Access Control for Information-Theoretically Secure Data

Yin Li
Dongguan University of Technology, China

Sharad Mehrotra
University of California, Irvine, USA

Shantanu Sharma
New Jersey Institute of Technology, USA

Komal Kumari
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.

PVLDB Reference Format:
Yin Li, Sharad Mehrotra, Shantanu Sharma, and Komal Kumari. Access Control for Information-Theoretically Secure Data. PVLDB, 18(10): 3243 - 3255, 2025.
doi:10.14778/3748191.3748192

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SecretDeB/DocStar-VLDB-2025.

1 INTRODUCTION

While secure data outsourcing and query over encrypted data has been widely studied over the past two decades [32, 42], 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 [1] 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 [2], 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 [24, 25]. 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

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:10.14778/3748191.3748192

[1] DOC* uses a policy model that permits only explicitly allowed actions, due to its stronger security guarantees as discussed in [40], 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.

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 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 previous literature [24, 39, 45] and the (simpler-version of the) solution we develop can also be applied to key-level access control in KV stores.

We develop an access control mechanism, entitled Doc* (where $\star$ 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 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 [3, 4]; *computational efficiency* to perform addition and multiplication on ciphertext; avoiding the risk of a single point of failure [5]; avoiding the risk of data theft [6, 7]; and tolerating malicious servers. Secret-sharing has gained popularity for data processing [8, 18, 20, 30, 43] and multi-party computation [22, 23, 27, 33, 37] by both academia and industries. While we develop our access control mechanism using secret-sharing for outsourcing, homomorphic encryption [9], which also offers addition and multiplication over ciphertexts, can also be used.

The full version of the paper is available in [10] and includes appendices on : (*i*) dynamic operations–adding/deleting new files with new/existing keywords, (*ii*) access grant/revocation, (*iii*) optimization methods for Phase 2, (*iv*) methods for enhancing Phase 3, (*v*) client-side result verification methods, (*vi*) formal security proofs with proofs of theorems given in this paper, (*vii*) extension of Doc* for attribute- and file-id-based access control, (*viii*) degree reduction method, and (*ix*) 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 [41]. 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 $a_i{\in}\mathbb{N}$ $(1{\le}i{\le}c')$. 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 [26] 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 by Ben-Or et.al [21]). SSS is *additive homomorphic* [19]: the sum of shares at servers produces the sum of cleartext value after interpolation. SSS is, also, *multiplicative homomorphic* [19]: 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. Note that such documents may contain other keywords for which a

[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 [11–15]. Leveraging multi-cloud environments enables organizations to outsource shares more effectively.

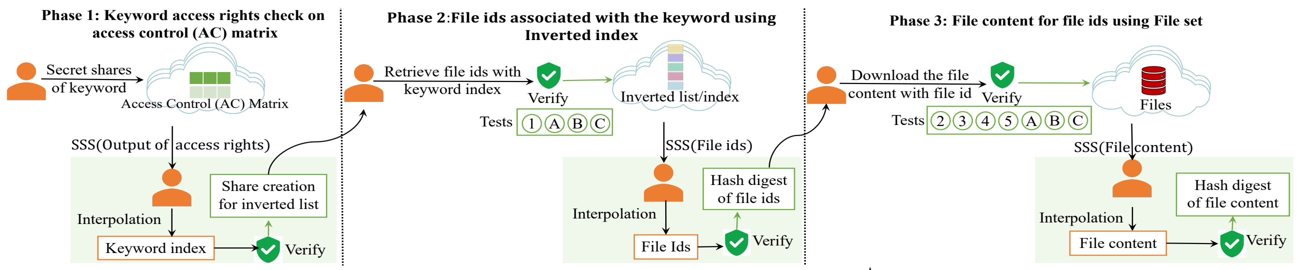

Phase 1: Keyword access rights check on access control (AC) matrix
Secret shares of keyword
Access Control (AC) Matrix
SSS(Output of access rights)
Interpolation
Share creation for inverted list
Keyword index
Verify
Phase 2:File ids associated with the keyword using Inverted index
Retrieve file ids with keyword index
Verify
Inverted list/index
Tests ① Ⓐ Ⓑ Ⓒ
SSS(File ids)
Interpolation
Hash digest of file ids
File Ids
Verify
Phase 3: File content for file ids using File set
Download the file content with file id
Verify
Files
Tests ② ③ ④ ⑤ Ⓐ Ⓑ Ⓒ
SSS(File content)
Interpolation
Hash digest of file content
File content
Verify

Figure 1: Execution of three phases of Doc★.

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 keyword King (indicated by $\mathbb{M}(0)$), not Horror (indicated by a random number). All values except client name in AC matrix are secured using SSS, (method is explained below §5). Second, DBO outsources an ***inverted list/index*** using SSS, see below, where the gray part is not outsourced to cloud and is written to keyword. Finally, DBO outsources all files using SSS.

Access control (AC) matrix.

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

Inverted index.

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

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 ⟨0, random number, . . .⟩, 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 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

Table 1: Frequently used notations in this paper.

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

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 [35].

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 [35] 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 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 [35] uses the equality test of [38]. 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 [21] 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:

Table 2: Cleartext files.

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

Table 3: Access control (AC) matrix.

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 4: Inverted index.

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 5: Files.

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 6: Share1 of AC matrix.

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 7: Share2 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 8: Share3 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 9: Three shares of inverted index.

File-ids	File-ids	File-ids
2, 3, $\mathbb{M}(x_1)$	3, 4, $\mathbb{M}(y_1)$	4, 5, $\mathbb{M}(z_1)$
3, 1, $\mathbb{M}(x_2)$	4, 2, $\mathbb{M}(y_2)$	5, 3, $\mathbb{M}(z_2)$
4, 1 $\mathbb{M}(x_3)$	5, 2 $\mathbb{M}(y_3)$	6, 3, $\mathbb{M}(z_3)$

(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 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 full version [10]. Finally, DBO adds a dummy/fake file that is used to hide the volume from servers during Phase 3 of file retrieval.

[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: $\mathsf{H}(f_2, (\mathsf{H}(f_1, \mathsf{H}(\text{keyword}))))$, where H is a secure hash function [29]. 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 full version [10]).

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.

[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.

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 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 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

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 $\mathbb{M}(\mathbf{0})$ ***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 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.

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

[7] A well-known method for polynomial randomization is BGW [21]. 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.

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 other servers. On receiving shares from other servers, $\mathcal{S}_z$ aggregates them. $\mathbb{M}(\mathsf{RN})[]$ denotes such random numbers in share form at each server. The value of q could be the number of keywords in AC matrix if random numbers are used in Phase 1 or the size of a file if used in Phase 3.

Meanwhile, $\mathcal{S}_z$ generates $\mathbb{M}(0)$ ***of degree two*** by multiplying $\mathbb{M}(\mathsf{RN})$ 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 $\mathbb{M}(\mathsf{RN})$).

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

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 $\mathbb{M}(\mathsf{RN})$. 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 6.1. ***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*** $LI(*)$ ***is 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:

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

$\mathcal{S}_{z\in\{1,2,3\}}$ 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 $\mathbb{M}(\mathsf{RN})[]$ 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.

Table 10: Example of Phase 1.

`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.

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 6.2. ***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 6.3. ***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).

Table 11: Example of Phase 2.

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:

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

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.

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 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 $\underline{\textit{correctness}}$ 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 6.3*. Of course, we can also hide this value too, if it is required using the method given in full version [10].
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 $O(\gamma)$.

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.

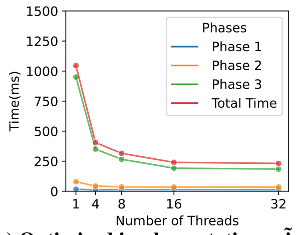
For example, Lisa, who searched for the keyword 'Are' during Phase 1, learned that files 1 and 2 both match the keyword 'Are' after Phase 2. In Phase 3, the server returns to her the actual file 1, as well as a fake file in place of file 2. Although both files contain the keyword 'Are,' file 2 also contains 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.

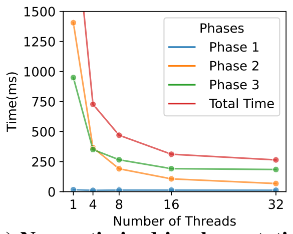
Algorithm design objective. An algorithm in Phase 3 needs to:
(*i*) *Prevent sending an unauthorized file to a client.* A client must not obtain a file F that contains a keyword k that the client has searched in Phase 1 for which the client has allowed search access, and the file F also contains a keyword 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.

[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 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.

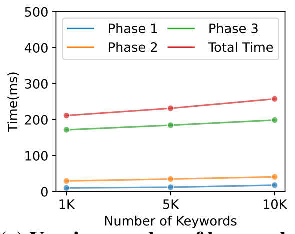


(a) Optimized implementation using OptInv and AddrList.

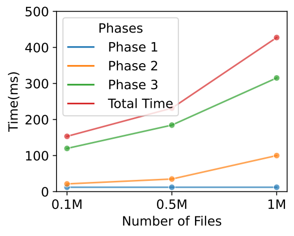


(b) Non-optimized implementation using an inverted list.

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



(a) Varying number of keywords.



(b) Varying number of files.

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

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, if a keyword appears in $x<\gamma$ files, then the $\gamma - x$ vectors will fetch a fake file with zeros.

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.[11]

Once the server verifies all the conditions, the server will send the file obliviously, *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.

STEP 7: ***Client:*** interpolates the file content, obtained from servers.

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:

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

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.

[11] Due to space limitations, we do not provide full details of Phase 3 in this version of the paper.

7 OPTIMIZATION OF THE INVERTED INDEX

The method of §6.3 for Phase 2 has the 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). E.g., 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 vast size inverted list results in the overhead. Due to space restrictions, below, we provide only the intuition of how we reduce the space and computational overheads during Phase 2. To do so, we develop an algorithm that is based on the following two new data structures that DBO outsources using SSS.

(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.

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.

Phase 1 (STEP 1-STEP 3A) is executed on AC matrix without any modification. After executing Phase 1, the client learns which row of AddrList needs to be fetched. Then, Phase 2 will be executed on AddrList (Table 12) and OptInv (Table 13). Particularly, in Phase 2, the client fetches a desired row (that results in SiP, CuT, and HD) of AddrList. Note that the servers keep HdV value of the returned row, for verification of the client vector at a later stage. After this, the client fetches γ file-ids from the server based on SiP and CuT. The algorithm that DOC* develops ensures that (*i*) the client fetches the desired file-ids from OptInv for the keyword they have searched in Phase 1, without revealing access-patterns and volume to servers, and (*ii*) the servers obliviously return γ file-ids, regardless of a query, such that the client learns only the file-ids associated with the keyword; nothing else, where γ be (a publicly known) the maximum number of file-ids associated with a keyword. Finally, Phase 3 will be executed on the desired file-ids that are associated with the keyword that the client has searched in Phase 1.

Table 12: Cleartext AddrList array.

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

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

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) Performance of DOC★ in a wide-area network — Exp 5.
(6) Time taken by DOC★ and other systems — Exp 6.

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 [16], a commonly used real data to evaluate techniques for document stores. This *dataset was also used in other recent document store work [24, 25, 28]*. 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 14 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 [24, 25].

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

Table 14: Exp 1: Size of the shares at a single server.

	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

of the inverted list (due to the absence of padding; as shown in Table 14). Table 15 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 15 shows the variation of time in different phases.

Table 15: Exp 2: Time on 5K keywords, 500K files, 32 threads.

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

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).

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*.

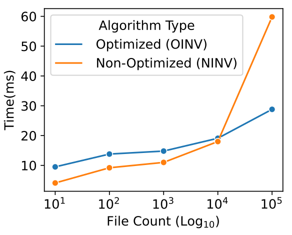


Figure 4: Exp 4: Different implementations of Phase 2.

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 client and transmission time (1.3ms), while OINV took 12.1ms processing time and 2.4ms for transmission. Here, the size

Table 16: 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	**Sieve [44]**	**Ghoster [34]**	**Dory [28]**	**Metal [24]**	**Titanium [25]**	**Doc★**
Offered security	Computational security					Unconditional security
Cryptographic techniques	Encryption	Encryption	Encryption	Encryption & binary shares	Additive shares & trusted proxy for access control (§7 of [25])	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**
Trusted proxy for access control	**No**		N/A	**No**	**Yes**	**No**
Complexity of granting access	○	○ out-of-band	N/A	● out-of-band	○	○
Complexity of revoking access	●	●	N/A	◐	○	○
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 17: Exp 6: Comparing Doc★ & other systems on 1 thread.

Systems	**MySQL**	**Baseline**	**Doc★-Leaky**	**Doc★-Secure**	**Secrecy [36]**	**Dory [28]**
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

of 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.*

Exp 5: 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 enables full utilization of the bandwidth between two different data centers.

Additional experiments in [10] provide: (*i*) different implementations of AP list in Phase 3, (*ii*) granting/ revoking access rights, (*iii*) adding new file-id(s), (*iv*) deleting an existing file, (*v*) deleting an existing keyword, and (*vi*) increasing the number of servers.

8.2 Comparing Doc★ against Other Systems

Exp 6: Time taken by Doc★ and other systems. We compare Doc★ against different systems on 5K keywords in 500K files: (*i*) ***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. (*ii*) ***The baseline method***: is given §4. (*iii*) ***Secrecy [36]***: is a secret-sharing (SS)-based relational-data processing system (not a secure document storage system). 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 $O(\beta)$. (*iv*) **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. (*vi*) ***Dory [28]***: 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 [31] 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★. (*vii*) ***Metal [17, 24] and Titanium [25]***: are recent conditionally secure document storage systems, code of these systems is not available. Table 16 compares these systems on different parameters. Table 17 shows the time for different systems. *For this experiment, we do not include the time to fetch a file, since MySQL and Dory do not support such an operation.*

9 CONCLUSION

We develop a secure key-based access control technique for secret-shared document storage. Operations at servers hide access-patterns and volume, and do not reveal any information to servers about the data. Our technique takes 231.5ms over 5K keywords in 500K files.

ACKNOWLEDGMENTS

We are thankful to Murat Kantarcioglu for his feedback on 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] Redis Key-Value Store: https://redis.io/.

[2] Redis Access Control List: https://tinyurl.com/bdhcc7m6.

[3] Survey: Insider Threats Surge Across U.S. Critical Infrastructure. Available at: https://tinyurl.com/mry3hmy7.

[4] NSA Moves to Prevent Snowden-Like Leaks. Available at: https://tinyurl.com/3xsvnpea.

[5] MySpace lost 13 years worth of user data after botched server migration?. Available at: https://tinyurl.com/2b6vjs27.

[6] LastPass Reveals Second Attack Resulting in Breach of Encrypted Password Vaults. Available at: https://tinyurl.com/4evra455.

[7] CrowdStrike: Attackers focusing on cloud exploits, data theft. Available at: https://tinyurl.com/3hvr2nd2.

[8] Jana: Private Data as a Service. Available at: tinyurl.com/47jemma5.

[9] Microsoft Seal: https://github.com/microsoft/SEAL.

[10] Code, data, and the full version of the paper: https://github.com/SecretDeB/DocStar-VLDB-2025.

[11] How Many Companies Use Cloud Computing in 2022? All You Need To Know. Available at: https://tinyurl.com/2p983aau.

[12] Multi-Cloud Data Solutions for Today (and Tomorrow). Available at: https://tinyurl.com/2v2bvymp.

[13] Multicloud. Available at: https://www.ibm.com/cloud/learn/multicloud.

[14] Multi-cloud mature organizations are 6.3 times more likely to go to market and succeed before their competition. Available at: tinyurl.com/2ym8xcx7.

[15] More and more companies are spreading their data over public clouds. Available at: https://tinyurl.com/46fph54z.

[16] Enron Email Dataset. Available at: http://www.cs.cmu.edu/ enron/.

[17] https://www.oblivious.app/.

[18] D. W. Archer et al. From keys to databases - real-world applications of secure multi-party computation. *Comput. J.*, 61(12):1749–1771, 2018.

[19] G. Asharov and Y. Lindell. A full proof of the BGW protocol for perfectly-secure multiparty computation. *IACR Cryptol. ePrint Arch.*, page 136, 2011.

[20] J. Bater et al. SMCQL: secure query processing for private data networks. *PVLDB*, 10(6):673–684, 2017.

[21] M. Ben-Or et al. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.

[22] D. Bogdanov et al. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In *NordSec*, pages 59–74, 2014.

[23] R. Canetti et al. Adaptively secure multi-party computation. In G. L. Miller, editor, *STOC*, pages 639–648, 1996.

[24] W. Chen et al. Metal: A metadata-hiding file-sharing system. In *NDSS*, 2020.

[25] W. Chen et al. Titanium: A metadata-hiding file-sharing system with malicious security. In *NDSS*, 2022.

[26] R. M. Corless et al. A graduate introduction to numerical methods. *AMC*, 10:12, 2013.

[27] I. Damgård et al. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, pages 285–304, 2006.

[28] E. Dauterman et al. DORY: an encrypted search system with distributed trust. In *OSDI*, pages 1101–1119, 2020.

[29] M. 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.

[30] F. Emekçi et al. Dividing secrets to secure data outsourcing. *Inf. Sci.*, 263:198–210, 2014.

[31] N. Gilboa et al. Distributed point functions and their applications. In *EUROCRYPT*, volume 8441, pages 640–658, 2014.

[32] H. Hacigümüs et al. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.

[33] K. Hamada et al. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *ICISC*, pages 202–216, 2012.

[34] Y. Hu et al. Ghostor: Toward a secure data-sharing system from decentralized trust. In *NSDI*, pages 851–877, 2020.

[35] M. Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*, 2020.

[36] J. Liagouris et al. SECRECY: secure collaborative analytics in untrusted clouds. In *NSDI*, pages 1031–1056, 2023.

[37] E. Makri et al. Rabbit: Efficient comparison for secure multi-party computation. In *FC*, pages 249–270, 2021.

[38] T. Nishide et al. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC*, pages 343–360, 2007.

[39] E. Pattuk et al. Bigsecret: A secure data management framework for key-value stores. In *International Conference on Cloud Computing*, pages 147–154, 2013.

[40] J. H. Saltzer et al. The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308, 1975.

[41] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[42] D. X. Song et al. Practical techniques for searches on encrypted data. In *IEEE SP*, pages 44–55, 2000.

[43] N. Volgushev et al. Conclave: secure multi-party computation on big data. In *EuroSys*, pages 3:1–3:18, 2019.

[44] F. Wang et al. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *NSDI*, pages 611–626, 2016.

[45] X. Yuan et al. Secure multi-client data access with boolean queries in distributed key-value stores. In *CNS*, pages 1–9, 2017.