STEP 2: ***Servers:*** Based on Table 6, $\mathcal{S}_1$ performs the following:

$$(112816-112820+1)\times 4 \bmod p = 499997$$
$$(112412-112820+2)\times 5 \bmod p = 497979$$
$$(161918-112820+3)\times 6 \bmod p = 294606$$

$\mathcal{S}_1$ sends 499997, 499997, 294606 to `Lisa`. Based on Table 7, $\mathcal{S}_2$ performs the following:

$$(112817-112825+2)\times 5 \bmod p = 499979$$
$$(112413-112825+3)\times 6 \bmod p = 497555$$
$$(161919-112825+4)\times 7 \bmod p = 343686$$

$\mathcal{S}_2$ sends 499979, 497555, 343686 to `Lisa`. Based on Table 8, $\mathcal{S}_3$ performs the following:

$$(112818-112830+3)\times 6 \bmod p = 499955$$
$$(112414-112830+4)\times 7 \bmod p = 497125$$
$$(161920-112830+5)\times 8 \bmod p = 392760$$

$\mathcal{S}_3$ sends 499955, 497125, 392760 to `Lisa`.
STEP 3A: ***Client:*** `Lisa` performs interpolation and obtains the results: $[0, 498397, 245520]$ that indicates that `Lisa` is allowed to search the keyword `are`, which appears at the first position.

Table 10: Example of Phase 1.

E.g., if Lisa has searched for the keyword 'Are' in Phase 1, then after Phase 2, she will learn that files 1 and 2 contain the keyword 'Are.'

A client sends a vector, containing all zeros except for a single one, in SSS form to retrieve file-ids associated with the keyword they searched in Phase 1. The one is placed in the vector according to the row-id, revealed to the client in Phase 1. Before query execution, servers obliviously verify the **correctness of the vector** (*i.e.*, containing all zeros except a single one at the desired position), ensuring that clients do not fetch file-ids associated with a keyword that is not allowed to be searched, **even if they skip Phase 1**. If the vector is *correct*, servers perform a dot product between the vector and the inverted index and send the output of the dot product (*i.e.*, file-ids) to the client, who learns the file-ids after interpolation.

Algorithm design objectives. An algorithm in Phase 2 needs to:
(*i*) *Handle a malicious client.* Although the output of Phase 1 informs the client whether they are authorized to proceed to Phase 2 and, if so, reveals the specific row index in the inverted list, a malicious client may attempt to fetch an arbitrary row of the inverted list. Thus, the servers must be able to obliviously verify that the client's request corresponds to a row of the inverted list that is associated with a keyword the client is authorized to search. This requirement presents a significant challenge, as the query keyword, the Phase 1 result, and Phase 2's query vector are all in the ciphertext at the server.
(*ii*) *Prevent information leakages.* The servers must not learn any additional information, such as the keyword of Phase 1 and the requested row-id of the inverted list.

6.3.1 ***Algorithms in Phase 2:*** work as follows:

STEP 3B: ***Client:*** creates a vector v of length β containing zeros except for a single one at the i^{th} position that corresponds to the

We continue with the example of Phase 1, given in Table 10.
STEP 3B: ***Client:*** `Lisa` after STEP 3A creates a vector $\langle 1,0,0\rangle$, since the keyword `are` appears at the first index in the inverted list (see Table 4). Finally, `Lisa` creates three multiplicative shares of the vector using $f(x)=(10x+s) \bmod p$, where $p=500009$: $\langle 11,10,10\rangle$ sent to $\mathcal{S}_1$, $\langle 21,20,20\rangle$ sent to $\mathcal{S}_2$, $\langle 31,30,30\rangle$ sent to $\mathcal{S}_3$.
STEP 4: ***Servers:***, first performs the three tests of §6.5 to ensure the vector has only one and all zeros, and then the following Test 1 to ensure `Lisa`'s access to the keyword:

$$\mathcal{S}_1\colon [(1,2,3)\odot(11,10,10) \bmod p] = 61$$
$$\mathcal{S}_2\colon [(2,3,4)\odot(21,20,20) \bmod p] = 182$$
$$\mathcal{S}_3\colon [(3,4,5)\odot(31,30,30) \bmod p] = 363$$

Finally, each server sends the output of the test to other servers and interpolates them. The final answer is $\langle 0\rangle$, showing `Lisa` has created the vector correctly (*i.e.*, she has access to search the keyword). Afterward, servers perform a dot product between the vector and three share tables given in Table 9:

$$\mathcal{S}_1 : \langle (2,3),(3,1),(4,1)\rangle \odot (11,10,10) \bmod p = 92,53$$
$$\mathcal{S}_2 : \langle (3,4),(4,2),(5,2)\rangle \odot (21,20,20) \bmod p = 243,164$$
$$\mathcal{S}_3 : \langle (4,5),(5,3),(6,3)\rangle \odot (31,30,30) \bmod p = 454,335$$

STEP 5A: ***Client:*** `Lisa` interpolates: $\{(1,92),(2,243),(3,454)\}$, $\{(1,53),(2,164),(3,335)\}$ and obtains the secret 1 and 2, which correspond to the file-ids in row one of the inverted list.

Table 11: Example of Phase 2.

position of the single zero in *vecR1* of STEP 3A. The client generates SSS of this vector v, denoted by $\mathbb{M}(v)$, and sends them to $\mathcal{S}_{z\in\{1,2,3\}}$.

STEP 4: ***Server:*** has three objectives: (*i*) obliviously verifying the *correctness* of the client's vector, (*ii*) obliviously checking the client's access rights for the keyword, and (*iii*) obliviously returning the file-ids associated with the keyword, if the vector is correct.

1. Correctness of the client's vector. To achieve the first objective, Tests A and B of §6.5 are executed and *prevents malicious clients from generating a wrong type of vector* v, *e.g.*, $v=\{0,0,\ldots,0,0\}$, $v=\{1,0,\ldots,1,0\}$, or $v=\{0,0,\ldots,10,-9\}$.

2. Ensuring allowed access to the client for the keyword. After that, for the second objective, servers obliviously check the client's access to the keyword via the following Test 1:

$$\text{Test 1: } \mathbb{M}(test_1) \leftarrow \mathbb{M}(AC)\odot\mathbb{M}(v)$$

Test 1 ensures the client has access rights for searching the keyword. $\mathcal{S}_z$ performs a dot product between the received vector $\mathbb{M}(v)$ and the capability list of the client. If the *vector* $\mathbb{M}(v)$ *is correct and the client has search access to the keyword, then* $\mathbb{M}(test_1)$=$\mathbb{M}(0)$; otherwise, a random number, which corresponds to the no access right value. To obtain the values of $\mathbb{M}(test_1)$ in cleartext, $\mathcal{S}_z$ sends the output of the test, which is in share form, to other servers. Then, each server interpolates the values. Note that at this step, the malicious client/server will learn the value corresponding to the no access right. However, *this does not enable the malicious client to learn sw, due to Theorem 3*. Of course, we can also hide this value too, if it is required using the method given in Appendix G in [26].

3. Returning file-ids. If the interpolated value is $\langle 0\rangle$[9] for Test 1, then, $\mathcal{S}_z$ performs a dot product between the vector $\mathbb{M}(v)$ and the inverted list, and *this results in all file-ids, denoted by* $\mathbb{M}(fid\mathcal{S}_z)[]$, *that are associated with the keyword uw.*[10] $\mathbb{M}(fid\mathcal{S}_z)[]$ is sent to the client.

STEP 5A: ***Client:*** interpolates the file-ids received from servers and learns a set of file-ids, say $fid[]$, associated with the query keyword.

6.3.2 *Example of Phase 2:* is given in Table 11 .

6.3.3 *Correctness.*
The approach works at servers since an i^{th} position of the vector v having one indicates that i^{th} keyword of AC matrix has the search permission for the client. The formulation of Test 1 ensures the client has the search right to the i^{th} keyword; otherwise, $\mathbb{M}(test_1)\neq\mathbb{M}(0)$. Thus, STEP 4 produces only file-ids associated with the i^{th} keyword to which search permission is allowed.

6.3.4 *Security Discussion.*
Servers: (*i*) Servers receive a vector of SSS form. Also, the output of Test 1 contains 0 if the vector is correct, regardless of the keyword or its position in AC matrix (*e.g.*, an i^{th} keyword in which the client is interested). Thus, servers cannot learn the keyword, its position, and/or the query keyword in STEP 4. (*ii*) Since servers communicate with others to exchange the output of the test, even in the presence of a minority of malicious servers colluding with the client, the test cannot fail. This prevents the client from fetching file-ids that are not associated with the keyword to which the client has no access. The reason is that malicious entities cannot generate the vector v and shares of the test's output; thereby, interpolated values result in zero at non-malicious servers unless malicious entities know the keywords in AC matrix and their random numbers for non-access. (*iii*) Servers perform identical operations on the inverted list, hiding access-patterns from servers. Servers always return the maximum number of file-ids in which a keyword can appear regardless of the query keyword; thus, volume is also hidden from servers.
Clients: (*i*) Firstly, Tests A and B verify that client generated a legal vector (consisting of all zeros except a single one). But clients may generate a wrong vector to know file-ids, which are associated with a keyword to which search access is disallowed. In this case, Test 1 will fail only if the client possesses knowledge of the index of the keyword in AC matrix and their random number for non-access. However, a client cannot have such information. (*ii*) Clients learn the maximum number of file-ids in which a keyword can appear.

6.3.5 *Cost Analysis.*
Communication cost from a client to a server is $O(\beta)$, and from a server to the client is $O(\gamma)$, where β is the number of searchable keywords and γ is the maximum number of files associated with a keyword. Communication cost among servers involves only a few numbers. Computation cost at a server is $O(\beta\gamma)$ and at a client is $O(\gamma)$.

[9] If one of the three servers does not perform computation correctly, then Test 1 and the subsequent Tests of Phase 3 will fail. This malicious behavior can be detected by non-malicious servers by involving the fourth server and executing the computation (*i.e.*, the tests) at all four servers. Suppose $\mathcal{S}_1$ is non-malicious and $\mathcal{S}_3$ is malicious. To learn the output of the tests, $\mathcal{S}_1$ performs interpolation over values of $\langle\mathcal{S}_1,\mathcal{S}_2,\mathcal{S}_3\rangle$, $\langle\mathcal{S}_1,\mathcal{S}_2,\mathcal{S}_4\rangle$, and $\langle\mathcal{S}_1,\mathcal{S}_3,\mathcal{S}_4\rangle$. Now, all these interpolated values will not produce identical results, showing malicious behavior by one of the servers, (and non-malicious servers may terminate the protocol).

[10] This keyword could be uw, used in Step 1 or a keyword to which the client has search access. We can also make sure that the file-ids are only those associated with the keyword uw used in Phase 1, by adding the following test: $\mathbb{M}(test) \leftarrow (\mathbb{M}(sw)\odot\mathbb{M}(v))-\mathbb{M}(uw)$, and $\mathbb{M}(test)$=0 ensures this.

6.4 Phase 3: Retrieving Documents/Files

High-level idea. Phase 3 allows a client to retrieve the files corresponding to the keyword queried in Phase 1, based on the file-ids they learned in Phase 2. If a file contains even a single keyword for which the client lacks search access, that file is *not* returned to the client, and in this case, servers will obviously return a fake file.

For example, Lisa, who searched for the keyword 'Are' during Phase 1, learned that files 1 and 2 both match the keyword 'Are' after Phase 2. In Phase 3, the server returns to her the actual file 1, as well as a fake file in place of file 2. Although both files contain the keyword 'Are,' file 2 also contains the keyword 'Ana,' to which Lisa does not have search access (see Tables 3,4,5). This reveals to Lisa that the keyword 'Are' appears in two files, but she is only authorized to access one of them. Importantly, she does not learn which specific unauthorized keyword prevents access to file 2, thereby preserving the confidentiality of other keywords.

A client creates a new vector of length δ (where δ is the number of files) containing all zeros and only one at the position corresponding to one of the file-ids obtained in Phase 2 and sends this vector in share form to the servers. Servers operate over the file data structure (Table 5) and, first, obliviously verify the ***vector's correctness***, like Phase 2. If the vector is correct, servers obliviously ensure that the file does not have a keyword to which the client does not have search access and then obliviously send the desired file to the client.

Algorithm design objective. An algorithm in Phase 3 needs to:
(*i*) *Prevent sending an unauthorized file to a client.* A client must not obtain a file F that contains a keyword k that the client has searched in Phase 1 for which the client has allowed search access, and the file F also contains a keyword k' that the client is disallowed to search.
(*ii*) *Prevent information leakages.* The protocol must guarantee that neither the client nor the server learns additional information during query execution. Particularly, the server must not learn the queried keyword from Phase 1, the requested row-id of the inverted list, and whether any specific file is not returned in Phase 3 due to access restrictions. Similarly, the client must not infer the existence of keywords for which it lacks access, nor determine which specific keyword within a file has led to access denial.

6.4.1 ***Algorithms in Phase Three:*** work as follows:

STEP 5B: ***Client:*** creates $x \leq \gamma$ (a keyword appears in at most γ files) vectors v_x, each of size δ, to fetch x files containing the keyword uw. Recall that the client learns x file-ids, $fid[]$, in STEP 5A. Each such vector contains zeros, except one at the i^{th} file-id position, based on $fid[]$. Client creates SSS of such vectors ($\mathbb{M}(v_x)$) and sends them to $\mathcal{S}_{z\in\{1,2,3\}}$. Note that if a keyword appears in $x<\gamma$ files, then the $\gamma-x$ vectors will fetch a fake file with zeros. Below, *for simplicity, we consider a case, when the client fetches only a single file by sending a vector $\mathbb{M}(v)$ of length δ to $\mathcal{S}_{z\in\{1,2,3\}}$.*

Ensuring Correctness of the Vector.

STEP 6: ***Server:*** receive $\mathbb{M}(v)$ vector from the client and has three objectives to ensure: (*i*) $\mathbb{M}(v)$ contains all zero and except a single one; (*ii*) $\mathbb{M}(v)$ contains one at the position of the file-id that was sent in Phase 2 by servers; and (*iii*) the requested file does not contain a keyword to which the client has no access.

1. Ensuring $\mathbb{M}(v)$ having all zeros and a single one. Test A and Test B of §6.5 achieve the first objective.

2. Ensuring the correct position of one in $\mathbb{M}(v)$. The following Test 2 achieves the second objective.

$$\text{Test 2: } \mathbb{M}(test_2) \leftarrow (\mathbb{M}(file\_id) \odot \mathbb{M}(v)) - \mathbb{M}(fid\mathcal{S}_z)[i]$$

On the success of Tests A and B, $\mathcal{S}_z$ performs Test 2 that executes a dot product between the received vector $\mathbb{M}(v)$ and file-ids $\mathbb{M}(file\_id)$ (see the first column of Table 5). This produces the i^{th} file-id that is subtracted from one of the file-ids $\mathbb{M}(fid\mathcal{S}_z)[i]$, sent by servers in STEP 4 of Phase 2. The client informs the file-ids used for subtraction. *If the $\mathbb{M}(v)$ is correct, $\mathbb{M}(test_2) = \mathbb{M}(0)$.* $\mathcal{S}_z$ communicates with other servers to receive their shares of this computation and performs interpolation to know the value of $test_2$ in cleartext.

3. Ensuring the file excludes keywords restricted to the client. Test 3 and subsequent STEPS 7-9 achieve the third objective.

$$\text{Test 3: } \mathbb{M}(test_3)[] \leftarrow \mathbb{M}(v) \odot \mathbb{M}(AP)$$

Test 3, on the success of Test 2, performs a dot product between $\mathbb{M}(v)$ and the AP list (denoted by $\mathbb{M}(AP)$; see the second column of Table 5). This results in the position of the keywords appearing in the file and their hash digest — all in SSS form of degree two. $\mathcal{S}_z$ sends all the positions to the client, while keeping the hash digest (denoted by $\mathbb{M}(\mathrm{H}(AP))$, *i.e.*, the last value of AP list) at their end.

Return the File.

STEP 7: ***Client:*** interpolates the received values to learn the position of keywords in the file—*the client does not learn the keyword.* An honest client wishes to enable the server to know the access right for all these keywords by creating a keyword position vector, say kpv, filled with zeros except for the position of the keywords returned in STEP 6 to be one. The client creates shares of kpv, denoted by $\mathbb{M}(kpv)$, and sends them to servers. We will argue how servers will detect malicious behavior of the client in creating $\mathbb{M}(kpv)$.

STEP 8: ***Server:*** (*i*) ensures $\mathbb{M}(kpv)$ contains only zeros and ones using Test C of §6.5; (*ii*) ensures $\mathbb{M}(kpv)$ is created for positions of keywords sent in STEP 6; (*iii*) returns the file.

$$\text{Test 4: } \mathbb{M}(test_4) \leftarrow (\mathbb{M}(kpv) \odot \mathbb{M}(\mathrm{H}(ACT\_pos))) - \mathbb{M}(\mathrm{H}(AP))$$

Test 4 verifies the second condition. Here, $\mathbb{M}(\mathrm{H}(ACT\_pos))$ denotes the hash digest for the positions in AC matrix (see the blue-colored third row of Table 3) and $\mathbb{M}(\mathrm{H}(AP))$ is the hash digest, computed in STEP 6. *If $\mathbb{M}(kpv)$ is correct, Test 4 produces zero.*

Now, servers will send the file. Note that the *objective is to ensure that the* ***file return operation is oblivious***, *i.e.*, the server cannot determine whether the file that the client is accessing contains a keyword to which access is allowed or not. The server always returns a file—either a real or a garbage file—without distinguishing between the two. Further, if a file contains even a single keyword for which the client does not have access rights, the client must not be able to learn the file's content, even if the file is retrieved. To do so, $\mathcal{S}_z$ performs the following:

$$\text{Test 5: } \mathbb{M}(test_5) \leftarrow \mathbb{M}(kpv) \odot \mathbb{M}(AC)$$

$\mathcal{S}_z$ performs a dot product between $\mathbb{M}(kpv)$ and the access capability list $\mathbb{M}(AC)$ of the client (see yellow-colored rows of Table 3). It is important to note that *zero in SSS form, being the result of Test 5,*

indicates that the client has access to all keywords, appearing in the file. Otherwise, Test 5 will produce a secret-shared random number equal to the sum of the non-access right value, as in Test 1.

Finally, $\mathcal{S}_z$ selects $\mathbb{M}(\mathsf{RN}_z)$ equals to the size of a file, multiplies $\mathbb{M}(\mathsf{RN}_z)$ with the output of Test 5, and adds this to the file before sending it to the client. Before multiplication, servers reduce the degree of the polynomial of the output of Test 5 from two to one.[11]

STEP 9: ***Client:*** interpolates the file content, obtained from servers.

6.4.2 ***Correctness.*** Obviously, STEP 6 verifies if the client generates the correct vectors to fetch the i^{th} file, and its correctness is similar to what we analyzed in §6.3.3. An important aspect to discuss is how the algorithm can detect malicious behavior of the client in STEP 8A. The reason is: a dot product between a wrong vector, created by the client in STEP 7, with the respective vectors of hash digest ($\mathbb{M}(\mathsf{H}(ACT\_pos))$), and then the given subtraction will not result in a value of zero of Test 4 and Test 5.

6.4.3 ***Security Discussion.*** ***Servers:*** (*i*) Servers receive vectors in SSS form; thus cannot learn the file-id and the number of real files requested by the client. (*ii*) Servers execute an identical operation to check the vector's correctness and to send files to the client. Thus, servers cannot learn anything based on access-patterns. (*iii*) The algorithm is designed to hide the volume. If $x<\gamma$, then $\gamma-x$ files containing $\mathbb{M}(0)$ will be sent to the client; thus, hiding the volume from servers. While this comes with additional computational cost at servers and communication cost, this does not increase the space overhead, since DBO adds a single dummy file; see Table 5. Servers cannot learn the number of times they send the dummy file, since access-patterns are hidden from servers. Also, this does not reveal to the client any other file, which does not contain the queried keyword. ***Client:*** only receives the desired file and cannot fetch files, containing at least a keyword to which access is disallowed. STEP *7 reveals the number of keywords and their positions in AC matrix.*[12]

6.4.4 ***Cost Analysis.*** Communication cost from a client to a server is $O(\gamma\delta)$, and from a server to the client is $O(\gamma\eta)$, where η is the size of a single file. Communication cost among servers is $O(\delta)$. Computation cost at a server is $O(\gamma\eta\delta)$ and at the client is $O(\gamma\eta)$.[13]

6.5 The Tests A, B, and C

During different phases of the query processing, the client sends a bit vector in shared form to the servers. However, a malicious client may create incorrect vectors by either placing one at the wrong places or having non-binary values. The servers' objective is to ensure that the vector v is valid, *i.e.*, the vector contains (*i*) all zeroes and a single one, or (*ii*) many zeros and many ones only. To do so, $\mathcal{S}_z$ performs the following tests, which we have used in Phases 1-3:

$$\text{Test A: } \mathbb{M}(test_A) \leftarrow \sum \mathbb{M}(v)$$
$$\text{Test B: } \mathbb{M}(test_B) \leftarrow \sum \mathbb{M}(v^2)$$
$$\text{Test C: } \mathbb{M}(test_C) \leftarrow \mathbb{M}(v_i^2) - \mathbb{M}(v_i);\ \forall i \in \{1, |v|\}$$

[11] Servers can reduce the degree using an existing method [29] or our method of Appendix G in [26].
[12] Appendix A.1 provides a way to hide from a client number of keywords, their positions in AC matrix.
[13] Appendix A.2 extends the above method to fetch multiple files with less communication cost.

Test A and Test B achieve the first objective. These tests were executed in STEP 4 of §6.3 and STEP 6 of §6.4. In Test A, $\mathcal{S}_z$ adds the values of the vector. In Test B, $\mathcal{S}_z$ multiplies the value itself, and then adds all the values. If the client has created a correct vector, Test A and Test B produce one. Test C achieves the second objective and is executed in STEP 8A of §6.4 to prevent a malicious client from accessing files with keywords to which the client has no access. In Test C, $\mathcal{S}_z$ computes values-wise subtraction between the square of the value and the value itself. If the client has created a correct vector, Test C produces a vector with all zeroes. $\mathcal{S}_z$ learns the results of these tests in cleartext by exchanging shares with other servers.

7 OPTIMIZATION OF THE INVERTED INDEX

Objectives and high-level idea. The method of §6.3 has the space and computational overheads, due to the size of the inverted list, in which each entry is padded to the same maximum size to make them identical in size (see §5). For example, if a keyword k_1 appears in 1000 files, while all the remaining keywords appear only in a few files, then fake file-ids are added to all the remaining keywords, thereby each row of the inverted list has 1000 file-ids, and this huge size inverted list results in the overhead. To overcome the overhead, below, we develop an algorithm based on two new data structures and explain the high-level idea of the algorithm.

New data structures. We create two arrays:

(1) **AddrList** (Table 12). Each entry in AddrList contains the starting index position (SiP) of the file-ids associated with the keyword in the second array, the count (CuT) for the files-ids having the keyword, the hash digest (HD) of the row, and the hash digest (HdV) that is the sum of the hash digest of p positions of the file-ids associated with the keyword (where SiP$<p<$SiP + CuT) and is used by servers for client's query verification. We use two colors for differentiating HD and HdV. AddrList stores keywords according to the positions in AC matrix; see the second row of Table 3.

(2) **OptInv** (Table 13). OptInv is a single-dimensional array and stores the file-ids associated with the keyword and the hash digest. We allocate all file-ids associated with a keyword in adjacent slots and hash digests over the file-ids in a slot after the last file-ids. To handle new documents, OptInv has some empty slots, filled with zeros or random numbers) for each keyword (see "empty' slots in Table 13). The decision on the number of empty/fake slots depends on DBO or the frequency of the keywords. Of course, these empty/fake slots will be occupied as new files are inserted. Appendix B.1 develops a method to extend the OptInv when it has no free slots.

Outsourcing. DBO outsources AddrList and OptInv using SSS.

Example. AddrList and OptInv are created for cleartext files of Table 2 and AC matrix of Table 3. Gray parts of Tables 12 and 13 is written for the purpose of explanation and is not outsourced.

Keywords	SiP	CuT	HD	HdV
are	1	3	$H_1 = \mathsf{H}(1,3,\text{are})$	$h_1 = \mathsf{H}(1) + \mathsf{H}(2) + \mathsf{H}(3)$
ana	4	3	$H_2 = \mathsf{H}(4,3,\text{ana})$	$h_2 = \mathsf{H}(4) + \mathsf{H}(5) + \mathsf{H}(6)$
how	9	2	$H_3 = \mathsf{H}(9,2,\text{how})$	$h_3 = \mathsf{H}(9) + \mathsf{H}(10)$

Table 12: Cleartext AddrList array.

1	2	3	4	5	6	7	8	9	10
$f1$	$f2$	hd_1	$f2$	$f3$	hd_2	empty	empty	$f1$	hd_3

Table 13: Cleartext OptInv array. **Notations.** $hd_1 = \mathrm{H}(f2, (\mathrm{H}(f1, \mathrm{H}(\mathrm{are}))))$, $hd_2 = \mathrm{H}(f3, \mathrm{H}(f2, \mathrm{H}(\mathrm{ana})))$, $hd_3 = \mathrm{H}(f1, \mathrm{H}(\mathrm{how}))$

7.1 Details of Query Execution

Objective. This section explains query execution on AddrList (Table 12) and OptInv (Table 13). Let γ be the maximum number of file-ids associated with a keyword, and let n be the size of OptInv. γ and n are known to each entity in DOC*. The client's objective is to fetch the desired file-ids from OptInv for the keyword they have searched in Phase 1, without revealing access-patterns and volume to servers. While returning γ file-ids hides volume from servers, the goal of DOC* is to allow the client to learn only the file-ids associated with the keyword; nothing else. Also, DOC* must allow the server to, first, verify that the client is only fetching the file-ids associated with the keyword to which they have search permission.

High-level idea. Query execution. Phase 1 (STEP 1-STEP 3A) is executed on AC matrix without any modification. Then, Phase 2 is executed with the new steps (developed below) on AddrList and OptInv. After executing Phase 1, the client learns which row of AddrList needs to be fetched. Then, STEP 3B - STEP 5A of §6.3.1 are executed, resulting in SiP, CuT, and HD of the desired row of AddrList. Note that the servers keep HdV value of the returned row, for verification at a later stage. Next, the client executes the following STEP A - STEP C, to learn the file-ids from OptInv.

STEP A: *Client — transmitting two vectors*. Clients want to fetch γ file-ids, regardless of the number of files associated with a keyword. To do so, the client creates two vectors: *row_vec* and *pos_vec*.

row_vec: The client interprets OptInv as a $x \times y = n$ matrix and computes the row number of the matrix that will contain file-ids, starting from SiP to SiP+CuT for the keyword. Based on the row number, the client creates a *row vector*, denoted by *row_vec*. We assume $y \geq \gamma$; thus, up to γ file-ids associated with a keyword may span at most two rows in the matrix. For simplicity, we assume all γ file-ids to be fetched appear in a single, i^{th}, row of the matrix, and thus, *row_vec* contains x elements, with $x - 1$ zeros and one at the i^{th} position.

pos_vec: The client constructs a position vector (denoted *pos_vec*) of length y, where all elements are ones except for $\Delta \leq \gamma$ zeros at the specified positions from SiP to SiP+CuT (where a keyword may be associated with $\Delta \leq \gamma$ file-ids). Shares of *row_vec* and *pos_vec* are then created by the client and sent to servers.

STEP B: *Servers — returning y file-ids*. The objective of servers is to send only the desired Δ file-ids to the client. To do so, servers: (*i*) obliviously selects the desired row of the matrix — by organizing OptInv in the form of $x \times y$ matrix, multiply all values of the i^{th} row of the matrix by $row\_vec[i]$, and finally, adds all the values of each column, resulting in $y \geq \gamma$ file-ids, *i.e.*,

$\mathbb{M}(fids)[i] \leftarrow \sum_{1 \leq j \leq y} (\mathbb{M}(\mathsf{OptInv})[i][j] \times \mathbb{M}(row\_vec)[i])$

(*ii*) To prevent sending extra file-ids, obliviously remove the file-ids not associated with the keyword by position-wise multiplying $\mathbb{M}(\mathsf{RN})[i]$ with *pos_vec*, and then, position-wise adding the output to $\mathbb{M}(fids)[]$ file-ids, $i \in \{1, y\}$, *i.e.*,

$\mathbb{M}(final\_fids)[i] \leftarrow \mathbb{M}(fids)[i] + (\mathbb{M}(\mathsf{RN})[i][i] \times \mathbb{M}(pos\_vec)[i])$

STEP C: *Client — interpolation for knowing file-ids*. The client interpolates $\mathbb{M}(final\_fids)$. Since servers have added random numbers to non-desired file-ids (*i.e.*, $\mathbb{M}(\mathsf{RN})[i] \times \mathbb{M}(pos\_vec)[i]$, as $pos\_vec[i] = 0$ for the desired position; otherwise one), the client will not learn any additional non-desired file-ids.

7.1.1 *Example of Query Example*.

We continue from the example of Table 11. In STEP 3B, the client `Lisa` creates a vector $\langle 1, 0, 0 \rangle$ and sends it to three servers in share form. *For the purpose of understanding, below, we do not show operations over shares.*

STEP 4: *Servers:* perform a dot product of the received vector $\langle \mathbb{M}(1), \mathbb{M}(0), \mathbb{M}(0) \rangle$ and AddrList (Table 12) and returns $\langle \mathbb{M}(1), \mathbb{M}(3), \mathbb{M}(H_1) \rangle$ to the client. Servers ***keep*** $\mathbb{M}(h_1)$ ***at their ends***. All such shares are of degree two

STEP A: *Client:* interpolates the values and learns SiP to be 1 and CuT to be 3. Also, H_1 verifies the correctness of SiP and CuT. Client learns based on SiP and CuT that they need to fetch index 1, 2, and 3 from OptInv. By interpreting OptInv as a 3×4 matrix, indexes 1-3 will appear in the first row. Thus, the client creates $row\_vector = [1, 0, 0]$, $pos\_vec = [0, 0, 0, 1]$, and outsources them to three servers in the share form.

STEP B: *Servers:* organize OptInv in a 3×4 matrix (see Table 14) and compute the following:

$\mathbb{M}(f_1)$	$\mathbb{M}(f_2)$	$\mathbb{M}(hd_1)$	$\mathbb{M}(f_2)$	$\mathbb{M}(1)$
$\mathbb{M}(f_3)$	$\mathbb{M}(hd_2)$	$\mathbb{M}(empty)$	$\mathbb{M}(empty)$	$\mathbb{M}(0)$
$\mathbb{M}(f_1)$	$\mathbb{M}(hd_3))$	$\mathbb{M}(empty)$	$\mathbb{M}(empty)$	$\mathbb{M}(0)$
$\mathbb{M}(f_1)$	$\mathbb{M}(f_2)$	$\mathbb{M}(hd_1)$	$\mathbb{M}(f_2)$	

$\odot$

Table 14: Left most: OptInv. Right most: a row vector in cyan. Bottom part is the output $(\sum_{j=0}^{j=y} (\mathbb{M}(\mathsf{OptInv})[i, j] \times \mathbb{M}(row\_vec)[i]))$ **in yellow.**

Assume $\mathbb{M}(\mathsf{RN})[i] = \langle \mathbb{M}(r_1^z), \mathbb{M}(r_2^z), \mathbb{M}(r_3^z), \mathbb{M}(r_4^z) \rangle$. Servers do the following computation and send their output to the client:

$\mathbb{M}(r_1^z) \times \mathbb{M}(0) + \mathbb{M}(f_1) = \mathbb{M}(f_1)$, $\mathbb{M}(r_2^z) \times \mathbb{M}(0) + \mathbb{M}(f_2) = \mathbb{M}(f_2)$

$\mathbb{M}(r_3^z) \times \mathbb{M}(0) + \mathbb{M}(hd_1) = \mathbb{M}(hd_1)$,[14] $\mathbb{M}(r_4^z) \times \mathbb{M}(1) + \mathbb{M}(f_2) = \mathbb{M}(Random)$

STEP C: *Client:* interpolates and learns the desired file-ids: f_1, f_2, and their hash digests, which is used to verify the correctness of the file-ids. The random number will not provide any information about the last file-id, (though it is the second desired file-id f_2).

7.2 Server Size Verification

We have shown how the client can fetch γ file-ids, without incurring space overhead in maintaining the optimized inverted list in §7. Below, we develop a method for servers to verify *row_vec* and *pos_vec* used to fetch data from OptInv.

Objectives and high-level idea. The client may try to retrieve random file-ids and/or more than the desired file ids by sending *incorrect row_vec and pos_vec* (both in SSS form). For instance, in the example of §7.1.1, Table 10, Lisa may create $row\_vec = [0, 1, 0]$ and $pos\_vec = [0, 0, 0, 0]$ to learn the entire second row of the matrix;

[14] We multiply very large random numbers also to hide the location of hash digests, if the returning row contains several hash digests corresponding to other keywords/file-ids.

see Table 14. Thus, the servers need to verify such vectors against the SiP and CuT values, which were sent in STEP 4 (see §7) before sending file-ids to the client. To do so, the servers compute hash digest on some numbers and compare the hash digest value of the HdV column of AddrList, particularly, the HdV value, which was obtained in STEP 4 (see §7.1) and has not been sent to the client. The following steps will be executed before STEP B of §7.1, *i.e.*, before sending the file-ids to the clients.

7.2.1 ***Details of the Method.*** Servers perform the following:

(1) Subtract the *pos_vec* from a vector containing y one in cleartext, *i.e.*, $\langle 1_1, \ldots, 1_y \rangle$. Note that the output will be in SSS form of degree one.
(2) Compute hash digest over numbers 1 to n (here, the hash digest will be in cleartext) and then, organize such hash digests of 1 to n numbers into a $x \times y$ matrix.
(3) Multiply the i^{th} value of $\mathbb{M}(row\_vec)$ to all the values of the i^{th} row of the matrix of hash digests and compute the sum over each of the y columns. This results in a single row of the matrix in SSS form of degree one, since *row_vec* is in SSS form of degree one.
(4) Perform a dot product of the output of the step (1) and the output of step (3), and this results in a value, say $\mathbb{M}(val)$, in SSS form of polynomial degree two.
(5) Subtract $\mathbb{M}(val)$ with HdV value computed in STEP 4 (see §7.1.1), and this results in a value, say $\mathbb{M}(op)$, which is sent to all other servers. Note that *if the client has created the correct row_vec and pos_vec, then* $\mathbb{M}(op) = \mathbb{M}(0)$.
(6) Interpolate the values, after receiving shares from the other two servers. If the output value is zero, then servers send the output of STEP B of §7.1 to the client.

7.2.2 ***Example of Server-side Verification.*** Suppose, the client learns SiP to be 1 and CuT to be 3 (see STEP 4 of §7.1.1), but creates wrong $row\_vec = [0, 1, 0]$ and correct $pos\_vec = [0, 0, 0, 1]$ in SSS form. Below, we show with the help of an example, how servers can learn about the wrong $\mathbb{M}(row\_vec)$, without knowing it in cleartext. For the purpose of simplicity and understanding, we explain operations over cleartext.

Servers subtract $\mathbb{M}(pos\_vec)$=$\mathbb{M}(0, 0, 0, 1)$ from y ones, as: $1111 - \mathbb{M}(0)\mathbb{M}(0)\mathbb{M}(0)\mathbb{M}(1) = \mathbb{M}(1)\mathbb{M}(1)\mathbb{M}(1)\mathbb{M}(0)$. Next, servers compute hash digests on 1 to 10 numbers, organize them in a matrix form, and multiply $\mathbb{M}(row\_vec)$; and this results in $\langle \mathbb{M}(H(5)), \mathbb{M}(H(6)), \mathbb{M}(H(7)), \mathbb{M}(H(8)) \rangle$ (see Table 15). Then, servers perform a dot product between $\langle \mathbb{M}(H(5)), \mathbb{M}(H(6)), \mathbb{M}(H(7)), \mathbb{M}(H(8)) \rangle$ and $\langle \mathbb{M}(1), \mathbb{M}(1), \mathbb{M}(1), \mathbb{M}(0) \rangle$, and this produces $\mathbb{M}(val)$=$\mathbb{M}(H(5))$+$\mathbb{M}(H(6))$+$\mathbb{M}(H(7))$.

H(1)	H(2)	H(3)	H(4)
H(5)	H(6)	H(7)	H(8)
H(9)	H(10)		
$\mathbb{M}(H(5))$	$\mathbb{M}(H(6))$	$\mathbb{M}(H(7))$	$\mathbb{M}(H(8))$

$\odot$

$\mathbb{M}(0)$
$\mathbb{M}(1)$
$\mathbb{M}(0)$

Table 15: Left most: Hash digest of 10 numbers. Right most: a wrong row vector in cyan. Bottom part is the output in yellow.

Then, servers subtract $\mathbb{M}(val)$ from the output of STEP 4 that is $\mathbb{M}(h_1)$, which equals to $\mathbb{M}(H(1)) + \mathbb{M}(H(2)) + \mathbb{M}(H(3))$ (see Table 12). The output of the subtraction operation is sent to all servers, and after interpolation, the final output is not zero. This shows that the client has not created the vectors correctly.[15]

7.2.3 ***Discussion on Security.*** The basic idea of above this verification method is comparing the corresponding hash digest h in Table 12 with the calculated value h' from Table 15. If the client creates wrong vectors, the server will obtain the wrong hash addition, which is not equal to the fixed one in Table 12. The whole procedure is executed in SSS form. That is to say, servers cannot learn any extra information except the verification result (*i.e.*, 0 or a random number). Meanwhile, the client's wrong action will also be detected.

8 EXPERIMENTAL EVALUATION

In this section, we evaluate the following:

(1) The size of secret-shared data produced by DOC* — Exp 1.
(2) The end-to-end processing time for a query in DOC* — Exp 2.
(3) Time on different sizes of data to show scalability of DOC* — Exp 3.
(4) Time taken by different implementations of inverted list — Exp 4.
(5) Time taken by verification algorithms at the server and the client— Exp 5.
(6) Performance of DOC* in a wide-area network — Exp 6.
(7) Time taken by DOC* and other systems — Exp 7.

Machines. We selected four `c7a.32xlarge` AWS machines, each with 128 cores and 256 GB RAM, playing the role of three servers. The client machine was with 32 cores and 64 GB RAM. These machines were located in different zones (which are connected over wide-area networks), of AWS Virginia region.

Data. We use Enron dataset [39], a commonly used real data to evaluate techniques for document stores. This *dataset was also used in other recent document store work [6, 7, 40]*. For all experiments, except Exp 3 for scalability, we created a dataset, containing 5K searchable keywords in 500K files. The code is written in Java and contains more than 5000 lines. Time is calculated by taking the average of 20 program runs and shown in milliseconds (ms).

Client-side storage. The maximum amount of data that a client stores is the number of file-ids to be retrieved from servers during Phase 3. In the worst case, if a keyword appears in all files, the space overhead equals the size of all file-ids. Since we selected 500K files, storing 500K file-ids requires a client to have 2MB of space.

8.1 Evaluation of DOC*

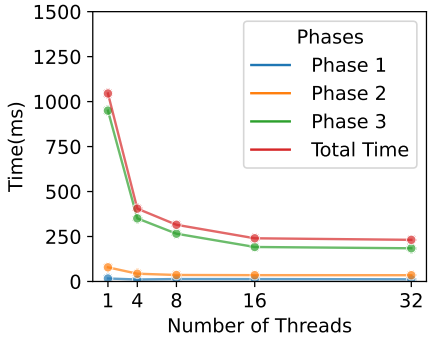
Exp 1: Share generation. We create shares of AC matrix, inverted lists, and files in the non-optimized version (§6.3) and of AC matrix, AddrList, OptInv, and files in the optimized version (§7).

Table 16 shows the size of these data structures in SSS form at a single server for the dataset, containing 5K keywords in 500K files. Keywords appear on average in 38 files, and the median was 23. Skewness (*i.e.*, the difference between the maximum and minimum number of files associated with a keyword) results in a large size of the inverted list (≈1.6GB), due to padding each entry of the inverted list with fake values to have them 110K elements, as a keyword appears in at most 110K files. AC matrix contains access rights for 5K keywords. We created 4,096 clients, like existing work [6, 7].

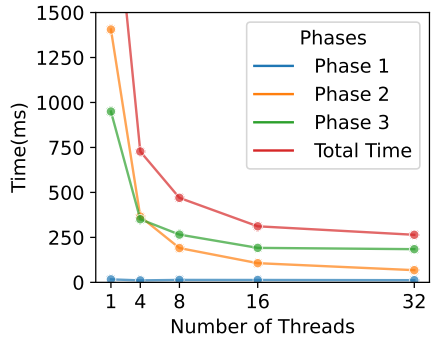
[15] Recall that all these arithmetic operations are performed in $\mathbb{Z}_p$, the equation $h' = h$ is in fact $h' \bmod p = h \bmod p$. If p is chosen large enough, the error probability can be ignored.

	AC matrix	Phase 2 data	Files
Non-optimized, *i.e.*, **inverted list-based DOC***	618.7 MB	1.6 GB	3.5 GB
Optimized, *i.e.*, **OptInv & AddrList-based DOC***	618.7 MB	139.6 MB	3.5 GB

Table 16: Exp 1: Size of the shares at a single server.



(a) Optimized implementation using OptInv and AddrList.



(b) Non-optimized implementation using an inverted list.

Figure 2: Exp 2: DOC* end-to-end processing time.

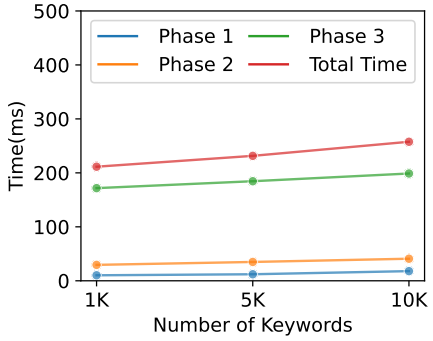
Entity	Phase 1	Phase 2	Phase 3	Total
Client	3	8.7	32.9	44.6
Server	7.3	9.9	67.9	85.1
Network	1.8	16.3	83.7	101.8
Total (milliseconds)	**12.1**	**34.9**	**184.5**	**231.5**
Variation	±2.18	±2.5	±3.42	±8.1

Table 17: Exp 2: Time on 5K keywords, 500K files, 32 threads.

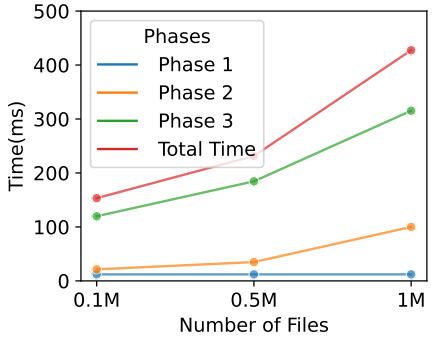
Exp 2: DOC* end-to-end processing time. We implemented multi-threaded programs for all three phases of DOC*. These programs partition the data into multiple blocks of equal size, and each block is processed by a separate thread. Figure 2 shows that as increasing the number of threads from 1 to 32, the processing time decreases for each phase. We implemented both versions of Phase 2, *i.e.*, OptInv (Figure 2a) and inverted list (Figure 2b).

OptInv-based DOC*, for all three phases, took ≈1 second using one thread, whereas took ≈231.5ms using 32 threads. OptInv performs better compared to the inverted list for 5K keywords in 500K files, since the size of OptInv is much smaller than the size of the inverted list (due to the absence of padding; as shown in Table 16). Table 17 shows a breakdown of the time taken by the client, servers, and network for each phase of DOC* with its optimized implementation that uses AddrList and OptInv. Important to note that ***our access control method is also highly efficient, taking at most 12.1ms*** (see the first column). Further, the client's processing time is less than the server's in all three phases. Exp 4 will show a situation when the inverted list will work better compared to OptInv. Last row of Table 17 shows the variation of time in different phases.

Exp 3: Scalability of DOC*. To evaluate the scalability of DOC*, we created two types of datasets: (*i*) varying the number of keywords from 1K, 5K, 10K in 500K files, and (*ii*) varying the number of files from 100K, 500K, and 1M with a fixed number of keywords to 5K. We run this and all the following experiments using 32 threads, since Exp 2 justifies the best performance with 32 threads. Figure 3 shows that as the data increases, the entire computation time increases. Particularly, DOC* took 257.6ms for 10K keywords in 500K files (Figure 3a), and 427.4ms for 5K keywords in 1M files (Figure 3b).



(a) Varying number of keywords.



(b) Varying number of files.

Figure 3: Exp 3: Scalability test using 32 threads.

Exp 4: Different implementation of the inverted list. Phase 2 can be implemented using the inverted list, denoted by NINV below, (Table 4, §6.3) or an optimized approach using AddrList and OptInv, denoted by OINV below, (Table 12, Table 13, §7). NINV and OINV differ in the following two aspects: (*i*) NINV takes one round of communication between a server and a client, while OINV takes two rounds, and (*ii*) NINV adds fake file-ids to each keyword to

make them identical in length in inverted list, while OINV does not add fake file-ids. This experiment investigates when we can use one of the methods for Phase 2. We implemented NINV and OINV on varying numbers of maximum numbers of files associated with a keyword. We considered four cases: a keyword can appear in at most 10, 100, 1,000, 10,000, and 100,000 files — in other words, an entry of the inverted list will contain these many files, and we selected 5,000 keywords. Figure 4 shows the results of this experiment, where *x-axis is on a log scale and refers to these file numbers*.

For the case of at most 1,000 files containing the same keyword (where on average a keyword appears in 528 files), NINV works best, due to less total processing time compared to the time of OINV. Particularly, in NINV, the processing time (9.5ms) at both servers and the client and transmission time (1.3ms), while OINV took 12.1ms processing time and 2.4ms for transmission. Here, the size of the inverted list (25MB) was larger than the size of AddrList and OptInv (22MB). NINV works similar to OINV in the case of 10K files. As we increase the number of keywords to 100K, NINV does not work best, due to significantly increasing the size of the inverted list by padding fake file-ids (1.52GB in NINV vs 257MB in OINV). The total time was 59.8ms for NINV compared to 28.8ms for OINV.

These observations highlight that NINV *(inverted list-based) method is effective when the difference between the maximum and minimum number of files associated with a keyword is relatively small. In contrast,* OINV *optimized implementation performs well in scenarios, having a significant difference between the maximum and minimum number of files associated with a keyword.*

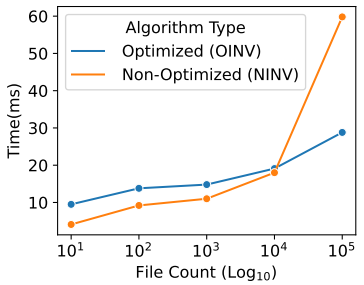


Figure 4: Exp 4: Different implementations of Phase 2.

Exp 5: Impact of verification. DOC* provides verification options for both servers and clients. Table 18 compares the processing time of DOC* with verification and without verification for 5K keywords in 500K files using 32 threads. Only Phase 2 takes more time in server-side verification for *row_vec* over OptInv, which involves multiplication over hash digests of 160 bits for each value of OptInv.

	Phase 1	Phase 2	Phase 3	Total
Small data				
Without verification	**12.1**	**34.9**	**184.5**	**231.5**
With verification	**13.1**	**90.0**	**210.1**	**313.20**

Table 18: Exp 5: Verification time (ms) using 32 threads.

Exp 6: Performance in a wide-area network. We evaluate DOC* in a WAN setting, with servers located in three different AWS regions: Ireland, London, and Paris. The client was located in Frankfurt. Before data transmission, we compressed data at the servers and the client in all three phases. Over 5K keywords and 500K files, query execution took 1123.7ms, of which computation at the client and the server took 129.7ms, compression/decompression took 381ms, and data transmission took 613ms. This network time can be reduced by using multiple sockets that enable full utilization of the bandwidth between two different data centers.

Additional experiments are given in [26]. They investigate the impact of: (*i*) different implementations of AP list in Phase 3, (*ii*) granting/ revoking access rights, (*iii*) adding new file-id(s) to an existing keyword, (*iv*) deleting an existing file, (*v*) deleting an existing keyword from AC matrix, and (*vi*) increasing the number of servers.

8.2 Comparing DOC* against Other Systems

Exp 7: Time taken by DOC* and other systems. We compare DOC* against different systems on 5K keywords in 500K files:

(1) ***A cleartext database system, MySQL***: is used to store AC matrix, AddrList, and OptInv in a tabular format, with the data stored in cleartext. We execute SQL queries using the index supported by MySQL to know the desired file-ids associated with a keyword.

(2) ***The baseline method***: is given §4.

(3) ***Secrecy [43]***: is a secret-sharing (SS)-based relational-data processing system (not a secure document storage system), and hence cannot be used to retrieve files). We store AC matrix, AddrList, OptInv in Secrecy and execute SQL queries to support Phase 1 and Phase 2 of DOC*. While we recognize that Secrecy offers complex operations, such systems are slower to support Phase 1 and Phase 2, due to their multiple rounds of communication among servers storing the shares to execute a search query — particularly, for searching keywords over a set of β keywords require $O(\ell)$ communication rounds where ℓ is the maximum length of a word, and the total amount of information flow among servers (and between a server and a querier) can be $O(\beta)$.

(4) **DOC*-**Leaky: keeps AC matrix, AddrList, OptInv, and files in SS, and leaks access-patterns/volume during query execution. (*v*) **DOC*-**Secure: refers to the secure algorithms of all three phases, developed in this paper.

(5) ***Dory [40]***: is an encryption-based secure document store, which allows knowing only file-ids associated with a keyword and hides access-patterns and volume. While Dory does not provide the same security guarantees as DOC*, we select Dory to show the impact of computationally secure techniques. Dory does not support access control and returns file-ids not associated with a query keyword, *i.e., false positive file-ids*. On the dataset of 5K keywords in 500 files, Dory returns 765 false positives on average, while the maximum, minimum, and median false positives were 27467, 0, 403, respectively. Dory takes more time due to their technique, namely distributed point function [44] for hiding access-patterns, which requires executing a pseudo-random generator (PRG) for each row of the database. Executing PRG is more time-consuming compared to simple addition and multiplication operations as in DOC*.

(6) ***Metal [6, 45] and Titanium [7]***: are recent conditionally secure document storage systems. The code of these systems is not available.

Table 19 compares these systems on different parameters. Table 20 shows the time for different systems. *For this experiment, we*

Systems	Sieve [41]	Ghoster [42]	Dory [40]	Metal [6]	Titanium [7]	DOC*
Offered security	Computational security					Unconditional security
Cryptographic techniques	Encryption	Encryption	Encryption	Encryption & binary shares	Additive shares & trusted proxy for access control (§7 of [7])	Shamir's shares
Number of servers	1	1	2	2	2	4
Access control	Attribute	File-ID	N/A	File-ID	File-ID	Keyword, Attribute, File-ID
Granularity of access	**Fine-grain**	**Coarse-grain**	N/A	**Coarse-grain**		**Fine-grain & coarse-grain**
Complexity of granting access	○	○ out-of-band	N/A	● out-of-band	○	○
Complexity of revoking access	●	●	N/A	◐	○	○
Trusted proxy for access control	**No**		N/A	**No**	**Yes**	**No**
False positives in returning answers	**No**	**No**	Yes	**No**	**No**	**No**
Malicious servers' handling	**Yes**	No	**Yes**	**No**	**Yes**	**Yes**
Malicious clients' handling	No	No	N/A	**Yes**	**Yes via trusted proxy**	**Yes**
Information leakage from ciphertext	Yes♯	Yes¶	**No**	**No**	**No**	**No**
Preventing access pattern and/or volume leakage	None	Only AP (& volume leakage does not matter as fetching files based on ids)				AP, V

Table 19: Comparing DOC* against other secure document systems. Green color indicates good aspects of the systems. Red color indicates negative aspects of the systems. Malicious servers can change data or collude with malicious clients, who want to access any file. Processing time for Dory is >2sec†), which is not included, since Dory does not offer access control and only supports finding file-ids. Time for DOC*, Metal, and Titanium includes access control checking and file retrieval time. The dataset includes 5K searchable keywords in 500K files. ‡: Sieve time for a single object of size 375KB was 0.44sec. ¶: reveals which object can be accessed by how many clients. ♯: reveals attributes and which clients can access which attributes. Complexity of operations: ●: hard, ◐: medium, ○: easy.

Systems	MySQL	Baseline	DOC*-Leaky	DOC*-Secure	Secrecy [43]	Dory [40]
Phase 1 time	0.8ms	24.23sec	16.6ms	16.6ms	7ms	NA
Phase 2 time	2.2ms	§4	4.2ms	78.7ms	>4sec	>2sec
Avg. false positives	0	0	0	0	0	765

Table 20: Exp 7: Comparing DOC* & other systems on 1 thread.

do not include the time to fetch a file, since MySQL and Dory do not support such an operation.

9 EXISTING WORK & THEIR LIMITATIONS

Access control for KV Stores. We classify works on access control for KV and KD stores into two classes: (*i*) ***Trusted proxy-based.*** BigSecret [8] and Titanium [7] support key-level access over KV and document-level access KD stores, respectively, by using a trusted proxy to check access rights. (*ii*) ***Cryptographic access control.*** Such works are either encryption-based [9, 42] that allows key-level access over KV and document-level access over KD stores, respectively, or secret-sharing with encryption-based (Metal [6]) that offers document-level access over KD stores using secret-sharing for implementing access control and encryption for storing files.
Limitations. These work deals with either a simple version of key-level access over KV store or document-level access over KD stores, inheriting all limitations of scalability and implementations, as mentioned in §1. Other limitations are shown in red color in Table 19.

Access control techniques. Access control methods, *e.g.*, query keyword-, attribute-, and role-based, have been developed [46]. Cryptographic access control methods, *e.g.*, attribute-based encryption (ABE) [47–49], quorum secret-sharing-based access (QSSAC) [50] and [51] function secret-sharing (FSS) [52], have been proposed.
Limitations. ABE is highly inefficient and harder for revocation [53]. While related, traditional ABE approaches over encrypted data are not directly applicable to our setting, due to leaking access-patterns and not handling scenarios where access must be denied based on the presence of specific keywords a client does not have permission for. Extending ABAC-based techniques to mitigate such issues and apply them in our setting is an interesting direction for future work. QSSAC [50] and [54–57] have limitations: work only over encrypted databases, need access control and database servers as two different entities, keep access rights in cleartext, and/or reveal queries to access control servers. [51] provides function secret-sharing (FSS) [52]-based access control techniques, restricting clients from executing any functions. [51] works over cleartext only. Titanium [7] uses a trusted proxy to implement access control. Furthermore, existing SS-based database systems (*e.g.*, [17, 43, 58]) do not support integrated access control.

Secure document storage. MongoDB [59, 60], GarbleCloud [61], Virtru [62], Enveil [63], and Proton [64] are industrial document stores based on variants of searchable encryption [1]. None of them supports key-based access control.

Secret-sharing (SS)-based systems. Several database systems (*e.g.*, Sharemind [58], Jana [17], Conclave [18], Secrecy [43]) uses secret-sharing (SS). None of these systems deal with access control and malicious clients. Such systems are also not efficient due to multiple communication rounds among servers during query processing; *e.g.*, Jana [17] takes ≈475s for a selection query on 1M rows. We have also discussed Secrecy [43] in §8.2.

Access-pattern hiding techniques. Techniques, *e.g.*, Path-ORAM [65], Private Information Retrieval (PIR) [66], or Distributed Point Function (DPF) [44], hide access-patterns. These techniques differ in the way they access/read the data and the amount of data they return to a querier. For instance, to fetch a single item/object, Path-ORAM accesses and returns the polylogarithmic size of data, while DPF accesses the entire data but returns only the desired answer. DPF is used in Dory [40], and variants of ORAM are used in Metal [6] and Titanium [7]. However, ORAM techniques have several problems: (*i*) The disclosure of additional data beyond the desired answer to a query by Path-ORAM poses a challenge when the client is not allowed to learn those extra answers. (*ii*) ORAMs restrict the system throughput, due to their structures, see [67]. (*iii*) When used to prevent volume leakage by sending the maximum number of files, say ℓ, associated with a keyword, ORAM sends ℓ times polylogarithmic size of data. The use of such techniques comes with query inefficiency. For example, Dory [40] takes less than 0.1 seconds for *searching file-ids (not retrieving the file)* associated with the desired keyword over 2^{20} files when *revealing access-patterns* (Figure 9 of [40]); while Dory takes ≈10s for the same operation when hiding access-patterns and volume on 2^{20} files. Titanium [7] takes ≈7.2s *to fetch a file* with access-patterns hidden on the same data. To fetch items based on an index-id of an object, PIR [66] or its several variants, *e.g.*, PIR-by keyword [68] or information-theoretically secure PIR [69], can be used. However, such techniques do not deal with malicious client and server, and also reveals additional data to the client, *e.g.*, the presence/absence of other keywords in using PIR-by-keywords.

Volume hiding techniques. [70–72] provides volume-hiding techniques. These techniques degrade the system's performance in terms of space requirements due to storing a map and a stash at the data owner, making it harder to deal with new insertions, and reveal extra information to the clients than the desired answers.

10 CONCLUSION

This paper develops an unconditionally-secure document/file storage system, DOC*, with query keyword-based access control. Operations at servers hide access-patterns and volume, and do not reveal any information to servers about the data. DOC* exhibits efficient performance and takes 231.5ms over 5K keywords in 500K files.

ACKNOWLEDGMENT

We are thankful to Murat Kantarcioglu for his feedback on the baseline method of §4. Y. Li was funded by National Natural Science Foundation of China Grant 62372107. S. Mehrotra was funded by NSF Grants 1952247, 2133391, 2032525, and 2008993. S. Sharma was funded by NSF Grant 2245374.

REFERENCES

[1] Dawn Xiaodong Song et al. Practical techniques for searches on encrypted data. In *IEEE SP*, pages 44–55, 2000.
[2] Hakan Hacigümüs et al. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.
[3] Redis Key-Value Store: https://redis.io/.
[4] Redis Access Control List: https://tinyurl.com/bdhcc7m6.
[5] Jerome H. Saltzer et al. The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308, 1975.
[6] Weikeng Chen et al. Metal: A metadata-hiding file-sharing system. In *NDSS*, 2020.
[7] Weikeng Chen et al. Titanium: A metadata-hiding file-sharing system with malicious security. In *NDSS*, 2022.
[8] Erman Pattuk et al. Bigsecret: A secure data management framework for key-value stores. In *International Conference on Cloud Computing*, pages 147–154, 2013.
[9] Xu Yuan et al. Secure multi-client data access with boolean queries in distributed key-value stores. In *CNS*, pages 1–9, 2017.
[10] Survey: Insider Threats Surge Across U.S. Critical Infrastructure. Available at: https://tinyurl.com/mry3hmy7.
[11] NSA Moves to Prevent Snowden-Like Leaks. Available at: https://tinyurl.com/3xsvnpea.
[12] MySpace lost 13 years worth of user data after botched server migration?. Available at: https://tinyurl.com/2b6vjs27.
[13] LastPass Reveals Second Attack Resulting in Breach of Encrypted Password Vaults. Available at: https://tinyurl.com/4evra455.
[14] CrowdStrike: Attackers focusing on cloud exploits, data theft. Available at: https://tinyurl.com/3hvr2nd2.
[15] Fatih Emekçi et al. Dividing secrets to secure data outsourcing. *Inf. Sci.*, 263:198–210, 2014.
[16] Jana: Private Data as a Service. Available at: tinyurl.com/47jemma5.
[17] David W. Archer et al. From keys to databases - real-world applications of secure multi-party computation. *Comput. J.*, 61(12):1749–1771, 2018.
[18] Nikolaj Volgushev et al. Conclave: secure multi-party computation on big data. In *EuroSys*, pages 3:1–3:18, 2019.
[19] Johes Bater et al. SMCQL: secure query processing for private data networks. *PVLDB*, 10(6):673–684, 2017.
[20] Ran Canetti et al. Adaptively secure multi-party computation. In Gary L. Miller, editor, *STOC*, pages 639–648, 1996.
[21] Ivan Damgård et al. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, pages 285–304, 2006.
[22] Koki Hamada et al. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *ICISC*, pages 202–216, 2012.
[23] Dan Bogdanov et al. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In *NordSec*, pages 59–74, 2014.
[24] Eleftheria Makri et al. Rabbit: Efficient comparison for secure multi-party computation. In *FC*, pages 249–270, 2021.
[25] Microsoft Seal: https://github.com/microsoft/SEAL.
[26] Code, data, and the full version of the paper: https://github.com/SecretDeB/DocStar-VLDB-2025.
[27] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
[28] Robert M Corless et al. A graduate introduction to numerical methods. *AMC*, 10:12, 2013.
[29] Michael Ben-Or et al. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.
[30] Gilad Asharov and Yehuda Lindell. A full proof of the BGW protocol for perfectly-secure multiparty computation. *IACR Cryptol. ePrint Arch.*, page 136, 2011.
[31] How Many Companies Use Cloud Computing in 2022? All You Need To Know. Available at: https://tinyurl.com/2p983aau.
[32] Multi-Cloud Data Solutions for Today (and Tomorrow). Available at: https://tinyurl.com/2v2bvymp.
[33] Multicloud. Available at: https://www.ibm.com/cloud/learn/multicloud.
[34] Multi-cloud mature organizations are 6.3 times more likely to go to market and succeed before their competition. Available at: tinyurl.com/2ym8xcx7.
[35] More and more companies are spreading their data over public clouds. Available at: https://tinyurl.com/46fph54z.

[36] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*, 2020.
[37] Takashi Nishide et al. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC*, pages 343–360, 2007.
[38] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions, 2015. National Institute of Standards and Technology, NIST FIPS 202. Available at https://doi.org/10.6028/NIST.FIPS.202.
[39] Enron Email Dataset. Available at: http://www.cs.cmu.edu/ enron/.
[40] Emma Dauterman et al. DORY: an encrypted search system with distributed trust. In *OSDI*, pages 1101–1119, 2020.
[41] Frank Wang et al. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *NSDI*, pages 611–626, 2016.
[42] Yuncong Hu et al. Ghostor: Toward a secure data-sharing system from decentralized trust. In *NSDI*, pages 851–877, 2020.
[43] John Liagouris et al. SECRECY: secure collaborative analytics in untrusted clouds. In *NSDI*, pages 1031–1056, 2023.
[44] Niv Gilboa et al. Distributed point functions and their applications. In *EUROCRYPT*, volume 8441, pages 640–658, 2014.
[45] https://www.oblivious.app/.
[46] Ravi S. Sandhu et al. Access control: principle and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.
[47] Amit Sahai et al. Fuzzy identity-based encryption. In *EUROCRYPT*, pages 457–473, 2005.
[48] John Bethencourt et al. Ciphertext-policy attribute-based encryption. In *IEEE SP*, pages 321–334, 2007.
[49] Sanjam Garg et al. Attribute-based encryption for circuits from multilinear maps. In *CRYPTO*, pages 479–499, 2013.
[50] Moni Naor et al. Access control and signatures via quorum secret sharing. In *CCS*, pages 157–168, 1996.
[51] S. Servan-Schreiber et al. Private access control for function secret sharing. In *IEEE SP*, pages 1257–1276, 2023.
[52] Elette Boyle et al. Function secret sharing. In *EUROCRYPT*, pages 337–367, 2015.
[53] Muhammad I. Sarfraz et al. Dbmask: Fine-grained access control on encrypted relational databases. *Trans. Data Priv.*, 9(3):187–214, 2016.
[54] Jingwei Li et al. Efficient keyword search over encrypted data with fine-grained access control in hybrid cloud. In *NSS*, pages 490–502, 2012.
[55] Maithilee P. Joshi et al. Semantically rich, oblivious access control using ABAC for secure cloud storage. In *IEEE EDGE*, pages 142–149, 2017.
[56] Lukas Burkhalter et al. Timecrypt: Encrypted data stream processing at scale with cryptographic access control. In *NSDI*, pages 835–850, 2020.
[57] Hossein Shafagh et al. Droplet: Decentralized authorization and access control for encrypted data streams. In *USENIX Security*, pages 2469–2486, 2020.
[58] Dan Bogdanov et al. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.
[59] MongoDB Releases Queryable Encryption Preview. Available at: https://tinyurl.com/yc45dv6h.
[60] MongoDB Queryable Encryption. Available at: https://www.mongodb.com/docs/upcoming/core/queryable-encryption/.
[61] GarbleCloud. Available at: https://www.garblecloud.com/.
[62] Virtru. Available at: https://www.virtru.com/.
[63] Enveil. Available at: https://www.enveil.com/.
[64] Proton Drive: Secure cloud storage and file sharing. Available at: https://proton.me/drive.
[65] Emil Stefanov et al. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, pages 299–310, 2013.
[66] Benny Chor et al. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.
[67] Emma Dauterman et al. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, pages 655–671, 2021.
[68] Benny Chor et al. Private information retrieval by keywords. *IACR Cryptol. ePrint Arch.*, page 3, 1998.
[69] Ian Goldberg. Improving the robustness of private information retrieval. In *IEEE SP*, pages 131–148, 2007.
[70] Seny Kamara et al. Computationally volume-hiding structured encryption. In *EUROCRYPT*, pages 183–213, 2019.
[71] Sarvar Patel et al. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *CCS*, pages 79–93, 2019.
[72] Kui Ren et al. Hybridx: New hybrid index for volume-hiding range queries in data outsourcing services. In *ICDCS*, pages 23–33, 2020.
[73] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.
[74] Daniel Escudero. An introduction to secret-sharing-based secure multiparty computation. *IACR Cryptol. ePrint Arch.*, page 62, 2022.

APPENDIX

A PHASE 3 — ADDITIONAL OPERATIONS

This section develops methods for fully secure operations on AP list and another for fetching multiple files using a single vector of $O(\delta)$, where δ is the number of files.

A.1 Fully Secure Operations on AP List

Recall that based on AP list (see Table 5), the client will learn the position of the keywords in AC matrix and the number of keywords appearing in a file (see §5). To overcome this, we develop a method below that requires us to change the structure of AP list.

New structure of AP list. In the new structure of AP list for each file, we store a bit value of 1 or 0, for all keywords appearing in AC matrix, where 1 refers to the keyword appearance in the file; otherwise, 0. These values are placed according to the position of the keywords in AC matrix. For example, refer to the first row of Table 21 that corresponds to the first file, containing only one keyword (*i.e.*, Are). The keyword 'Are' appears at the first position of AC matrix and there is no other keyword of AC matrix appears in the first file; thus, we add $\langle 1, 0, 0\rangle$ in the first row of AP list. Further, the hash digest has been added as usual, like the old structure of AP list (Table 23). DBO creates SSS of this new AP list and outsources them with files.

Advantage of the new AP structure. The new structure of AP list prevents an adversary from knowing the maximum number of keywords appearing in a file. Now, the adversary learns that all the keywords of AC matrix appear/disappear in each file.

Query execution. Recall that we used the old structure of AP list in Phase 3 STEP 6, 8A (see §6.4). We use the new structure of AP list in the same steps, and below, we reproduce the same steps with the new AP list, denoted by *mAP*.

STEP 6: ***Server:*** receive the vector $\mathbb{M}(v)$ from the client and has three objectives: (*i*) whether the vector contains all zero and except for a single one; (*ii*) the vector contains one at the position of the file-id that was sent in Phase 2 by servers; and (*iii*) the requested file does not contain a keyword to which the client has no access. Test A and Test B of §6.5 achieve the first objective. Test 2 achieves the second objective, as mentioned in §6.4. Below, we produce Test 3 for achieving the third objective.

Test 3 on new AP list: $\mathbb{M}(test_3)[] \leftarrow (\mathbb{M}(v) \odot \mathbb{M}(mAP)) + \mathbb{M}(\mathsf{RN})_z$

Test 3, on the success of Test 2, performs a dot product between $\mathbb{M}(v)$ and the new AP list ($\mathbb{M}(mAP)$ (see the second column of Table 24) at $\mathcal{S}_z$. This results in a vector of size β (where β is the number of keywords in AC matrix) and the hash digest. The degree of polynomial for each such element will be two. $\mathcal{S}_z$ needs to send the vector to the client to reduce the degree to one. Adding $\mathbb{M}(\mathsf{RN})_z$ hides from the client the number of keywords in the file during degree reduction. Thus, even if a minority of servers (*i.e.*, f servers) collude with the client, they cannot learn any information about the number of keywords appearing in the file. Also, servers keep the hash digest (denoted by $\mathbb{M}(\mathsf{H}(AP))$) at their end.

File-ids	File content
1	How are you
2	Are you Ana
3	Fig is a fruit

Table 21: Cleartext files.

Keywords →	Are	Ana	Fig
Starting address in the inverted list →	1	2	3
Hash values →	H(1)	H(2)	H(3)
Clients' information ↓			
Lisa	**0**	**1**	**2**
Ava	**3**	**0**	**0**

Table 22: Access control matrix.

F-ids	AP list	File content with digest
1	1,0,H(1)	How are you, H(how are you, H(1))
2	1,2,H(1)+H(2)	Are you Ana, H(are you Ana, H(2))
3	3,0,H(3)	Fig is a fruit, H(Fig is a fruit, H(3))
0	0,0,0	Dummy, H(Dummy, H(0))

Table 23: The current AP list.

F-ids	AP list	File content with digest
1	1,0,0,H(1)	How are you, H(how are you, H(1))
2	1,1,0,H(1)+H(2)	Are you Ana, H(are you Ana, H(2))
3	0,0,1,H(3)	Fig is a fruit, H(Fig is a fruit, H(3))
0	0,0,0	Dummy, H(Dummy, H(0))

Table 24: The new AP list.

STEP 7: ***Client:*** interpolates the received values and creates SSS of the same vector with a polynomial of degree one and sends the share to servers. We denote this vector by $\mathbb{M}(vAP)$.

STEP 8A: ***Server:*** checks (*i*) $\mathbb{M}(vAP)$ contains only zeros and ones using Test C of §6.5; (*ii*) $\mathbb{M}(vAP)$ is created for positions of keywords appear in the vector sent in STEP 6 using Test 4 of §6.4.1; (*iii*) finally, the file does not contain any keyword to which access is denied to the client, using Test 5 of §6.4.1. Note that in Test 4 and Test 5, servers will use the vector $\mathbb{M}(vAP)$.

If the above tests succeed, then servers will execute STEP 8B to send the file.

A.2 Phase 3: Fetching Multiple Files

A query keyword can appear in multiple files. §6.4 has shown how we can a single file and server can verify the vector. To fetch γ files, we need to send γ vectors, each of size δ, where γ is the maximum number of files in which a keyword appears and δ is the number of files. This results in the communication overhead of $\gamma \times \delta$ between a server and the client. To reduce such a communication overhead, this section develops a method to fetch γ files *without* sending γ vectors, each of size equal to the number of files δ.

Disjoint Partitioning/Binning of Files. This method partitions the files into multiple blocks/bins, such that all the desired file-ids and files belong to different blocks/bins. For example, if the client wants to fetch γ files, the client can request the servers to create γ partitions/bins of the file set, such that each of the desired γ files stay in a different partition. Then, the client will create γ bit-vectors, each of size δ/γ, such that each vector contains zeros except only one that corresponds to the desired file-id's position in the partition/bin. Note that the total elements in this vector are equal to the number of files. The client creates secret-shares of the vector. Servers will perform a dot product over each vector and do the verification before sending the files to the client, using the method developed in §6.4.

F-ids	AP list	File content with digest
1	1,0,0,H(1)	How are you, H(how are you, H(1))
2	1,1,0,H(1)+H(2)	Are you Ana, H(are you Ana, H(2))
3	0,0,1,H(3)	Fig is a fruit, H(Fig is a fruit, H(3))
0	0,0,0	Dummy, H(Dummy, H(0))

Table 25: The new AP list.

F-ids	AP list	File content with digest
1	1,0,0,H(1)	How are you, H(how are you, H(1))
3	0,0,1,H(3)	Fig is a fruit, H(Fig is a fruit, H(3))

Table 26: The first bin.

F-ids	AP list	File content with digest
2	1,1,0,H(1)+H(2)	Are you Ana, H(are you Ana, H(2))
0	0,0,0	Dummy, H(Dummy, H(0))

Table 27: The second bin.

Example. For the three files and one dummy file (see Table 25) and for a query to fetch files f_1 and f_2, we can create two joint bins as shown in Table 26 and Table 27.

Problems of disjoint partitioning. Note that if a keyword is associated with $x \ll \gamma$ files, then all the $\gamma - x$ vectors will contain only zero. However, since DBO has placed only a single fake file (see Table 25), this method will fail. Furthermore, this method has another problem: if there are many requests for the same files, say f_1 and f_2, then these two files are always placed into two different bins. This will enable the adversary to observe which files the clients are trying to fetch after many queries. To overcome this problem, we develop the following method.

Overlapped Partitioning of Files. Instead of creating disjoint bins of the files, this method creates bins such that the intersection of two or more bins never be disjoint. Thus, a file-id/file will appear in multiple blocks. In what follows, the fake file, with content zero, will also appear in multiple or all blocks — overcomes the first problem of the disjoint partitioning, as mentioned in the last paragraph. Furthermore, due to replicating the files in multiple bins, the same file may appear always in the same bin, and here, the same file can be fetched from different bins in different queries — preventing the adversary from knowing anything about the requested files even after observing many queries. While this method overcomes the problem of disjoining partitioning, it comes with additional communication cost, as files are replicated in multiple bins, increasing the total number of files in the system. Important to note that such increased files are never written to the disk — thus, the space overhead is only during query execution.

B DYNAMIC OPERATIONS

This section explains dynamic operations — add and delete.

B.1 Add Operation

The add operation allows insertion of new files with existing keywords in AC matrix or with new keywords that need to be added to AC matrix. Below, we develop algorithms for both cases.

B.1.1 ***Case 1: Adding Files having Existing keywords.*** We first consider the case of adding a new file with existing keywords, in an inverted list (Table 4) or in OptInv (Table 13). Suppose there

Positions	File-ids
1 (are)	1, 2, 3, H(2, H(1, H(are)))
2 (Ana)	2, 0, 2, H(0, H(2, H(ana)))
3 (Fig)	3, 0, 2, H(3, H(Fig))
0 (Fake)	0, 0, 0, H(0, H(Fake))

Table 28: Inverted list (modified).

is a free slot available; see the second last positions with keywords Ana and Fig, in Table 4 and the 7th slot of Table 13.

Addition operation on the inverted list. To add the new file having an existing keyword in AC matrix, DBO needs to obliviously learn the row of the inverted list corresponding to the keyword, and then, the hash digest and the empty positions in the row, which can hold the new file-ids.

For example, DBO wants to add a file-id to the third row with the keyword 'Fig,' then DBO needs to know the existing hash digest H(3, H(Fig)) and the position (*i.e.*, the second position in the third row) of the free slot, where a new file-id can be inserted.

When free slots are available in an inverted list. Now, let us consider that the inverted list contains free slots; see Table 28 where 0 refers to free slot. Further, note that we modified the inverted list having the position number where a new file can be added (see orange-colored text). If this newly added orange-colored value equals to the position where it is written, then this shows that there is no free slot — for this case, we develop an algorithm soon. For example, in the row for 'are,' the position is three for the orange-colored value and it is written at the third position of the list, showing there is no free slot for new file-id having 'are.'

After knowing the hash digest and the empty positions in the row of the inverted list, DBO sends three vectors in SSS form, each of length β (the number of keywords). Vectors are constructed as follows: all positions are set to zero except for the desired position, which contains the following values:

(1) *Vector for file-id* — the first vector contains the new file-id,
(2) *Vector for counting free slots* — the second vector contains one, and
(3) *Vector for hash digest* — the third contains the difference between the new hash digest and the old hash digest.

Upon receiving these vectors, servers add each vector to the desired position in the inverted list.

Suppose, we want to add a file to the entry corresponding to Fig, *i.e.*, $\langle 3, 0, 2, \mathsf{H}(3, \mathsf{H}(\mathrm{Fig}))\rangle$. Here, the first vector will look like ⟨0,0,new_file-id,0⟩ and will be added to the second position of the entire inverted list. This will replace 0 at the second position in the row for 'Fig' with a new file-id. The second vector will look like ⟨0,0,1,0⟩ and will be added to the third position of the entire inverted list, making it 2+1=3 for the row of 'Fig,' showing that now this row cannot hold any new file-id. The third vector will look like ⟨0,0,new_hash-digest,0⟩ and will be added to the last position of the entire inverted list.

Note that by knowing the position of zero and fetching the existing hash digest, we avoid the communication cost of fetching one of the entire rows of the inverted list. Also, ***chaining of hash digest*** avoids fetching all entries and recomputing the hash over file-ids.

No free slots in an inverted list. Now, let us consider that there is no free slot in the inverted list to add a new file-id. For example, the first row $\langle 1, 2, 3, \mathsf{H}(2, \mathsf{H}(1, \mathsf{H}(\mathrm{are})))\rangle$ for the keyword 'Are' in Table 28.

In this case, we need to adjust the inverted list. Particularly, for each row of the inverted list, we need to adjust (*i*) the last value (*i.e.*, the hash digest) by shifting towards one position to the right, and (*ii*) the count of the free slot by shifting towards one position to the right. This will increase the size of the entire inverted list and make free slots.

For example, $\langle 1, 2, 3, \mathsf{H}(2, \mathsf{H}(1, \mathsf{H}(\mathrm{are})))\rangle$ will appear as $\langle 1, 2, *, 3, \mathsf{H}(2, \mathsf{H}(1, \mathsf{H}(\mathrm{are})))\rangle$ in secret-shared form.

Here, the first vector will look like ⟨new_file-id,0,0,0⟩ and will be added to the third position of the entire inverted list. This will replace $*$ at the third position in the row for 'are' with a new file-id. The second vector will look like ⟨1,0,0,0⟩ and will be added to the third position of the entire inverted list, making it 3+1=4 for the row of 'are,' showing that now this row cannot hold any new file-id. The third vector will look like ⟨new_hash-digest,0,0,0⟩ and will be added to the last position of the entire inverted list.

The addition operation does not reveal to servers the updated row of in the inverted list, but reveals the column number of the inverted list. To avoid this, we can mimic the operation by adding vectors having all zeros in several columns.

Addition operation over OptInv. To perform the add operation on OptInv, DBO needs to update AddrList and OptInv. DBO prepares a vector of size equal to the number of keywords with the desired keyword index set to one to fetch keyword's AddrList values of SiP, CuT, HD and HdV from the server. For example, for the keyword 'ana', DBO will fetch the value as: SiP as 4, CuT as 3, and H_2 and h_2. The update of AddrList is done similarly to the add operation on the inverted list.

Now, to add new file-ids in OptInv, DBO prepares *row_vector* and *pos_vec* based on the AddrList's SiP and CuT values. With these vectors, DBO fetches the file-ids currently associated with the keyword and their hash digest value. For keyword 'ana', to add a new file-id f_{new}, DBO needs to update the value of hash digest and fill the empty slot with the new file-id, *i.e.*, the slot 6 and slot 7 in Table 13 need to be updated.

To do so, DBO sends a vector in SSS form of length OptInv with all zeros, except for the desired positions containing the new file-id and hash digest. For example, Slot 6 of the vector will contain the new file-id, and slot 7 will store the new hash digest. Servers add this vector to the existing OptInv.DBO can also use this method to insert multiple new file-ids in a single round if there are several desired number of empty slots. Note that this method is completely oblivious, but incurs communication cost. Further, when there is no empty slot to insert the file-ids for a keyword, this method will not work.

To address the above problems, we propose the following method. We organize OptInv in the form of a matrix, as in §7. Let i be the row of the matrix, where we want to insert new file-ids, but there is no free slot. To insert the item in the i^{th} row, DBO fetches the entire i^{th} row and inserts the new file-ids, as they want. Note that this may increase the number of elements in the row. To have the row with the same number of elements as other rows of the matrix, all the exceeding elements are placed into a new row. This new row

is padded with fake items to have the same size. Finally, DBO sends two rows in SSS form to servers that replace the old i^{th} row with the new row and place the new row at the $(i+1)^{th}$ place in the matrix. Note that while this method enables inserting new file-ids, it reveals the row-ids of the matrix where the elements are inserted. However, servers do not know the exact place in the row where new file-ids are inserted.

Furthermore, AddrList is updated for each keyword. In case of the presence of a free slot in OptInv, updating SiP, CuT, HD, HdV for each keyword is easy. DBO sends four vectors, each of length β.

All these vectors contain zero, except at the desired position. Particularly, the first vector contains zero at the desired position also. The second vector contains x at the desired position, where x is number of files got inserted. The third vector contains $\mathsf{H}(\text{SiP}, \text{CuT}+x, \text{keyword}) - \mathsf{H}(\text{SiP}, \text{CuT}, \text{keyword})$ at the desired position. The last vector will contain $\mathsf{H}(\text{CuT}+1) + \ldots + \mathsf{H}(\text{CuT}+x)$ at the desired position. Servers will add such vector to AddrList.

In case of no free slots in OptInv, however, updating HD and HdV is not trivial, since the addition of a new row in OptInv shifts the proceeding values of OptInv. In this case, when a new row has been added, DBO reconstructs the entire AddrList.

B.1.2 *Case 2: Adding Files with New Keywords.*

To add a new keyword $\mathcal{K}$, which is not present in the AC matrix, we need to update AC matrix, inverted list or AddrList and OptInv. DBO performs a simple append operation to update each of these lists.

Updating AC matrix. To update AC matrix, we prepare a column vector with the first value as the keyword $\mathcal{K}$, followed by the keyword position value in AddrList or inverted list, following by the hash digest of the position, and finally, followed by the access value for each client.

For instance, for a new keyword say `is`, at the 5th position, for two clients (Lisa and Ava in Table 3) each of them having access to the keyword, DBO prepares a vector as $\langle \text{is}, \mathsf{H}(5), 0, 0\rangle$. Then, DBO creates shares of the vector and sends to the servers that append to the vector at the end of AC matrix

Updating inverted list. In a similar manner, DBO updates inverted list with a new row of file-ids containing the new keyword and their hash digest, and servers append at the end.

Updating AddrList. AddrList is updated similarly to the inverted list with a vector having values for SiP, CuT, HD and HdV for the keyword Bob.

Updating OptInv. To update OptInv, DBO creates a vector like the inverted list and servers append it to the end of OptInv. It may be possible that the newly added keyword could occur in files added previously. However, in this case, *we only consider updating the new keyword for newly added files.* To update the new keyword with previous file-ids, DBO needs to know the files containing the keyword and then update AP list also, along with AC matrix, AddrList, OptInv. However, adding a new keyword to existing files' AP list is a very costly operation, which may require the owner to fetch all the files, update the AP list, and then recreate the shares of all the files.

B.2 Delete Operation

DBO can delete a keyword or delete a file-id associated a keyword.

Case 1: Delete a keyword. DBO needs to update AC matrix to delete a keyword. To do so, DBO has two ways: one, which is completely secure but requires huge communication overhead, and second, which has low communication overhead but suffers from access pattern leakage.

(*i*) To update AC matrix, DBO prepares a matrix of dimensions that is the same as the AC matrix, filled with zeros for all places except for the capability of the keyword for all clients. The values corresponding to the deleted keyword will store the random number, representing no access for the keyword for any client. DBO sends the matrix in SSS form to the servers, which adds the newly received AC matrix to the existing AC matrix. Since each client now has no search access to the keyword, the keyword is considered to be deleted from AC matrix.

(*ii*) To update AC matrix, DBO prepares a vector of size equal to the number of clients. The vector will contain random number, representing no access to the keyword. DBO outsources the vector in secret shares to the servers, which adds this vector to the AC matrix at the position of the keyword to be deleted.

Although the approach is faster than the previous approach, the method suffers from access-pattern leakage as the position of the deleted keyword is revealed to the servers.

Case 2: Delete a file-id associated with a keyword. DBO needs to update the inverted list or OptInv to delete the file-id associated with a keyword. DBO can perform the deletion of file-ids in two ways — one provides complete security but suffers from huge communication overhead, and the second leaks access pattern but has low communication overhead.

(*i*) To update the inverted list, DBO first needs to obliviously know the i^{th} row of the inverted list associated with the keyword. Then, DBO prepares a new inverted list of size the same as the inverted list filled with zeros except for the i^{th} row. The i^{th} row is filled with zeros except for the j^{th} position containing the negative value of the file-id to be deleted and the last position to have the difference of the new hash digest and the previous hash digest. Then, DBO will create the shares of this new list and send to servers, which will add the new list to the existing inverted list.

Likewise, DBO updates OptInv after organizing OptInv in a matrix form, obliviously fetching the i^{th} row of the matrix, and finally, obliviously updating the entire OptInv along with new hash digest. This method, although completely secure, requires huge communication cost, since the data of size equals to the size of an inverted list or (OptInv) is transmitted to the servers.

(*ii*) To update the inverted list, after knowing i^{th} row of the inverted list associated with the keyword, DBO prepares a vector with deleted file-id and new hash digest. Then, DBO creates shares of the vector and sends to servers, which replaces the vector with the existing i^{th} row of the inverted list. Likewise, DBO can fetch i^{th} row by organizing OptInv in matrix form and asks the servers to replace the i^{th} row. While this method is efficient, this method reveals access-patterns to the servers.

Furthermore, there could be another possibility by fetching the j^{th} file to be deleted at DBO and replacing the j^{th} file at the server with dummy data.

C GRANTING OR REVOKING ACCESS RIGHTS

Granting or revoking access rights for a client in AC matrix (Table 3) is done by DBO. A trivial way to do this is to download all the keywords and the client's access rights at DBO, then update the access rights, and finally, re-create shares of the keywords and access rights. This method requires 4β communication cost. Instead, DBO can grant/revoke access rights with communication cost of 2β using the following method.

First, DBO needs to know the position of the desired keyword and the current access rights for the keyword, and then, needs to change the right for the keyword.

Recall that to show a no access for a keyword for a client, AC matrix contains a large random number; as mentioned in §5. To make grant and revoke operations easy, DBO can select such random numbers using $\mathcal{PRG}()$ which can take a secret seed, the keyword, and the client-id as input. Now, to grant or revoke access, DBO can work as follows:

To know the index (*i.e.*, the position of the keyword in AC matrix) and the access rights of the keyword for a client, DBO asks servers to perform Phase 1 without performing multiplication with $\mathbb{M}(\mathsf{RN}_z)$. DBO learns the output is either zero or a random number. Since DBO has allocated random numbers of each keyword for the client using a function, such as $\mathcal{PRG}$(seed, keyword, client), DBO can learn the index of the keyword in AC matrix.

Now, DBO prepares a vector of size equal to the number of keywords filled with all zeros except for the index of the keyword, which is replaced with value $\mathcal{PRG}$(seed, keyword, client) (or negative value of $\mathcal{PRG}$(seed, keyword, client)) for changing from access (or no access) to no access (or access). DBO creates shares of this vector and sends to servers, which perform an add operation with the vector corresponding to the client in AC matrix that updates the access rights for the client.

D VERIFICATION OPERATIONS AT CLIENTS

Three phases of DOC* come with three different verification objectives for the client. In a nutshell, *clients wish to verify the correctness of the access rights in Phase 1 and to verify the correctness and completeness (of file-id and files) in Phase 2 and Phase 3.*

Verification in Phase 1. Recall that Phase 1 executes at three servers. If one of the servers does not perform computation (*i.e.*, checking access rights) correctly on AC matrix, the client will always receive random numbers, showing no access to search for the keyword. Thus, in Phase 1, the client's objective is to verify that they really have no access to search the keyword, when receiving random numbers. For this, the client executes the computation of Phase 1 at four servers, among them, only one can be malicious. Then, the client interpolates the received vectors in triplets, (as $\langle \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3 \rangle$, $\langle \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_4 \rangle$, $\langle \mathcal{S}_1, \mathcal{S}_3, \mathcal{S}_4 \rangle$, and $\langle \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4 \rangle$) and compares the answer for each triplet. If one of the servers has not performed the work correctly, all four values will not be identical.

Verification in Phase 2. The client's objective is to verify that they receive all the correct file-ids, associated with the keyword. To do so, in STEP 5A, the client computes the hash function over all the received file-ids identically to DBO (see Table 4). If *the servers have returned all the file-ids associated with the keyword, the computed hash digest will match against the received hash digest.*

Verification in Phase 3. The client's objective is to verify that they receive the entire file, containing the keyword. To do so, they perform similar processing as for the verification in Phase 2.

Security discussion. Servers do not learn anything, since verification does not involve any additional steps at the server. Clients also learn nothing extra from the hash digest they compute.

E ADDITIONAL EXPERIMENTS

This section evaluates DOC* on additional criteria, as follows:

(1) Different implementation of AP list for Phase 3 — Exp A1.
(2) Processing time for granting or revoking access rights — Exp A2.
(3) Processing time for adding new file-id(s) to an existing keyword — Exp A3.
(4) Processing time for deleting an existing keyword — Exp A4.
(5) Processing time for deleting an existing file — Exp A5.
(6) Processing time on increasing the number of servers — Exp A6.

Exp A1: Different implementation of AP list. This experiment shows the impact of using different implementations of AP list in Phase 3. The one we developed in §6.4.1, denoted by *reduced-size AP list* in Figure 5, reveals the number of keywords in a file and their positions in AC matrix to the client, while another method developed in Appendix A.1, denoted by *large-size AP list* in Figure 5, does not reveal anything to the client. Recall that the method for the large-size AP list of Appendix A.1 places 0 or 1, equals to the number of keywords β in AC matrix, in each entry of AP list in share form, and this results in space overhead compared to reduced-size AP list. In particular, the large-size AP list at each server required 7.31GB for the database having 5K keywords and 500K files, while the reduced-size AP list at each server took 381MB for the same database.

Due to the less data in the reduced-size AP list, it also performs better compared to the large-size AP list; see Figure 5.

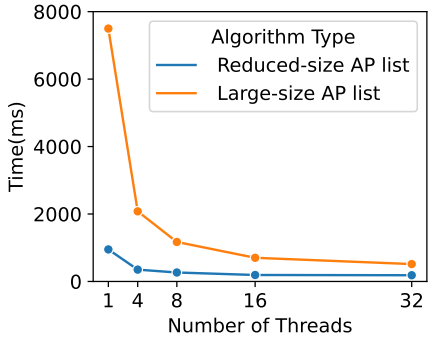


Figure 5: Exp A1: Different implementations of the AP list in Phase 3.

Exp A2: Processing time for granting or revoking access rights. To grant/revoke access to a keyword to a client, DBO needs to update the row corresponding to the client in AC matrix. To do so, DBO creates a vector of size equals to the number of keywords in AC matrix. Recall that as mentioned in Appendix C, the vector will contain all zeros except for the index of the keywords, which is replaced by a random number (*i.e.*, $\mathcal{PRG}$(seed, keyword, client)) in case of revoking access rights; otherwise the negative random number (*i.e.*, $-\mathcal{PRG}$(seed, keyword, client)) will appear at the index corresponding to the keyword in case of granting access.

For 5,000 keywords, the entire process took ≈25.9ms using a single thread. Particularly, DBO took 18.3ms to prepare the vector with updated access rights, servers took 5ms to update a row of AC matrix, and the communication time between the server and DBO was ≈2.6ms.

Exp A3: Processing time for adding new file-id(s) to an existing keyword. We performed this experiment only over AddrList and OptInv, since such data structures perform well compared to the inverted list, as shown in Exp 4 in §8. To add new file-id(s) to an existing keyword requires updating both OptInv and AddrList. There are two cases depending on the availability of free slots in OptInv; as mentioned in Appendix B.1.

First, when OptInv contains a free slot to accommodate the new file-id(s), and second is one when OptInv has no free slot to accomodate any new file-id.

In the first case, on a dataset of 5K keywords and 500K files, the total time taken to update AddrList and OptInv with the new file-id(s) for a keyword was 2821.3ms using a single thread. Specifically, DBO took 1637ms to create vectors, while the server took 173ms to update the existing AddrList and OptInv, and the time taken to transfer the updated vector from DBO to the server was 1011.3ms.

In the second case, the total time to add new file-id(s) was 18091.6ms using a single thread. Particularly, DBO took 16732ms to create updated information for AddrList and OptInv, whereas the server took 304ms to update these data structures. The network transfer time was 1055.6ms. Observe that DBO is taking more time, due to first reconstructing AddrList at their end, then updating the values of SiP, CuT, and hash digests, and finally, recreating the shares of AddrList.

Exp A4: Processing time for deleting an existing keyword. The deletion of an existing keyword using the *secure approach* of Appendix B.2 requires an update of the capability for all clients in AC matrix. For 4,096 clients and 5K keywords, the total time taken to perform the keyword deletion operation was 6620.1ms using a single thread. Specifically, DBO took 3209ms to prepare the vector, a server took 3158ms, and the network transmission time was 253.1ms.

Exp A5: Processing time for deleting an existing file. The deletion of an existing file for a keyword using the *secure method* requires an update of OptInv only to delete the file-id, see Appendix B.2. For 5K keywords and 500K files, the overall time taken to delete a file for a keyword was 3117.ms using a single thread. Particularly, DBO took 1761 ms to prepare and share the updated vector, while the server took 276ms to perform addition operation on existing OptInv. The total network time incurred in the deletion operation was 1080ms.

Exp A6: Processing time on increasing the number of servers. Our experiments in §8 used four servers for achieving the functionality of DOC*. This experiment evaluates how DOC* performs when increasing the number of servers from four to eight. To do so, we created two groups/clusters, each having four servers.

Each cluster stores a subset of DOC*'s data structures. We divide AddrList and Files equally between the two clusters. Based on the last row of AddrList and its SiP and CuT values at the first cluster, OptInv is partitioned appropriately such that the file-id(s) associated with the last keyword is included within the corresponding partition of OptInv in the first cluster. AC matrix remains undivided as required for the operation DOC* protocol, to execute Test 5 in Phase 3.

Table 29 shows the result of this experiment. As we increase the number of servers, the processing time decreases, since each cluster of four servers processes only a subset of OptInv, AddrList, and Files. For simplicity of comparison, we assume both client and server are trusted and hence eliminated the execution of the tests. For a database 500K files and 5K keywords using a single thread, the total processing time decreases from 1045.1ms to 653ms. The breakdown of time taken by each phase is mentioned in Table 29.

The size of AddrList and Files is reduced by half when using two clusters. However, the size of OptInv reduces from 138.6MB in the case of a single cluster to 110.6MB in the case of two clusters. Observe that the size of OptInv does not reduce by half, due to the distribution of keywords appearing in the file.

# servers	Phase 1	Phase 2	Phase 3	Total
4	**16.6ms**	**78.7ms**	**949.8ms**	**1045.1ms**
8	**16.4ms**	**63.9ms**	**574.5ms**	**654.8ms**

Table 29: Exp A6: The impact of increasing the number of servers.

F SECURITY PROOFS

This section provides the security of RN, security of randomization, and formal security proof.

F.1 Verification of Random number Generation

According to the description of distributed secret-shared random number generation in §6.1, we can see that RN is produced by adding three random numbers generated by $\mathcal{S}_{z\in\{1,2,3\}}$, respectively. However, a malicious server can do the following behaviors:

- Do not send anything to other servers, this malicious can be detected immediately;
- Sent arbitrary numbers instead of shares to other servers. This is what we should verify.

Note that due to incorporating the client-side verification as mentioned in §6.1, the secret-shared random numbers will be generated at four servers, and they will verify the random number. Below, we provide a case where three servers verify random numbers. The following case also generalizes to the four servers.

According to the Lagrange interpolation (*LI*), to generate $\mathbb{M}(\mathsf{RN})[i]$), no matter which random number is sent by the malicious server, the aggregated shares will produce a polynomial

of degree less than two. We classify these results into two cases: (*i*) polynomial of degree one, (we recognize it is legal); and (*ii*) polynomial of degree two. In this case, the computation of $\sum_{1\le i\le\beta}(PRG(seed)\times\mathbb{M}(\mathsf{RN})[i])$ mod p will also produce a polynomial of degree two. Please note that multiplying with $PRG(seed)$ can prevent malicious servers from passing this test, as no server can predict which number will multiply with $\mathbb{M}(\mathsf{RN})[i]$). Before proving Theorem 1, let us introduce a lemma.

Lemma 1: Let $F(x)$ be a polynomial of degree at most two over $\mathbb{F}_p$, $f_1(x)=(x-c_1)(x-c_2)$ and $f_2(x)=(x-c_1)(x-c_3)$, where c_1,c_2,c_3 are pairwise different. Then $F(x)$ mod $f_1(x)\neq F(x)$ mod $f_1(x)$.

PROOF. Provide that $F(x)=h_1(x)f_1(x)+r_1(x)$ and $F(x)=h_2(x)f_2(x)+r_2(x)$. Obviously, if $r_1(x)=r_2(x)$, then we can deduce that $h_1(x)f_1(x)=h_2(x)f_2(x)$. Recall that the degree of $F(x)$ is at most two. Thus both $h_1(x)$ and $h_2(x)$ are constants. Without loss of generality, let $h_1(x)=\alpha, h_2(x)=\beta$, we have $\alpha\cdot(x-c_1)(x-c_2)=\beta\cdot(x-c_1)(x-c_3)$. Note that c_1,c_2,c_3 are all different, no matter what α,β are, the equation $\alpha\cdot(x-c_2)$ is not equal to $\beta\cdot(x-c_3)$. Thus, the above equation will never hold; this is a contradiction. □

Theorem 1: If malicious servers have not created the shares of random numbers correctly, $LI(a_z,a_{z+1})$, $LI(a_{z+1},a_{z+2})$, $LI(a_{z+2},a_z)$, at $\mathcal{S}_z$ will produce different results, where $LI(*)$ means Lagrange interpolation.

PROOF. Let $F(x)$ denote the secret polynomial of $\sum_{1\le i\le\beta}(PRG(seed)\times\mathbb{M}(\mathsf{RN})[i])$ mod p. If malicious servers create shares incorrectly, the degree of $F(x)$ will be two. When performing interpolation for every two shares, $LI(a_z,a_{z+1})$=$F(x)$ mod $(x-c_1)(x-c_2)$, $LI(a_{z+1},a_{z+2})$=$F(x)$ mod $(x-c_2)(x-c_3)$ and $LI(a_{z+2},a_z)$=$F(x)$ mod $(x-c_3)(x-c_1)$, where $c_1\ldots,c_3$ are the values used in polynomials (*e.g.*, $f(x{=}c_1)=(k{\times}c_1+secret)$ mod p) to create Shamir's share, are pairwise different. According to Lemma 1, $LI(a_z,a_{z+1})\neq LI(a_{z+2},a_z)\neq LI(a_{z+1},a_{z+2})$; thus, we conclude Theorem 1. □

F.2 Security of Random Numbers

We further prove the security of Phase 1, as it involves an addition step of multiplication of RN. A similar step is also used in STEP B of server processing over OptInv in §7.1.

Theorem 2. ***If a malicious client colludes with a minority of malicious servers, such malicious entities cannot deduce any extra information for the shares they obtained, except for $ans\mathcal{S}_z$, in Phase 1.***

PROOF. We first recall the following keyword search formulation in Phase 1:

$$\mathbb{M}(ans\mathcal{S}_z)[i]\leftarrow[(\mathbb{M}(sw)[i]{-}\mathbb{M}(uw)+\mathbb{M}(AC)[i])\times\mathbb{M}(\mathsf{RN})[i]]+\mathbb{M}(0)\bmod p,\forall i\in\{1,\ldots,\beta\}$$

For the i-th answer, the malicious client has the share of $ans\mathcal{S}_z$, also the malicious server can provide its shares of each operand.

Without loss of generality, we can recognize the formula $sw[i]{-}uw+AC[i]$ as a integral, denoted by θ_i. Now, θ_i is secret-shared using a linear polynomial $f(x)=a_1x+\theta_i$. Similarly, $\mathsf{RN}[i]$ is shared by another linear polynomial, denoted by $g(x)=b_1x+\mathsf{RN}[i]$. Now, we assume that $\mathcal{S}_1$ is malicious, while $\mathcal{S}_2$ and $\mathcal{S}_3$ are honest. According to SSS, each of the servers keeps the shares of θ_i, $\mathsf{RN}[i]$, as follows:

$\mathcal{S}_1$: $a_1+\theta_i, b_1+\mathsf{RN}[i]$; for $x=1$,
$\mathcal{S}_2$: $2a_1+\theta_i, 2b_1+\mathsf{RN}[i]$; for $x=2$,
$\mathcal{S}_3$: $3a_1+\theta_i, 3b_1+\mathsf{RN}[i]$, for $x=3$.

The multiplications between shares of θ_i and $\mathsf{RN}[i]$ corresponds the polynomial multiplication of $f(x)\times g(x)$, *i.e.*,

$$\begin{aligned}f(x)\times g(x)&=(a_1x+\theta_i)\times(b_1x+\mathsf{RN}[i])\\&=a_1b_1x^2+(\theta_ib_1+\mathsf{RN}[i]a_1)x+\theta_i\mathsf{RN}[i]\end{aligned}$$

Also note that this product should be add with an extra polynomial $h(x)=h_1x^2+h_2x$, which corresponds to the secret polynomial of degree two for $\mathbb{M}(0)$.

The malicious client can obtain all the shares stored at the malicious server $\mathcal{S}_1$ and coefficients of $f(x)\times g(x)+h(x)$. Here, the malicious client wants to recover unknown θ_i, which may reveal the keyword. Then, malicious client has to solve a non-linear equation set:

$$\begin{aligned}&a_1+\theta_i=v_1\\&b_1+\mathsf{RN}[i]=v_2\\&a_1b_1+h_1=v_3\\&\theta_i\times b_1+\mathsf{RN}[i]\times a_1+h_2=v_4\\&\theta_i\times\mathsf{RN}[i]=v_5\\&h_1+h_2=v_6\end{aligned}\tag{1}$$

The parameters $\{a_1,b_1,\theta_i,\mathsf{RN}[i],h_1,h_2\}$ are all unknown. When we plug the first two equations into the remaining one, we can obtain

$$\begin{aligned}&a_1b_1+h_1=v_3\\&(v_1-a_1)\times b_1+(v_2-b_1)\times a_1+h_2=v_4\\&(v_1-a_1)\times(v_2-b_1)=v_5\\&h_1+h_2=v_6\end{aligned}$$

Further, simplifying the above three equations results in an equation without any unknowns $v_1v_2+v_6=v_3+v_4+v_5$. For the next query, as servers choose different polynomials for $\mathbb{M}(0)$ to randomize the secret polynomial, even malicious client searches the same uw, the client obtains similar equations with the different parameter set $\{h_1',h_2',b_1',\mathsf{RN}'[i]\}$ (a_1,θ_i is unchanged), which are independent of the previous $\{h_1,h_2,b_1,\mathsf{RN}[i]\}$. Therefore, new equations cannot help the client knowing more about θ_i. That is to say, the malicious client cannot deduce θ_i according to the equation set (1), which concludes the theorem. ■ □

F.3 Security of Randomization

We now consider another kind of malicious behavior of the client. Assume that server $\mathcal{S}_1$ is malicious. Note that $\mathbb{M}(sw[i])$ and $\mathbb{M}(AC)[i]$ are fixed. If a *malicious client generates $\mathbb{M}(uw)$ using the same secret polynomial for multiple queries*, the secret polynomial related to $\mathbb{M}(sw)[i]{-}\mathbb{M}(uw)+\mathbb{M}(AC)[i]$ is fixed, and denote such a polynomial by $f(x)=a_1x+\theta_i$. Meanwhile, the polynomials related to $\mathbb{M}(\mathsf{RN})[i]$ for two queries are chosen as $b_1x+\mathsf{RN}[i]$ and

$b_2x + \mathsf{RN}'[i]$. Therefore, when the client collects and interpolates shares from all the servers, they will obtain:

$$\begin{aligned} F_1(x) &= a_1b_1x^2 + (a_1\mathsf{RN}[i] + b_1\theta_i)x + \mathsf{RN}[i]\theta_i \\ &= a_1b_1(x + \theta_i a_1^{-1})(x + \mathsf{RN}[i] \times b_1^{-1}), \\ F_2(x) &= a_1b_2x^2 + (a_1\mathsf{RN}'[i] + b_2\theta_i)x + \mathsf{RN}'[i]\theta_i \\ &= a_1b_2(x + \theta_i a_1^{-1})(x + \mathsf{RN}'[i] \times b_2^{-1}). \end{aligned}$$

Then, the client can find the common divisor between $F_1(x)$ and $F_2(x)$ and guess the explicit formula of $a_1x + \theta_i$. It is easy to know $(x + \theta a_1^{-1})$ is one of its divisors. But its constant divisor is hard to guess, as in the finite field $\mathbb{Z}_p$, a_1b_1 and a_1b_2 can have arbitrary nonzero elements to be their common factors.

In this context, $\mathcal{S}_1$ send its shares, *i.e.*, $f(1) = a_1 + \theta_i$ to the client. The client can now compute $(1 + \theta a_1^{-1})$ and compare it with $f(1)$ to determine the explicit formula of $f(x)$. Therefore, the client can know $sw[i] + AC[i]$. Combined with the result of Test 1 of §6.3 (in the case of an illegal vector that produces the value for the non-access right), a malicious client can know $sw[i]$.

If ***the servers randomize the share product by adding*** $\mathbb{M}(0)$***, such an attack is avoided.***

Theorem 3. ***Randomization can prevent malicious clients from deducing*** $sw[i] + AC[i]$ ***over multiple queries, even colluding with a minority of malicious servers.***

Proof. Since the servers add $\mathbb{M}(0)$ to $(\mathbb{M}(sw) - \mathbb{M}(uw) + \mathbb{AC}[i]) \times \mathbb{M}(\mathsf{RN})[i]$ for each queries, the secret polynomials related previous shares are not identical. Let $f(x) = a_1x + \theta_i$ denote its former meaning. When such polynomials multiply with $b_1x + \mathsf{RN}[i]$, $b_2x + \mathsf{RN}'[i]$, and then add with $h_1x^2 + h_2x$, $h_3x^2 + h_4x$, respectively, the client can obtain two polynomials as:

$$F_1(x) = (a_1b_1 + h_1)x^2 + (a_1\mathsf{RN}[i] + b_1\theta_i + h_2)x + \mathsf{RN}[i]\theta_i$$
$$F_2(x) = (a_1b_2 + h_2)x^2 + (a_1\mathsf{RN}'[i] + b_2\theta_i + h_3)x + \mathsf{RN}'[i]\theta_i$$

At this time, there is no common divisor between $F_1(x)$ and $F_2(x)$ and the shares of $\mathcal{S}_1$ can be the output of any these divisors. Thus, malicious client cannot utilize polynomial factorization to obtain extra information. □

F.4 Security Proof of Doc⋆

We prove the security of Doc⋆ by following real-ideal paradigm. For the details of the real-ideal paradigm, interested readers may refer to Chapter 7 of [73], Chapters 1 and 4 of [74], or [30].

- In the real world, the adversary corrupts a subset of the servers of cardinality f, interacts with the honest servers, and learns the result as per the protocol at the end of the execution of Doc⋆. In other words, the real world executes Doc⋆ that reveals the result to the client and reveals the desired results of the tests to malicious servers.
- In the ideal world, an ideal functionality F_{Doc} realizes Doc⋆ with the desired security guarantees. The ideal world consists of the client and "non-malicious" servers sending their inputs to a third trusted party that computes the functionality and returns only the desired results to the client, without leaking any additional information. The functionality has access to the access matrix, inverted list, and files.

Ideal Functionality F_{Doc}. Below, we define the ideal functionality.
Initialization. F_{Doc} stores three data structures: an AC matrix, an inverted list, and file data structure (as explained in §5).
Query request for asking files having keyword uw. The client sends a request for a keyword uw with their identity. The functionality checks the access right for the keyword. If the client has access to search for the keyword, the functionality returns all the files having the queried keyword to the client. The client does not receive any file containing at least a single keyword to which they do not have permission to search.
Grant and revoke permission. If the DBO wishes to change the access rights to a client, the DBO sends the client-id along with 0 (if granting accesses) or random number (for revocation) to the functionality that takes action appropriately.

Now, we are ready to define the formal security property.

Theorem 4. *Let F_{Doc} be the ideal functionality. Let n be the number of servers that are connected over a secure point-to-point network among themselves and with the client. Let π be an n-server protocol for computing F_{Doc}. Each i^{th} server $\mathcal{S}_i$ holds input data and executes the protocol on the inputs. Let $\mathsf{F} \subset \{f_1, ..., f_f\} \subseteq [n]$, be the set of static malicious adversaries.*

The execution of π under F and a probabilistic adversary Adv in the real world is denoted by $real_{\pi,\mathsf{F},Adv}$.

The execution of F_{Doc} under F and a probabilistic algorithm of comparable complexity Sim (representing an ideal world adversarial strategy — comparable with Adv) in the ideal world is denoted by $ideal_{F_{Doc},\mathsf{F},Sim}$.

We say that π f-securely computes F_{Doc} under F of cardinality f if for every probabilistic algorithm Adv (representing a real would adversary strategy), there exists a probabilistic algorithm of comparable complexity Sim, the following hold

$ideal_{F_{Doc},\mathsf{F},Sim}$ and $real_{\pi,\mathsf{F},Adv}$ are indistinguishable.

The above theorem states that the adversary cannot distinguish between the real world and the ideal world.

In the ideal world, we define a coordinator or simulator *Sim* that interacts with the real adversary, *i.e.*, F servers, and the functionality. The simulator also provides an interface for the adversary to interact with "virtual" honest servers that do not have real input data. If the above theorem holds to be true, then the adversary cannot distinguish between its interaction/working in the real world's servers and the ideal world's simulator.

As per the discussion given in [30] on the security of the addition and multiplication operations, which are also involved in Doc⋆, over SSS, intuitively, Doc⋆ is f-private because the values that the F servers will see during the computation of each phase are random, *i.e.*, unrelated to any dataset, and eventually, *Sim* can generate the output of the tests at F servers whatever *Sim* wants, *i.e.*, unrelated to the random data and computation at F servers. Now, we need to describe the simulator *Sim*.

The Simulator, *Sim*. Below, to make proof simple, we first discuss the case, where the client and $n \backslash \mathsf{F}$ servers are honest. Here, the job of *Sim* is to "simulate" messages to the adversary that are similar to what the client and honest servers send to the malicious server in the real world. However, *Sim* can only interact with the malicious server and F_{Doc}. Also, *Sim* does not know the actual real inputs at the honest server.

- **Inputs.** *Sim* learns from the functionality F_{Doc} certain information about the query keyword *sw* and knows the schema of access control matrix, inverted list, and files — the number of clients in AC matrix, the number of searchable keywords in AC matrix, the size of each entry in the inverted list, the number of entries in the inverted list, the number of files, the size of files, and the maximum number of files associated with a keyword.
- **Simulation.**
  - **Initialization.** Without loss of generality, we assume that the server $\mathcal{S}_4$ is a malicious server, *i.e.*, $\mathsf{F} = \{\mathcal{S}_4\}$, and thus, f=1. Based on the inputs received from F_{Doc}, *Sim* creates fake shares of the access matrix, inverted list, and files, and such shares are given to $\mathcal{S}_4$. Note that at this time, only the malicious server has data.
  - **Access a file.** On receiving a query for a keyword *sw* from F_{Doc}, *Sim* executes the three phases. Below, let us see how *Sim* will execute Phase 1 and Test A in Phase 2, and the other phases and tests will be executed in the same manner.

    In Phase 1, *Sim* receives the query keyword *sw* and the client identity from F_{Doc}. Then, *Sim* generates f=1 fake/dummy shares of the query keyword *sw* and sends them to malicious server $\mathcal{S}_4$. Note that at this time, the remaining three servers do not have any share of the query keyword as well as any share of the data. $\mathcal{S}_4$ executes Phase 1 on fake data of AC matrix and returns fake results to *Sim*. At this time, $\mathcal{S}_4$ is not aware of the computation at other servers. Since *Sim* knows the query keyword, *Sim* can execute the query using F_{Doc} and provide the correct answer of Phase 1 to the client, nothing to $\mathcal{S}_4$.

    In Phase 2, *Sim* can again generate a fake vector for the dot product operation at $\mathcal{S}_4$. This vector can have either all zeros, all ones, or any random numbers. Here, Phase 2 will execute Tests 1, A, B, C. Let us see Test A, *i.e.*, $\mathbb{M}(test_A) \leftarrow \sum \mathbb{M}(v)$. Here, $\mathcal{S}_4$ will do addition on the vector and return the result to *Sim*. At this time, for the interpolation of the result of Test A, *Sim* needs to provide additional shares, corresponding to "virtual" honest servers, to $\mathcal{S}_4$. Here, *Sim* will create a polynomial of degree f for each of the "virtual" honest servers, such that the constant of the polynomial is either 0 or a random number, depending on what *Sim* wants.

    Other steps of Phase 2 and Phase 3 will be simulated in an identical way to Phase 1, and all the tests will also be simulated identically to Test A.

In the simulated world, we need to show two things: first, the data at the F malicious server does not enable the adversary to distinguish between real and fake data/world, and second, the computation output (particularly the tests) does not reveal anything to the adversary.

In the initialization, the adversary receives f=1 random values in AC matrix, inverted list, and files. Due to the properties of SSS, the shares of a secret are uniformly distributed at random in the finite field, and any subset of at most f shares does not reveal any information about the secret (see the proof of Claim 1 and Claim 2 in [30]). Thus, the adversary cannot learn any information from the stored data at F servers.

During the computation, the malicious server $\mathcal{S}_4$ provides either a list of β numbers in Phase 1, a row of the inverted list in Phase 2, a file in Phase 3, or the output of all tests, which will include one number or a vector. First note that whatever $\mathcal{S}_4$ will provide as the final answer to each phase does not reveal anything to $\mathcal{S}_4$, as storing only one share, which corresponds to nothing, due to the initialization phase. Now, let us discuss the outputs of all the tests, which are revealed to each server. Since *Sim* knows the answer to the query keyword using F_{Doc}, based on that *Sim* can simulate the entire output of the tests at malicious servers as *Sim* wants by the following steps: Recall that the malicious servers hold only f shares, which are also known to *Sim* due to the communication involved in the tests for result interpolation. At this time, *Sim* knows the f shares from f malicious servers and know the output of the tests as per the desire of *Sim*. Thus, *Sim* can generate shares corresponding to "virtual" honest servers and send them to the f malicious servers. On interpolation, the malicious servers will obtain the answer that *Sim* wants. This is just the outputs that correspond to the outputs of the malicious servers, and nothing else. Thus, based on the computation output, f malicious servers do not learn additional information.

Combining the above two arguments creates a view that is indistinguishable from the real world's protocol execution. ■

G HIDING TEST 1 OUTPUT FOR INCORRECT VECTOR AND DEGREE REDUCTION

Recall that in Test 1, a malicious client can learn the value of non-access by creating a wrong vector in STEP 3B.

$$\text{Test 1: } \mathbb{M}(test_1) \leftarrow \mathbb{M}(AC) \odot \mathbb{M}(v)$$

If we want to hide non-access value, *i.e.*, the output of Test 1, we can modify Test 1 as follows:

$$\text{Modified Test 1: } [DegR(\mathbb{M}(AC) \odot \mathbb{M}(v))] \times \mathbb{M}(\mathsf{RN}).$$

Here, $\mathcal{S}_z$ first reduces the degree, denoted by $DegR(*)$, of the output of the dot product, then multiplies a random number, and finally, sends the output of the test to other servers. Then, each server interpolates the values. If the vector $\mathbb{M}(v)$ is correct, then the output of modified test 1 will be 0; otherwise, a random number, which will be different from the non-access value.

Degree reduction. For degree reduction of a value, say $\mathbb{M}(obj)$, of degree two to degree one, we do the following

1. $\mathbb{M}(op) \leftarrow \mathbb{M}(obj) + \mathbb{M}(\mathsf{RN})$, where $z = 1, 2, 3$ and $\mathbb{M}(\mathsf{RN})$ is of degree one.
2. $\mathcal{S}_2$ and $\mathcal{S}_3$ sends $\mathbb{M}(op_2)$ and $\mathbb{M}(op_3)$to $\mathcal{S}_1$.
3. $\mathcal{S}_1$ interpolates $op \leftarrow \mathsf{Interpolate}(\mathbb{M}(op_1), \mathbb{M}(op_2), \mathbb{M}(op_3))$.
4. $\mathcal{S}_1$ creates shares of op, denoted by op' using a polynomial of degree one.
5. $\mathcal{S}_1$ sends shares to $\mathcal{S}_2$ and $\mathcal{S}_3$.
6. $\mathbb{M}(obj'_z) \leftarrow \mathbb{M}(op'_z) - \mathbb{M}(\mathsf{RN})$, where $z = 1, 2, 3$, $\mathbb{M}(\mathsf{RN})$ is of degree one used in step 1, and $\mathbb{M}(obj')$ is of degree one.