Delft University of Technology

# Keyword Search Shareable Encryption for Fast and Secure Data Replication

Wang, Wei; Liu, Dongli; Xu, Peng; Yang, Laurence Tianruo; Liang, Kaitai

**DOI**
10.1109/TIFS.2023.3306941

**Publication date**
2023

**Document Version**
Final published version

**Published in**
IEEE Transactions on Information Forensics and Security

**Citation (APA)**
Wang, W., Liu, D., Xu, P., Yang, L. T., & Liang, K. (2023). Keyword Search Shareable Encryption for Fast and Secure Data Replication. *IEEE Transactions on Information Forensics and Security*, *18*, 5537-5552. https://doi.org/10.1109/TIFS.2023.3306941

**Important note**
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

**Copyright**
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

**Takedown policy**
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

*This work is downloaded from Delft University of Technology.*
*For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.*

*Green Open Access added to TU Delft Institutional Repository*

*'You share, we take care!' - Taverne project*

*https://www.openaccess.nl/en/you-share-we-take-care*

Otherwise as indicated in the copyright section: the publisher is the copyright holder of this work and the author uses the Dutch legislation to make this work public.

# Keyword Search Shareable Encryption for Fast and Secure Data Replication

Wei Wang, *Member, IEEE*, Dongli Liu, Peng Xu, *Member, IEEE*, Laurence Tianruo Yang, *Fellow, IEEE*, and Kaitai Liang, *Member, IEEE*

***Abstract*— It has become a trend for clients to outsource their encrypted databases to remote servers and then leverage the Searchable Encryption technique to perform secure data retrieval. However, the method has yet to be considered a crucial need for replication on searchable encrypted data. It calls for challenging works on Dynamic Searchable Symmetric Encryption (DSSE) since clients must share the search capability of the encrypted data replicas and guarantee forward and backward privacy. We define a new notion called "Keyword Search Shareable Encryption" (KS$^2$E) and the corresponding security model capturing forward and backward privacy. In our notion, data owners are allowed to share search indexes of the encrypted data with users. A search index will be updated with a new search key before sharing to guarantee the data privacy of the source database. The target database also inherits data search efficiency along with the shared data. We further construct an instance of KS$^2$E called *Branch*, prove its security, and use real-world datasets to evaluate Branch. The evaluation results show that Branch's performance is comparable to classical DSSE schemes on search efficiency and demonstrate the effectiveness of searching encrypted data replicas from multiple owners.**

***Index Terms*— Searchable symmetric encryption, forward and backward privacy, encrypted data replication.**

## I. Introduction

In 2012, Kamara et al. developed Dynamic Searchable Symmetric Encryption (DSSE) [1] to bring forward/backward privacy attention to newly updated retrieval entries on historical/future search results [2]. DSSE schemes usually employ search indexes to retrieve logical file addresses, and they should update search indexes along with the update of ciphertexts. Research works on DSSE consider data updating as adding or deleting outsourced encrypted data on a server via maintaining a solo search index table [3]. Besides, the high efficiency of DSSE also attracts researchers to extend the search function from a single client to multiple clients. This advantage avoids the fundamental efficiency bottleneck in public encryption schemes like PEKS [4].

Multi-Key Searchable Encryption (MKSE) [5] can be applied as a heuristic ciphertext-sharing approach, which enables search tokens to search shared ciphertexts via the adjust-key that can re-encrypt the search token from one search key to another search key. In our view, the search capability relies on the corresponding search key other than the client. Moreover, [6] additionally uses shared data information as the input of sharing process in the Multi-Key Searchable Encryption scheme. We call such key-sharing processes for searching shared ciphertexts "*static*" methods since this scheme predefines the usage of outsourced ciphertexts. Inspired by the DSSE research of adding and deleting data along with search indexes, we consider that sharing processes may need variability on data users and ciphertexts in "*dynamic*" ways. Hence, we seek efficient schemes that can search the shared data via search indexes without long-term maintenance of auxiliary keys, which is desirable in realistic scenarios:

### A. Data Backup

Inter-cloud data replication [7], [8] is an essential operation for backup, even though the data are stored as ciphertexts. The data owner delivers replicas to the nearest backup database, and the replicas will be re-encrypted with the key of the backup database. The indexes associated with the data play a critical role in determining the performance of search queries [9]. In this example, sharing search indexes can help clients to achieve a natural efficiency in securely searching encrypted replicas authorized by the data owner.

### B. Health Monitoring

Electronic health record (EHR) data [10] are usually encrypted to guarantee privacy. When patients are transferred among different health centers, their EHR should be replicated and shared with doctors automatically. Recall that the replication is encrypted with patients' private keys. Without those keys, the doctors should be able to find a way to put the encrypted records into the local database but also to search them correctly and efficiently.

Manuscript received 17 October 2022; revised 4 March 2023 and 10 June 2023; accepted 3 August 2023. Date of publication 21 August 2023; date of current version 8 September 2023. The work of Wei Wang was supported in part by the National Key Research and Development Program of China under Grant 2021YFB3101304 and in part by the National Natural Science Foundation of China under Grant 62372201. The work of Peng Xu was supported in part by the National Natural Science Foundation of China under Grant 62272186. The work of Kaitai Liang was supported by the European Union's Horizon Europe Research and Innovation Programme under Grant 101073920 (TENSOR), Grant 101070052 (TANGO), and Grant 101070627 (REWIRE). The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Sushmita Ruj. *(Corresponding author: Peng Xu.)*

Wei Wang and Dongli Liu are with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: viviawangwei@hust.edu.cn; nsffldl@hust.edu.cn).

Peng Xu is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Laboratory, Big Data Security Engineering Research Center, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430073, China (e-mail: xupeng@mail.hust.edu.cn).

Laurence Tianruo Yang is with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China, and also with the Department of Computer Science, St. Francis Xavier University, Antigonish, NS B2G 2W5, Canada (e-mail: ltyang@ieee.org).

Kaitai Liang is with the Department of Intelligent Systems, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: kaitai.liang@tudelft.nl).

Digital Object Identifier 10.1109/TIFS.2023.3306941

1556-6021 © 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.

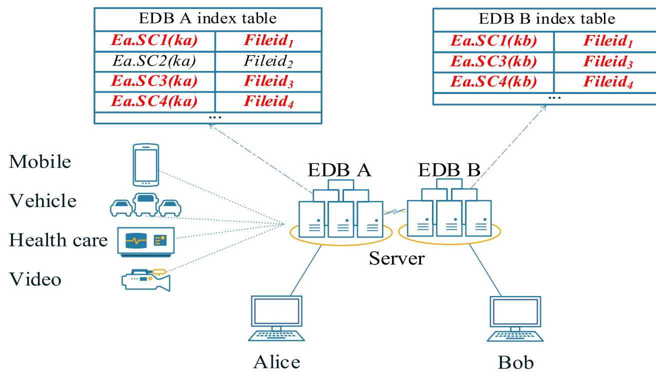



Fig. 1. The generic scenario of secure data replication. Alice and Bob outsource EDB A and EDB B to Server, respectively. The highlighted search indexes are shared from Alice to Bob. In the index table, $Ei.SCj(k_i)$ denotes the $j$th searchable ciphertext encrypted by $k_i$ from EDB $i$, and $Fileid_i$ denotes the corresponding logical file address $id_i$.

Fig. 1 illustrates this scenario. Alice and Bob use separate encrypted databases and search indexes, but they need to exchange data. When the data is shared from Alice to Bob, Alice takes the owner role while Bob is the user. Firstly, the server sends the encrypted data from Alice's database to Bob as the replication. Secondly, Bob derives the search capability over the shared data. To this end, Bob should derive the search index for the replica.

*1) Motivation:* Most prior works *predefine* sharing keys to establish long-term sharing relationships. For example, MKSE [5], [6] delegates the search capability by adjust-keys. An adjust-key is predefined to re-encrypt the search tokens from the user to the owner on the server side. Multi-user Searchable Encryption (MUSE) [11], [12], [13] allows multiple users to search an owner's encrypted data securely. To do so, the owner must enroll the keys of the users in advance. We conclude the characteristic of MKSE and MUSE in Tab. I. Key-Aggregate Searchable Encryption (KASE) [14] uses aggregate-keys to share the search capability over encrypted ciphertexts. An aggregate-key enables re-encrypt search tokens to be generated from the keywords specified by the owner. Hybrid Searchable Encryption (HSE) [15] combines the cryptographic primitives of DSSE and ID-Coupling Key-Aggregate Encryption (ICKAE). This scheme prepares a set of initialized keys for owners who can use the keys to update their local database, in which the updated ciphertexts are related to the user's public key.

Considering scalability and search efficiency, Alice and Bob should dynamically and securely share search indexes with a *temporary* sharing key. We further observe the followings:

- For dynamic sharing, it is impractical to preset any sharing keys beforehand for each dynamic request of search indexes.
- For securely sharing, the sharing process should not cause serious leakages on the previously updated search index, which is not shared, or the later updated search index.
- For the efficiency of search and index sharing, the search complexity after sharing should rely on the number of keywords.

*2) Our Contributions:* Inspired by the primitives of DSSE, we propose a novel scheme called ***K**eyword **S**earch **S**hareable **E**ncryption* (KS$^2$E) to achieve our goals. KS$^2$E is applied to construct the outsourced encrypted database with shareable search indexes authorized by *sharing tokens*. The owner and user temporarily negotiate the sharing tokens to "snapshot" search indexes. The "snapshot" is achieved by a specific structure among encrypted search indexes, and the internal state generates this structure. In this approach, the sharing token represents the search indexes to be shared in the current internal state, and Server can retrieve replicas by shared search indexes with a search token. The main contributions are summarized as follows:

- We define the KS$^2$E scheme. KS$^2$E has functions of searching and sharing indexes, and it differs from other models in the sharing process or the solo search index table. We analyze potential threats and security risks accordingly by redefined leakage functions and semantic security from the indistinguishability between real and ideal games, i.e., simulation-based security.
- To instantiate KS$^2$E, we construct an instance called *Branch*. Branch uses cryptographic hash functions and secure Pseudo Random Functions (PRFs) to generate search indexes, sharing tokens, and search tokens. Branch generates a bi-directional structured search index to achieve sub-linear efficiency in the search processes. We prove that Branch has adaptive security under our security model.
- We code Branch in Python and simulate it in the inter-cloud data replication circumstances with the leverage of real-world datasets to perform efficiency analysis. We compare Branch with the heuristic KS$^2$E instance based on a DSSE instance Diana [3], an MKSE instance constructed from Perfect Hash Table [6], and a MUSE instance Q-$\mu$SE [11]. We thoroughly evaluate time overheads on actual sharing performance. Branch achieves sub-linear efficiency according to its sharing and search performances. As shown in Tab. I, the bi-directional index brings KS$^2$E significant advantages in search time complexity as compared to MUSE and MKSE.

*3) Organization:* Section II introduces the related work on Searchable Symmetric Encryption (SSE) schemes. Section III introduces the KS$^2$E model. Section IV proposes KS$^2$E's instance called *Branch*. Section V presents our evaluation results. Section VI concludes this paper.

## II. Related Works

The first SSE scheme [16] proposes a secure system that designs search tokens to retrieve ciphertexts. Inspired by that, researchers have delivered massive works on designing secure and fast SSE schemes in the single-user setting, which are applicable to many real-world applications [17], [18]. Other works investigate how to securely share the search capability in the context of SSE. None of the prior SSE schemes has considered the followings: (1) data owner and data users may probably have different setups, initializing respective private search keys; (2) encrypted database could have replications, and search is required over those replicas.

### A. Dynamic Searchable Symmetric Encryption (DSSE)

DSSE [1], [19], [20], as compared to traditional SSE, provides the addition and deletion of search indexes via update queries. In 2017, Bost et al. [3], [21] proposed the forward and backward privacy notions for DSSE. Forward privacy forbids implying new *add* contents to current search results, while backward privacy is used to securely exclude previous

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.

TABLE I

COMPARISON OF PREVIOUS REPRESENTATIVE WORKS AND OUR WORKS. $a_s$ REPRESENTS ALL SHARING KEYS. $a_w$ REPRESENTS ALL KEYWORD/IDENTIFIER PAIRS $(w, id)$ ABOUT $w$. |DIFF| REPRESENTS THE SIZE OF THE DIFFERENCE QUEUE. $n(I_w)$ AND $n(I_{id})$ REPRESENTS THE NUMBER OF KEYWORD $w$ AND FILE $id$ ITERATORS, RESPECTIVELY. $F^Q$ AND $B^Q$ REPRESENT THE MUSE'S FORWARD AND BACKWARD PRIVACY, RESPECTIVELY. $F^B$ AND $B^B$ REPRESENT THE KS$^2$E'S FORWARD AND BACKWARD PRIVACY, RESPECTIVELY

| Scheme | Unlimited users | Sharing Token | Computation | | | Communication | | | FP | BP |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Search | Share | Update | Search | Share | Update | | |
| PKC18 (MKSE) [6] | ✓ | ✗ | $O(a_s)$ | $O(1)$ | - | $O(a_s)$ | $O(1)$ | - | - | - |
| Q-$\mu$SE (MUSE) [11] | ✗ | ✗ | $O(a_w + \|\text{diff}\|)$ | $O(1)$ | $O(1)$ | $O(a_w + \|\text{diff}\|)$ | $O(1)$ | $O(1)$ | $F^Q$ | $B^Q$ |
| Branch (Ours) | ✓ | ✓ | $O(n(I_w))$ | $O(n(I_{id}))$ | $O(1)$ | $O(n(I_w))$ | $O(n(I_{id}))$ | $O(1)$ | $F^B$ | $B^B$ |

*del* content from search [3], [22], [23]. The definition acts as an essential security requirement in practical applications, such as non-interactive communication [24], small client storage [25], and efficient search [26]. Recently, several works have extended the security definitions to particular contexts, such as Robustness [27], multi-user [11], and document collections [28]. In this work, we refer to the forward and backward privacy as basic security notions of KS$^2$E, capturing the leakage from the addition and deletion of search indexes.

### B. Multi-User Searchable Encryption (MUSE)

Curtmola et al. [12] first proposed a multi-user SSE (MUSE) scheme where an arbitrary group of users can submit search queries for the owner's database. Following that work, many researchers have designed various variants covering security [29], [30], efficiency [31], non-interactive [32], and sharing revocation [13]. Recently, Chamani et al. [11] developed a new concept called Multi-user Dynamic Searchable Symmetric Encryption (DMUSE). They considered the leakage of the sharing upon the forward and backward privacy in the multi-user setting and proposed an OMAP-base $\mu$SE instance and a queue-base $\mu$SE instance.

Assigning sharing keys to users beforehand (by data owners) easily limits the dynamic search delegation. For example, the number of users needs to be pre-fixed [11], [13]. Besides, existing schemes supporting search sharing may consume more search complexity while multiple scheme instances are needed. Users leverage the sharing keys of all owners to compute the search tokens and search for a keyword in each MUSE instance. In KS$^2$E, we enable users to leverage private search keys to retrieve the shared data replication in their databases so that we do not have to limit the number of users in advance. We also ensure search efficiency regardless of the number of owners.

### C. Multi-Key Searchable Encryption (MKSE)

Popa and Zeldovich [5], [33] introduced the definition of MKSE. It enables users to search keywords over data encrypted with different keys. The first MKSE scheme [5] is based on elliptic curves with bi-linear mappings, but it is vulnerable to Leakage-Abuse attacks [34]. Later, Hamlin et al. [6] revisited the concept and explicitly defined a setting for MKSE where users who store the encrypted documents on a remote server can selectively share documents with each other. They proposed the indistinguishability-based security notions and two concrete constructions from a perfect hash function and an indistinguishability obfuscation, respectively. Recently, some works [35], [36] proposed alternative formulations to verify users' privileges.

In MKSE (e.g., [6]), we observe that a sharing key must be established for each encrypted file so as to delegate a search successfully. The search token can not directly match the search index, as it first needs to be computed with the sharing key. In KS$^2$E, the token is directly performed on search indexes, which results in much higher efficiency (see Section V).

## III. KEYWORD SEARCH SHAREABLE ENCRYPTION

In this section, we introduce the definitions and the generic construction of KS$^2$E for sharing search indexes during data replication. We also define the security model with forward and backward privacy.

### A. Problem Statement

We consider two parties:

- Clients: They initialize the system by a setup with security parameters and outsource the encrypted databases to a Server. A client can have two roles separately - owner and user. As an owner (Owner), it can send queries to update search indexes, delegate the sharing token to a user (User), and issue search queries to Server. User can receive the sharing token from Owner and send sharing requests and search queries to Server.
- Server: Server (which could be one or more server devices) manages the encrypted databases outsourced by clients. It further handles the requests to setup and update the databases, converts search sharing, and launches secure search over encrypted data. Upon receiving an update query, Server adds or deletes a search index from the corresponding database. Server also participates in sharing the index from Owner to User. It can use a search token to match the indexes to return the corresponding search results.

### B. Scheme Definitions

*Definition 1 (The KS$^2$E Scheme): A* KS$^2$E *scheme consists of the following protocols:*

- Setup$(\lambda)$*: This protocol is run between Owner and Server or User and Server for initialization by inputting the security parameter $\lambda \in \mathbb{N}$. When it is run between Owner and Server, it probabilistically outputs an internal state $\sigma$, master secret key $K_\sigma \in \mathcal{K}$ stored by Owner, and an encrypted database* $\mathrm{EDB}_\sigma$ *outsourced on Server. In another case, it probabilistically outputs an internal state $\sigma'$, master secret key $K_{\sigma'} \in \mathcal{K}$ stored by User, and an encrypted database* $\mathrm{EDB}_{\sigma'}$ *outsourced on Server.*
- Update$(K_\sigma, \sigma, (w, id, op); \mathrm{EDB}_\sigma)$*: This protocol is run between Owner and Server to update the search index in the encrypted database. Owner inputs $K_\sigma$, internal state $\sigma$, and a tuple of the keyword $w$, file address $id$, operation*

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.

*$op \in \{add, del\}$. Server inputs the encrypted database $\mathrm{EDB}_\sigma$. At the end of this protocol, it adds ($add$) or deletes ($del$) the search index about the pair $(w, id)$ in $\mathrm{EDB}_\sigma$.*

- $\mathsf{Search}(K_\sigma, \sigma, w; \mathrm{EDB}_\sigma)$*: This protocol is run between Owner and Server or User and Server to search the index. When it is run between Owner and Server, Owner inputs $K_\sigma$, $\sigma$, and a specific keyword $w$. Server inputs the $\mathrm{EDB}_\sigma$. At the end of this protocol, Owner gets the search result $S$ or $\perp$ (null output). In another case, User inputs $K_{\sigma'}$, $\sigma'$, and a specific keyword $w$. Server inputs the $\mathrm{EDB}_{\sigma'}$. At the end of this protocol, User gets the search result $S'$ or $\perp$.*
- $\mathsf{Sharetoken}(K_\sigma, \sigma, id)$*: This protocol is run between Owner and User to generate the sharing token $P_{id}$ which can access a specific set of search indexes. It takes as input the $K_\sigma$, internal state $\sigma$, and the shared file $id$ from Owner. Finally, it outputs $P_{id}$ to User.*
- $\mathsf{Share}(P_{id}, \sigma', K_{\sigma'}; \mathrm{EDB}_{\sigma'}, \mathrm{EDB}_\sigma)$*: This protocol is run between User and Server for sharing search indexes. User inputs sharing token $P_{id}$, internal state $\sigma'$, and the $K_{\sigma'}$. Server inputs the databases $\mathrm{EDB}_{\sigma'}$ and $\mathrm{EDB}_\sigma$. At the end of this protocol, $\mathrm{EDB}_{\sigma'}$ stores the shared search indexes.*

*Correctness: Except with negligible probability, the $KS^2E$ scheme is correct if the* Search *protocol returns correct results for the searched keyword and if the* Share *protocol shares correct search indexes for the accessed $P_{id}$. For formalism, we ignore the case where Owner attempts to add a file with an existing identifier or delete/edit with an identifier not present in EDB.*

### *C. The Generic Construction of* $\mathrm{KS^2E}$ *Scheme*

Fig. 2 illustrates our generic construction that includes four functions:

- *Setup:* Owner and User can separately run the protocol Setup with the security parameter $\lambda$. At the end, Owner stores a master secret key $K_\sigma$ and the internal state $\sigma$ locally, and outsources the encrypted database $\mathrm{EDB}_\sigma$ to Server. Similarly, User stores a master secret key $K_{\sigma'}$ and the internal state $\sigma'$ locally, and sends the $\mathrm{EDB}_{\sigma'}$ to Server.
- *Index Updating:* This function directly uses the protocol Update between Owner and Server to update search indexes. Let the file $id$ be associated with a set of keywords $\{w_1, w_2, \dots w_n\}$ and $C_{w,id}$ denote the encrypted search index. The two parties can run the protocol Update to update indexes $\{C_{w_i,id} | i \in [1, n]\}$ in $\mathrm{EDB}_\sigma$.
- *Sharing:* Owner shares search indexes with User. Firstly, Owner runs the protocol Sharetoken to generate the sharing token $P_{id}$ associated with the search indexes for User, where $P_{id}$ can be securely received (via a key encapsulation [37]). Secondly, User and Server run the protocol Share to generate the search indexes $\{C'_{w_i,id} | i \in [1, n]\}$. At last, the User's search indexes are stored in $\mathrm{EDB}_{\sigma'}$.
- *Search:* Owner and User can run the protocol Search to retrieve matched indexes, respectively. When Owner and Server run the protocol Search, Owner chooses a keyword $w_i \in \{w_1, w_2, \dots w_n\}$ and sends the search token $T_{w_i}$ to Server. Server makes result $S$ from the matched indexes in $\{C_{w_i,id} | i \in [1, n]\}$ and sends it back to Owner. We note that User can perform a similar Search with Server.

We consider that all clients are provided search capability at the initialization; User is allowed to obtain the shared index specified in the sharing stage by Owner. Unlike existing works, our construction does not require any sharing token in the search stage; instead, we reflect the sharing directly into the search index, leading to an efficient search process.

### *D. Security Model*

According to the fact that the server is always considered as honest-but-curious, Server should know the query content and the outsourced encrypted databases as little as possible. More precisely, the adversary tracks the queries from Owner and User, views the full transcripts of queries on Server, knows who initiates these transcripts, and wants to learn anything beyond some explicit leakages.

We typically capture the security model from the indistinguishability between *Real and Ideal Game* formulation [3], [38]. In the real game, the adversary runs all operations originally and observes all transcripts from her view. In the ideal game, the adversary runs all operations constructed from the set of simulator $\mathcal{S}$ with the leakage function $\mathcal{L}$ and gets all simulated transcripts from her view. Finally, the adversary decides on which game is running and distinguishes the real and ideal game.

*Definition 2 (Adaptive Security of $KS^2E$): Assume a $\mathrm{KS^2E}$ scheme is $\mathcal{L}$-adaptively secure iff for all sufficiently large security parameters $\lambda \in \mathbb{N}$ and PPT adversary $\mathcal{A}$, there is a set of efficient simulators $\mathcal{S}$ with a set of leakage functions $\mathcal{L}$ that has*:

$$|\mathbb{P}\left[\mathrm{Real}_{\mathcal{A}}^{\mathrm{KS^2E}}(\lambda) = 1\right] - \mathbb{P}\left[\mathrm{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\mathrm{KS^2E}}(\lambda) = 1\right]| = negl\,(\lambda)$$

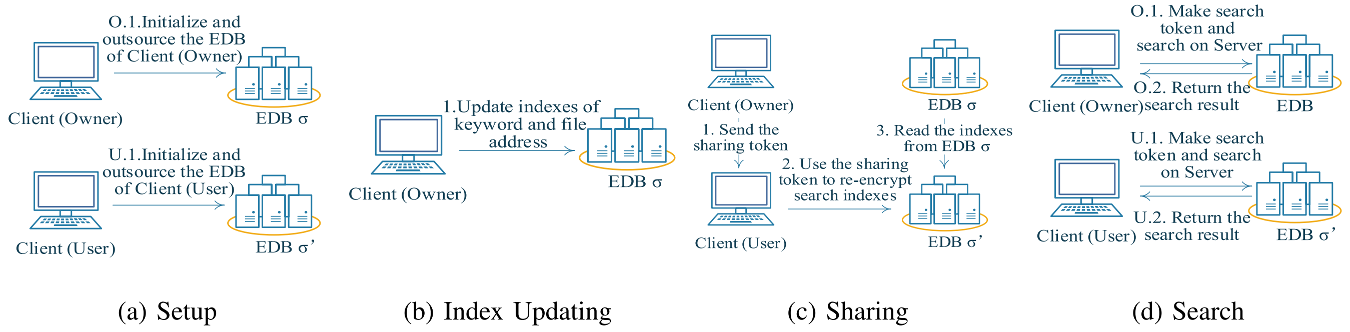
*where the game $\mathrm{Real}_{\mathcal{A}}^{\mathrm{KS^2E}}(\lambda)$ and the game $\mathrm{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\mathrm{KS^2E}}(\lambda)$ are defined as below:*

- *Game $\mathrm{Real}_{\mathcal{A}}^{\mathrm{KS^2E}}(\lambda)$: In this game, the adversary triggers Owner and User to run real protocols. Firstly, the adversary triggers the protocol* Setup *and gets two encrypted databases $\mathrm{EDB}_\sigma$, $\mathrm{EDB}_{\sigma'}$. Secondly, with the parameters of her choice, she adaptively triggers* Update, Search, Share, Sharetoken, *and then she gets the real transcript $(q_1, q_2, \dots q_n)$ from the view of Server. Finally, she outputs a bit $b$ decided from the real transcript $(\mathrm{EDB}_\sigma, \mathrm{EDB}_{\sigma'}, q_1, \dots q_n)$.*
- *Game $\mathrm{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\mathrm{KS^2E}}(\lambda)$: In this game, all of the real protocols are replaced by the set of simulators $\mathcal{S}$. The adversary first triggers the simulator $\mathcal{S}_{\mathsf{setup}}$ and gets two simulated databases $\mathrm{EDB}_\sigma^{\mathcal{S}}$, $\mathrm{EDB}_{\sigma'}^{\mathcal{S}}$. Secondly, with the parameters of her choice, she adaptively triggers $\mathcal{S}_{\mathsf{update}}$, $\mathcal{S}_{\mathsf{search}}$, $\mathcal{S}_{\mathsf{share}}$, $\mathcal{S}_{\mathsf{sharetoken}}$, and then she gets the simulated transcript $(q_1^{\mathcal{S}}, q_2^{\mathcal{S}}, \dots q_n^{\mathcal{S}})$ from the view of Server. Finally, she outputs a bit $b$ decided from the simulated transcript $(\mathrm{EDB}_\sigma^{\mathcal{S}}, \mathrm{EDB}_{\sigma'}^{\mathcal{S}}, q_1^{\mathcal{S}}, q_2^{\mathcal{S}}, \dots q_n^{\mathcal{S}})$.*

### *E. Leakage Function*

In this section, we borrow and define some basic leakage functions. Without the construction from data-oblivious memory, *search pattern* leakage $sp(w)$ is a common leakage in most DSSE schemes [3]. Let $u$ denote a timestamp. Let $Q$ denote a set of all queries issued up to now. $sp(w) = \{u : (u, w) \in Q\}$ leaks which search queries are related to the same keyword $w$. We borrow this leakage function to capture the search pattern leakage in $\mathrm{KS^2E}$.

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.




(a) Setup (b) Index Updating (c) Sharing (d) Search

Fig. 2. The generic KS$^2$E construction.

Inspired by [21], we also use the update history UpHist$(w)$ in KS$^2$E. UpHist$(w)$ outputs the list of all updates on keyword $w$. Moreover, the adversary also knows which database is updated. Let O denote the Update queries issued on the database from Owner. Let U denote the Update queries associated with the sharing requests from User. In our notion, the types of elements in the list UpHist$(w)$ include $(\mathsf{O}, u, w, id, op)$ and $(\mathsf{U}, u, w, id, op)$.

We define the *replication pattern* $rp(id)$ of each replication in the sharing process. Inspired by search patterns [12], the adversary observes the memory accessed on the Owner's database during the sharing process. The adversary can find out which sharing tokens access the same memory address and speculate that these sharing tokens share the same file $id$.

$rp(id)$ reveals the repetition of which $id$ generates a replicated index and is defined as $rp(id) = \{u : (u, id) \in Q\}$

### F. Forward and Backward Privacy

Forward and backward privacy is important security definitions when the SSE scheme has addition and deletion [3]. Especially, Zhang et al. [39] prove that adaptive file injection attack is powerful on encrypted databases without forward privacy. Meanwhile, search queries leak the deleted entries after the search protocol without backward privacy. We analyze the leakage from each protocol to determine forward and backward privacy in KS$^2$E.

*1) Forward Privacy:* As KS$^2$E holds forward privacy, the adversary should not use the previous transcript to imply the newly added content before executing the protocol Search or Share. More precisely, the source of the disclosure is the protocol Update and Share. If Owner and User want to replicate the $id$, the sharing token needs to access previous indexes about the $id$. Besides, if Owner and User want to search the index, the protocol Search needs to access the matched index by the keyword $w$. Hence, the leakage from Update can not disclose the content about $w$ and $id$. Besides, the Share protocol should not use previous sharing tokens to imply the content of newly added indexes.

To capture the forward privacy in KS$^2$E, we introduce the functions TimeST$(id)$ and UpSHist$(id)$. TimeST$(id)$ captures the timestamps when Owner generates the sharing token about $id$. With the input $id$, TimeST$(id)$ returns the timestamp set of sharing tokens $P_{id}$.

$$\text{TimeST}(id) = \{u | (u, P_{id}) \in Q\}$$

UpSHist$(id)$ captures the leakage of replicating the encrypted search indexes from Server. With the input $id$, UpSHist$(id)$ returns the set of the timestamps from Update queries. The timestamps should not be larger than the maximum of TimeST$(id)$.

$$\text{UpSHist}(id) = \{u | (\mathsf{O}, u, w, id, op) \in Q \quad and\ u \leq max\,[\text{TimeST}(id)]\}$$

Combining our analysis of the protocols of Update and Share, we have the definitions of forward privacy in KS$^2$E.

*Definition 3 (Forward Privacy): An $\mathcal{L}$-adaptively-secure* KS$^2$E *scheme has forward privacy iff its Update, Share leakage functions $\mathcal{L}_{Upt}$, $\mathcal{L}_{Shr}$ can be written as:*

$$\mathcal{L}_{Upt}(w, id, op) = \mathcal{L}'(op)$$
$$\mathcal{L}_{Shr}(id) = \mathcal{L}''(\text{UpSHist}(id))$$

*where $\mathcal{L}'$, $\mathcal{L}''$ are state-less functions.*

Recall that Bost et al. [3] introduced the forward and backward privacy for DSSE. As for the forward privacy, KS$^2$E further considers the leakage in the sharing stage. This leakage reveals the timestamp of the Owner's update operations before the last sharing on a file $id$. The adversary can not use the previous timestamp information to infer the content of the new update operation, and thus we can achieve the forward privacy.

*2) Backward Privacy:* As KS$^2$E holds backward privacy, the adversary should not use new transcripts to imply previous content after executing Search. In the protocol Share, the search index (in the Owner's database) is only re-encrypted and is further restored in the User's database. The Share does not leak deleted indexes about $id$. Besides, transcripts from the protocol Search should not reveal the deleted file address $id$. Considering the independence of search index tables on all databases, we require that Search can not disclose the content of $id$ *at most.*

To capture backward privacy in KS$^2$E, we introduce the functions TimeDBS$(w)$ and DelHistS$(w)$. Let $u^{add}$ denote the timestamp of an addition Update query. Let $u^{del}$ denote the timestamp of a deletion Update query. TimeDBS$(w)$ is the list of all $id$ matching $w$, excluding the deleted ones, together with the timestamps of when they were inserted in all databases.

$$\begin{aligned}\text{TimeDBS}(w) = \{(u, \mathsf{O}, id) |\ & (\mathsf{O}, u, w, add, id) \in Q \\ & and\ \ \forall\, u', (\mathsf{O}, u', w, del, id) \notin Q\} \\ & \cup \{(u, \mathsf{U}, id) |\ (\mathsf{U}, u, w, add, id) \in Q \\ & and\ \ \forall\, u', (\mathsf{U}, u', w, del, id) \notin Q \\ & and\ \ u \leq max\,[\text{TimeST}(id)]\}\end{aligned}$$

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.

DelHistS($w$) captures the leakage of the relation among addition and deletion about the same $(w, id)$ in all databases. With the input $w$, DelHistS($w$) returns the set of addition and deletion timestamp tuples. These timestamps reveal the correspondence between *add* and *del* queries about $w$ in all databases.

$$\begin{aligned} &\text{DelHistS}\,(w) \\ &= \Big\{\Big(u^{add}, u^{del}\Big) \mid \exists\, id\ s.t.\ \Big(\mathsf{U}, u^{add}, w, add, id\Big) \in Q \\ &\quad and\ \Big(\mathsf{U}, u^{del}, w, del, id\Big) \in Q \\ &\quad and\ u^{add} \le max\,[\text{TimeST}\,(id)] \\ &\quad and\ \ u^{del} \le max\,[\text{TimeST}\,(id)]\Big\} \\ &\quad \cup \Big\{\Big(u^{add}, u^{del}\Big) \mid \exists\, id\ s.t.\ \Big(\mathsf{O}, u^{add}, w, add, id\Big) \in Q \\ &\quad and\ \Big(\mathsf{O}, u^{del}, w, del, id\Big) \in Q\Big\} \end{aligned}$$

From these functions and our analysis, we define the backward privacy of the KS$^2$E scheme.

*Definition 4 (Backward Privacy): An $\mathcal{L}$-adaptively-secure* KS$^2$E *scheme has backward privacy if its* Update, Search *leakage functions $\mathcal{L}_{Upt}$, $\mathcal{L}_{Srch}$ can be written as*:

$$\mathcal{L}_{Upt}\,(w, id, op) = \mathcal{L}'\,(op)$$
$$\mathcal{L}_{Srch}\,(w) = \mathcal{L}'''\,(\text{TimeDBS}\,(w)\,,\text{DelHistS}\,(\text{w}))$$

*where $\mathcal{L}'$, $\mathcal{L}'''$ are state-less functions.*

The backward privacy is inspired by the Type-III backward privacy [3]. The search leakage on the Owner's database is included in the original Type-III backward privacy. The search leakage on the User's database is also included in the Type-III backward privacy, although the User's search is limited to the shared search indexes. After combining the two parts, the leakage does not speculate the preservation of the Type-III backward privacy.

Similarly, DMUSE [11] gives the forward and backward privacy definitions for multi-user sharing. The leakages of DMUSE are captured when multiple users use the Owner's search indexes. DMUSE tries to incorporate the leakage under static MUSE [13] into considering the forward and backward privacy in its work. In our model, Users use their keys to search their databases' index. The privacy definitions of KS$^2$E inherit from traditional DSSE but also minimize the leakage incurred by the sharing stage.

### G. Heuristic $KS^2E$ From DSSE

Through careful investigations, we find a heuristic instance of KS$^2$E by using multiple DSSE instances. Specifically, Owner can prepare two DSSE instances. One instance is to search the keywords and retrieve the file's $id$. To securely share the search index, the other instance is to search the shared file's $id$, retrieve all keywords, and securely deliver this retrieval to User. Finally, User updates the information transferred by Owner to his DSSE instance.

This heuristic instance suffers from some efficiency defects. It requires many search tokens and sharing tokens from different keys, increasing the time overheads of matching search indexes in the database. Moreover, Owner should be online to update the two DSSE instances at the same time, handle many sharing requests, and transfer sharing information. These cause considerable local computational resources to handle User's request. We say that the heuristic instance can maintain security as a DSSE, but it is not practical and efficient.

TABLE II

NOTATIONS

| Symbol | Description |
|---|---|
| SE | A symmetric encryption scheme |
| $\mathcal{E}$ | The encryption algorithm of an SE |
| $\mathcal{D}$ | The decryption algorithm of an SE |
| $\text{EDB}_\sigma$ | The Owner's encrypted database |
| $\text{EDB}_{\sigma'}$ | The User's encrypted database |
| $\text{lastID}_\sigma$ | An Owner's map, which stores the latest $id$ associated with $w$ |
| $\text{lastW}_\sigma$ | An Owner's map, which stores the latest $w$ associated with $id$ |
| $\text{lastID}_{\sigma'}$ | A User's map, which stores the latest $id$ associated with $w$ |
| $\text{lastW}_{\sigma'}$ | A User's map, which stores the latest $w$ associated with $id$ |
| $C_{w,id}$ | A searchable ciphertext generated from $(w, id)$ |
| L | An identifier of a search index in an encrypted database |
| $I_w$ | A ciphertext that is used to iterate to the previous search index associated with $w$ |
| $J_w$ | A ciphertext that starts the iterate to the previous search index associated with $w$ |
| $J_{id}$ | A ciphertext that is used to iterate to the previous search index associated with $id$ |
| $I_{id}$ | A ciphertext that starts the iterate to the previous search index associated with $id$ |
| $R_w$ | A random number used in the iteration of $w$ |
| $R_{id}$ | A random number used in the iteration of $id$ |
| $C_w$ | A symmetric ciphertext generated from the $id$ of $(w, id)$ |
| $C_{id}$ | A symmetric ciphertext generated from the $w$ of $(w, id)$ |
| $S$ | A result set from Server |
| $T_w$ | A search token generated from keyword $w$ |
| $P_{id}$ | A sharing token generated from $id$ |
| $D_{id}$ | A partial token from sharing token $P_{id}$ |

## IV. Instance Construction

This section introduces our approach to constructing the instance of KS$^2$E. To construct our KS$^2$E instance, we introduce our notation, preliminary, and bi-directional index. Following the KS$^2$E generic construction, we show the detailed workflow of Branch. Finally, we analyze the correctness, security, and performance.

### A. Notation and Preliminary

*1) Notation:* Let $\lambda \in \mathbb{N}$ be the security parameter. Symbol $r \xleftarrow{\$} \mathcal{R}$ means randomly choosing $r$ from the space $\mathcal{R}$. Symbol $a||b$ means the concatenation between $a$ and $b$. Symbol $\{0, 1\}^\lambda$ means a $\lambda$-bit length string. Symbol $\{0, 1\}^*$ means an arbitrary-bit length string. Symbol $H$ means a cryptographic hash function. Symbol $F$ means a secure Pseudo Random Function (PRF). The frequently used notations are given in Tab. II.

*2) Symmetric Encryption:* Given a security parameter $\lambda \in \mathbb{N}$, message space $\mathcal{M} = \{0, 1\}^*$, ciphertext space $\mathcal{C} = \{0, 1\}^*$, and the key space $\mathcal{K}_{SE} = \{0, 1\}^*$, a symmetric encryption scheme consists of two algorithm $(\mathcal{E}, \mathcal{D})$ and runs under the following syntax:

- $\mathcal{E}(k, m)$: Input a symmetric key $k \in \mathcal{K}_{SE}$ and a message $m \in \mathcal{M}$, output a ciphertext $c \in \mathcal{C}$.
- $\mathcal{D}(k, c)$: Input a symmetric key $k \in \mathcal{K}_{SE}$ and a ciphertext $c \in \mathcal{C}$, recover a message $m \in \mathcal{M}$.

For correctness, with each message $m \in \mathcal{M}$ and secret key $k \in \mathcal{K}_{SE}$, the equation $c \leftarrow \mathcal{E}(k, m)$ should always make sense, and $m \leftarrow \mathcal{D}(k, c)$ can always recover the message $m$ from $c$ using the secret key $k$. In the security definitions, a popular requirement of symmetric encryption is the semantic security under chosen plaintext attack (SS-CPA).

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.

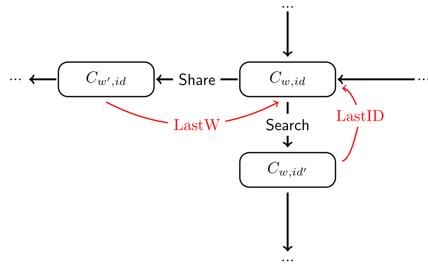



Fig. 3. Relation between tokens and encrypted search indexes.

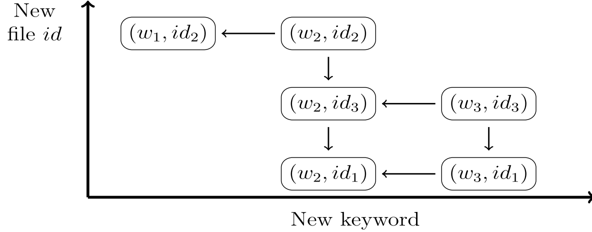



Fig. 4. Logical bi-directional index structure for the example sequence.

### *B. Bi-Directional Index*

The bi-directional index is a structure constructed from encrypted search indexes. A popular method in constructing DSSE is to reveal the following index from the calculation between tokens and the current matched index. Following this idea, we construct iterations from the last matched encrypted search index using the tokens of Share or Search.

Fig. 3 shows the relation between tokens and encrypted search indexes. The index $C_{w,id}$ needs to generate from two types of internal state LastID, LastW. LastID is a map storing the latest $id$ updated with $w$. LastW is a map storing the latest $w$ updated with $id$. In this approach, the search token can match the current $C_{w,id}$ and utilize previous LastID state information to find the previous index $C_{w,id'}$. Similarly, the sharing token can match the current $C_{w,id}$ and utilize previous LastW state information to find the previous index $C_{w',id}$.

Fig. 4 shows a logical bi-directional index table after adding the example sequence: $(w_2, id_1)$, $(w_1, id_2)$, $(w_2, id_3)$, $(w_3, id_1)$, $(w_2, id_2)$, $(w_3, id_3)$. We can observe that each unique pair $(w, id)$ pointing to the horizontal has the same $id$, and the pair pointing to the vertical has the same $w$.

*Advantage:* The bi-directional index avoids performance degradation. This structure enables the same private key to perform search and sharing operations. The generation of sharing and search tokens only needs the internal states about the chain tail's information. Compared with MUSE [11], [13] and MKSE [6], [36], we find that the bi-directional index enables Users to merge several search indexes collected from multiple owners into one index. Batching into one approach is naturally more practical and efficient when there are multiple owners that share files simultaneously.

### *C. The Workflow of Branch Instance*

We use functions defined in the KS[2]E construction to perform search index sharing from Owner to User. More precisely, we take inputs in the example sequence of Fig. 4 as update queries of Owner and let Owner share search indexes about $id_1$ with User. At the *Search*, we let User and Server run the protocol Search with the keyword $w_2$ and retrieve the search result.

**Protocol 1** Branch: Setup

**Setup**($\lambda$) : // the example between Owner and Server, otherwise Gen. $K_{\sigma'}$, lastID$_{\sigma'}$, lastW$_{\sigma'}$, EDB$_{\sigma'}$

1: Initialize $H : \{0,1\}^\lambda \times \{0,1\}^\lambda \rightarrow \{0,1\}^{2\lambda}$
2: Initialize $F : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$
3: Initialize SE: $\{\mathcal{E}, \mathcal{D}\}$
4: Gen. $K_\sigma$: $K_{\sigma,1} \xleftarrow{\$} \{0,1\}^\lambda$, $K_{\sigma,2} \xleftarrow{\$} \{0,1\}^\lambda$
5: Gen. empty maps lastID$_\sigma$, lastW$_\sigma$ as the internal state $\sigma$
6: Gen. encrypted database EDB$_\sigma$ $\leftarrow$ MAP

*1) Setup:* Owner and User separately run the protocol Setup. Owner gets the master secret key $K_\sigma$ and the maps lastID$_\sigma$, lastW$_\sigma$ as internal state $\sigma$. User gets the master secret key $K'_\sigma$ and the maps lastID$_{\sigma'}$, lastW$_{\sigma'}$ as internal state $\sigma'$. Owner sends EDB$_\sigma$ to Server, and User sends EDB$_{\sigma'}$ to Server. The lastID maps the latest added $w$ to $id$, and the lastW maps the latest added $id$ to $w$. These maps capture the endpoint in the bi-directional index structure.

**Protocol 2** Branch: Update

**Update**($K_\sigma, \sigma, (w, id, add)$; EDB$_\sigma$) :

**Owner:**
1: $w' \leftarrow$ lastW$_\sigma[id]$,$id' \leftarrow$ lastID$_\sigma[w]$
2: $L \leftarrow F(K_{\sigma,1}, w||id)$, $R_w \xleftarrow{\$} \{0,1\}^\lambda$, $R_{id} \xleftarrow{\$} \{0,1\}^\lambda$ //location and two random numbers for pointers
3: $k_w \leftarrow F(K_{\sigma,2}, w)$, $k_{id} \leftarrow F(K_{\sigma,2}, id)$
4: $C_w \leftarrow \mathcal{E}(k_w, id)$, $C_{id} \leftarrow \mathcal{E}(k_{id}, w)$
5: **if** $id'$ is $\perp$ **then**
6: $\quad I_w \leftarrow H(F(K_{\sigma,2}, w||id), R_w) \bigoplus \perp||\perp$
7: **else**
8: $\quad L' \leftarrow F(K_{\sigma,1}, w||id')$, $J'_w \leftarrow F(K_{\sigma,2}, w||id')$
9: $\quad I_w \leftarrow H(F(K_{\sigma,2}, w||id), R_w) \bigoplus (L'||J'_w)$
10: **end if**
11: **if** $w'$ is $\perp$ **then**
12: $\quad I_{id} \leftarrow H(F(K_{\sigma,2}, id||w), R_{id}) \bigoplus \perp||\perp$
13: **else**
14: $\quad L' \leftarrow F(K_{\sigma,1}, w'||id)$, $J'_{id} \leftarrow F(K_{\sigma,2}, id||w')$
15: $\quad I_{id} \leftarrow H(F(K_{\sigma,2}, id||w), R_{id}) \bigoplus (L'||J'_{id})$
16: **end if**
17: $C_{w,id} \leftarrow (L, I_w, R_w, C_w, I_{id}, R_{id}, C_{id})$
18: Update lastW$_\sigma[id] \leftarrow w$, lastID$_\sigma[w] \leftarrow id$
19: **Send** $C_{w,id}$

**Server:**
20: Parse $C_{w,id}$ as $(L, I_w, R_w, C_w, I_{id}, R_{id}, C_{id})$
21: EDB$_\sigma[L] \leftarrow (I_w, R_w, C_w, I_{id}, R_{id}, C_{id})$

*2) Index Updating:* Owner and Server run the protocol Update with the example sequence, and Server stores these indexes in EDB$_\sigma$ with a logical bi-directional index structure like the left part of Fig. 5. Owner uses his internal state lastID$_\sigma$, lastW$_\sigma$ to find the latest $w'$ added on $id$ and the latest $id'$ added on $w$ (step 1). He generates $L$ as the identifier of the search index in EDB$_\sigma$ and generates random $R_w$, $R_{id}$ to make randomness to the previous index pointer (step 2). He generates symmetric encryption keys $k_w$, $k_{id}$ and encrypts the $id$, $w$ to the $C_w$, $C_{id}$ as the retrieval of this search index (steps 3-4).

To construct the vertical index chain for search tokens, Owner checks the latest $id'$ whether it is $\perp$ or not. Then he creates a vertical chain head like the $I_w$ in nodes $L_{w_1,id_2}$, $L_{w_2,id_1}$, $L_{w_3,id_1}$ or makes the $I_w$ point at the vertical chain tail by encrypting the latest $L'||J'_w$ (steps 5-10). To construct the horizontal index chain for sharing tokens, Owner checks whether the latest keyword $w'$ is $\perp$ or not. Then he creates a

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.

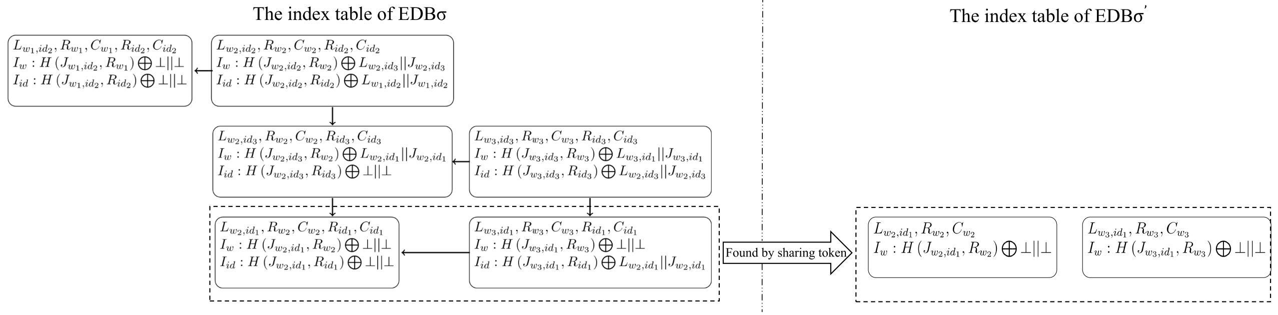



Fig. 5. The encrypted search indexes on Server.

horizontal chain head like the $I_{id}$ in nodes $L_{w_1,id_2}$, $L_{w_2,id_3}$, $L_{w_2,id_1}$ or makes the $I_{id}$ point at the horizontal chain tail by encrypting the latest $L'||J'_{id}$ (steps 11-16). Finally, Owner sends this index $C_{w,id}$, and Server store it in $\text{EDB}_\sigma$.

**Protocol 3** Branch: Sharetoken

**Sharetoken**$(K_\sigma, \sigma, id)$ :

**Owner:**
1: $w \leftarrow \text{lastW}_\sigma[id]$
2: **if** $w$ is $\perp$ **then**
3: return $\perp$
4: **end if**
5: $L \leftarrow F(K_{\sigma,1}, w||id)$, $J_{id} \leftarrow F(K_{\sigma,2}, id||w)$
6: $k_{id} \leftarrow F(K_{\sigma,2}, id)$
7: **Send** $P_{id} \leftarrow (L, J_{id}, id, k_{id})$ to User

**Protocol 4** Branch: Share

**Share**$(P_{id}, \sigma', K_{\sigma'}; \text{EDB}_{\sigma'}, \text{EDB}_\sigma)$ :

**User:** // the first-round interaction
1: Decapsulate and parse $P_{id} = (L, J_{id}, id, k_{id})$
2: Send $D_{id} \leftarrow (L, J_{id})$
**Server:**
3: $S \leftarrow \emptyset$, read the uploaded $\text{EDB}_\sigma$
4: **while** $L$ is not $\perp$ **do**
5: Choose $(I_{id}, R_{id}, C_{id})$ from $\text{EDB}_\sigma[L]$
6: $(L'||J'_{id}) \leftarrow H(J_{id}, R_{id}) \bigoplus I_{id}$ //get the previous location that has the same $id$
7: $S \leftarrow S \cup C_{id}$
8: Update $L \leftarrow L'$, $J_{id} \leftarrow J'_{id}$
9: **end while**
10: Return $S$
**User:** // the second-round interaction
11: $S' \leftarrow \emptyset$
12: **for** each symmetric ciphertext $s$ in S **do**
13: $w \leftarrow \mathcal{D}(k_{id}, s)$,$id' \leftarrow \text{lastID}_{\sigma'}[w]$
14: $C_w \leftarrow \mathcal{E}(F(K_{\sigma',2}, w), id)$
15: $L \leftarrow F(K_{\sigma',1}, w||id)$,$R_w \xleftarrow{\$} \{0,1\}^\lambda$
16: **if** $id'$ is $\perp$ **then**
17: $I_w \leftarrow H(F(K_{\sigma',2}, w||id), R_w) \bigoplus \perp||\perp$
18: **else**
19: $L' \leftarrow F(K_{\sigma',1}, w||id')$, $J'_w \leftarrow F(K_{\sigma',2}, w||id')$
20: $I_w \leftarrow H(F(K_{\sigma',2}, w||id), R_w) \bigoplus (L'||J'_w)$
21: **end if**
22: Update $\text{lastID}_{\sigma'}$[w] $\leftarrow id$
23: Update $S' \leftarrow S' \cup (L, I_w, R_w, C_w)$
24: **end for**
25: Send $S'$
**Server:**
26: **for** each ciphertext $s'$ in $S'$ **do**
27: parse $s'$ as $(L, I_w, R_w, C_w)$, $\text{EDB}_{\sigma'}[L] \leftarrow (I_w, R_w, C_w)$
28: **end for**

*3) Sharing:* First, Owner runs Sharetoken to generate the sharing token $P_{id}$ and sends it to User. He checks the internal state $\text{LastW}_\sigma$ to find the latest keyword $w$ about $id$ or returns when the keyword $w$ is $\perp$ (steps 1-4). He generates $L$ and $J_{id}$, which can identify the horizontal chain tail about $id$, and he sends the sharing token $P_{id}$ to User (steps 5-7).

After that, User and Server run Share. User sends the partial token of $P_{id}$ to Server in the first round of interaction (step 2). Server iteratively matches search indexes in the $\text{EDB}_\sigma$ of Owner by the identifier $L$ (steps 4-9). Server executes XOR to restore the previous $L'||J'_{id}$ (step 6). After finding all encrypted keywords $C_{id}$ on the file $id$, Server returns the encrypted keywords set $S$ to User.

In the second round of interaction, User decrypts each $C_{id} \in S$ and gets the keyword $w$. Then he checks his internal state $\text{lastID}_{\sigma'}[w]$ to get the latest $id'$ about this keyword (step 13). He generates symmetric ciphertext $C_w$ for retrieving the $id$ on Search (step 14). He generates $L$ as the identifier of this search index (step 15). He checks $id'$ whether it is $\perp$ or not, and then he creates a vertical chain head $I_w$ or makes the $I_w$ point at the chain tail (steps 16-21). Finally, User generates the set of search indexes $S'$, and Server stores it in $\text{EDB}_{\sigma'}$. In our example sequence, the shared search indexes in $\text{EDB}_{\sigma'}$ are shown like the right part in Fig. 5.

**Protocol 5** Branch: Search

**Search**$(K_{\sigma'}, \sigma', w; \text{EDB}_{\sigma'})$ : // the search example between User and Server, otherwise input $K_\sigma$, $\sigma$, $\text{EDB}_\sigma$ for Owner

**User:**
1: $id \leftarrow \text{lastID}_{\sigma'}[w]$
2: **if** $id$ is $\perp$ **then**
3: return $\perp$
4: **end if**
5: $L \leftarrow F(K_{\sigma',1}, w||id)$, $J_w \leftarrow F(K_{\sigma',2}, w||id)$
6: Send $T_w = (L, J_w)$ to the Server
**Server:**
7: Parse $T_w$ as $(L, J_w)$
8: $S \leftarrow \emptyset$
9: **while** $L$ is not $\perp$ **do**
10: Choose $(I_w, R_w, C_w)$ from $\text{EDB}_{\sigma'}[L]$ // get the previous location that has the same $w$
11: $(L'||J'_w) \leftarrow H(J_w, R_w) \bigoplus I_w$
12: $S \leftarrow S \cup C_w$
13: Update $L \leftarrow L'$, $J_w \leftarrow J'_w$
14: **end while**
15: Return $S$

*4) Search:* User and Server run the protocol Search to retrieve the file address by the keyword $w_2$. User checks $\text{lastID}_{\sigma'}$[w] to get the latest $id$ then sends the search token $T_w$ (step 5). Server uses this search token to restore the last step $L'||J'_w$ iteratively until the identifier $L$ is $\perp$ (steps 9-14). Finally, Server makes a set $S$ of matched symmetric ciphertexts $C_w$ from accessed indexes and returns this search result. To retrieve the actual file, User can extract the file $id$ and send it to Server who is then able to retrieve the file.

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.

### D. Instance Analysis

*1) Correctness:* According to the correctness in Def. 1, the correctness of Branch can be reduced to the collision resistance of tokens in the protocol Sharetoken and Search [21]. To prove the correctness of Branch, we further reduce the probability of finding the tokens' collision to that of solving the Birthday problem [40]. Let $\mu$ denote the maximum number of inputs to the PRF $F$. The collision probability is $\mathbb{P}_{Coll} = 1 - e^{\frac{-\mu^2}{2^{\lambda+1}}}$, which should be negligible in practice. For example, when using the SHA256 hash function in Branch and choosing $\lambda = 256$ and $\mu = 10^{20}$, the $\mathbb{P}_{Coll}$ is $4.32 \cdot 10^{-38}$. Hence, the probability is small enough to achieve collision resistance. Accordingly, the correctness is proved. In Appendix B, we present the calculation of the collision probability in detail.

*2) Security:* Branch securely generates the search indexes, search tokens, and sharing tokens, guaranteed by the cryptographic hash function and the secure PRF. Based on the security parameters used in the protocol Setup, external adversaries are prevented from violently cracking ciphertexts. For forward and backward privacy, Owner always uses the master secret key $K_\sigma$ and secure PRF to generate new tokens when adding new $(w, id)$. After receiving a new sharing request generated from the same $id$, Server only knows the new keyword count on specific search indexes. We use this to restrict the previous tokens from matching the newly added index, and thus we have the following theorem.

*Theorem 1 (Adaptive Security of KS$^2$E-Branch): Let PRF $F$ with a specific key be modeled as the random oracle $\mathcal{H}_F$, and let cryptographic hash function $H$ be modeled as the random oracle $\mathcal{H}$. We define $\mathcal{L}_B = (\mathcal{L}_B^{Upt}, \mathcal{L}_B^{Shr}, \mathcal{L}_B^{Srch})$ as:*

$$\mathcal{L}_B^{Upt}(w, id, op) = \emptyset$$
$$\mathcal{L}_B^{Shr}(id) = rp(id), \text{UpSHist}(id)$$
$$\mathcal{L}_B^{Srch}(w) = sp(w), \text{UpHist}(w)$$

*Branch is $\mathcal{L}_B$-adaptively-secure.*

In our proof of instance Branch, we use a game hop method to replace the real output of the hash function and PRF step by step. The cryptographic hash function $H$ is modeled as a random oracle $\mathcal{H}$. In the real game, the initial game is identical to the instance Branch. First, we replace $F$ with the random oracle $\mathcal{H}_F$ and replace Symmetric Encryption ciphertexts with randomly chosen strings. Next, we replace all strings generated from random oracles in the protocol Update with randomly chosen strings. In this step, we program the random oracles for the input to simulate the original hash functions in Search, Sharetoken, and Share. Finally, we make the game only use the pattern and the history to generate tokens. In this condition, the game only needs the output of leakage functions. In other words, we have a simulator.

In Appendix A, we give formal proof for our instance with an indistinguishable advantage.

*3) Complexity:* We analyze the complexity of our instance Branch. In the function Setup, Branch initializes the cryptographic hash function $H$, master secret key $K_\sigma$, internal state $\text{lastID}_\sigma$, $\text{lastW}_\sigma$, PRF $F$, and encrypted database $\text{EDB}_\sigma$ in constant time. In the protocol Update, Branch generates the search index $C_{w,id}$ in constant time using the cryptographic hash function $H$ and PRF $F$. A one-way interaction exists between Owner and Server for the $\text{EDB}_\sigma$. In the protocol Search, the PRF $F$ is used to make the search token $T_w$ within constant time, and the search token has a constant length. During the one interaction, Server needs $O(n(I_w))$ time to find all indexes matching this search token, where $n(I_w)$ denotes the count of $I_w$ in the current encrypted database. In the protocol Sharetoken, Owner uses the constant time to generate the constant length sharing token $P_{id}$. In the protocol Share, User has two round interactions with Server. In the first-round interaction, User needs constant time to compute $D_{id}$, and Server needs $O(n(I_{id}))$ time to retrieve ciphertexts where $n(I_{id})$ denotes the count of $I_{id}$ in the current encrypted database. In the second-round interaction, User needs constant time to generate each index, and Server takes constant time to add each index in $\text{EDB}_{\sigma'}$.

*4) Deletion:* Although Branch can only store the $add$ index, we can fix it using a "copy" of Branch instance [3]. We can use an instance to record the search index containing the $add$ operation and another instance for the $del$. After Owner or User sends the same content search queries to the two instances, the actual result can be implied by comparing the results of the two instances locally.

After creating two instances, the server can know the type of instance [3], enabling it to know when the deletion occurred during the update stage. Therefore, this multi-instance approach can provide forward privacy with weak backward privacy. To overcome this leakage during the update stage, we can deploy branch instances separately on the non-collusive servers, preventing the server from knowing the type of instances.

It is challenging to construct efficient KS$^2$E instances supporting both add and delete operations. Previous works usually added a file deletion content $id||del$ to the EDB in the update protocol [22], [26], which is indistinguishable from the file addition content. We tried adding a search index deletion content but faced a situation where we could not use the same private key to perform searching and sharing. Therefore, we implemented Branch with the bi-directional index to make the index efficiently searchable and shareable. Another possible solution is to use Puncturable Encryption [3], [23], [24], allowing the server to decrypt the non-deleted content. However, the puncture key should be renewed after each search and require rebuilding the bi-directional index structure.

## V. Evaluation of Inter-Cloud Context

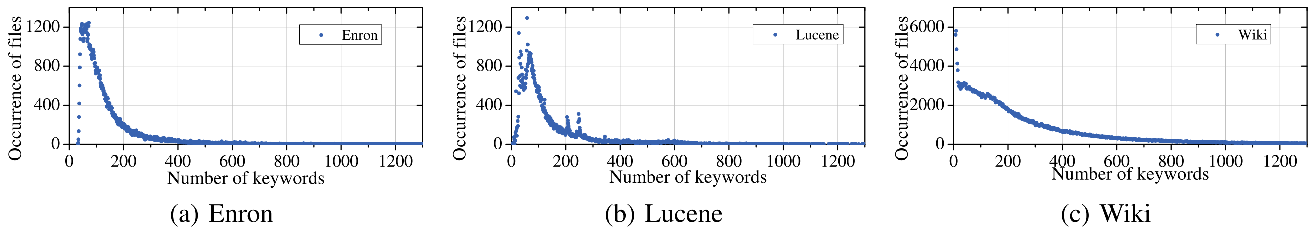
In this section, we select the Inter-cloud data replication context to test the performance of Branch. We specify Server as Cloud A and B to hold $\text{EDB}_\sigma$ and $\text{EDB}_{\sigma'}$. First, we describe several popular datasets for testing our instance's performance. Then, we show our test platform with cryptographic tools. Next, we introduce our evaluation setup for each aspect of KS$^2$E. Finally, we observe and analyze our evaluation results.

### A. Dataset

*1) Enron Email Dataset:*[1] We choose this popular dataset to create an EDB [39], [41]. Enron email dataset stores the data

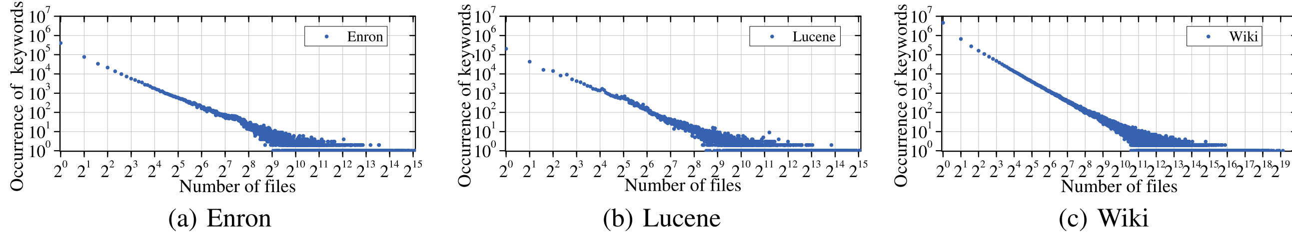
[1] Enron email dataset: online at https://www.cs.cmu.edu/ enron/

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.



(a) Enron (b) Lucene (c) Wiki

Fig. 6. The occurrence of files.



(a) Enron (b) Lucene (c) Wiki

Fig. 7. The occurrence of keywords.

converted from Enron corporation emails into text files. Our test applies 128,103 documents in the 'all-documents' folder as the data source. We remove the HTML format code to eliminate the effects caused by stop-words. Then we use the Python NLTK package to remove stop-words like punctuations and other unnecessary texts in these text files. After that, we use Porterstemmer [42] to extract keywords from the text files. The size of extracted keyword space is 638,070, and the total number of $(w, id)$ pairs is 18,309,166.

*2) Lucene:*[2] We choose the Lucene email dataset for testing on a similar scale dataset with another frequency distribution. Lucene email dataset is a java-user mailing list from the Lucene project. It stores the email sent by the developer of this project and is divided into each author. Our test samples this email data in its mailing list from Sept. 2001 to May 2020. After removing stop-words and extracting from Porterstemmer, we get the Lucene email dataset. It has 110,008 documents corresponding to a keyword space of 354,708 keywords, and the total number of $(w, id)$ pairs is 14,935,640.

*3) Wikipediadump:* [3] In order to test the performance of Branch in a larger scale data, we use the sample from Wikipediadump to create a large dataset. The Wikipediadump is an image from the content of Wikipedia. Our test downloaded the dump on Sept. 20, 2021, and sampled the first 2,000,000 dumped data without non-text files. Before extracting keywords, we use Wikiextractor [43] to remove the HTML format code and get the sliced Json structure data. Then we use the Python NLTK package [44] to 'tokenize' these data and remove stop-words. After this extraction, the size of Wikipediadump has 1,983,819 valid documents corresponding to a 6,441,361 keyword space, and the total number of $(w, id)$ pairs is 272,897,980.

Fig. 6 shows the occurrence of files in each dataset. It represents the number of files that meet a certain number of keywords. As we can see, all distributions of datasets are similar to the log-normal distribution. Most files contain less than 300 keywords. When the number of keywords is less than 100, many files have few words in the Wikipediadump dataset.

[2]Lucene: online at https://lists.apache.org/list.html?dev@lucene.apache.org

[3]Wikipediadump: online at https://dumps.wikimedia.org/

TABLE III

TEST ENVIRONMENT

| Platform | Intel® Xeon Gold 5120 CPU @ 2.20GHz, 128GB, Dell RERC H730 Adp SCSI Disk Device, Windows Server 2016 Standard |
|---|---|
| OpenSSL | v1.0.2g, 1 Mar 2016 |
| MongoDB | v3.0.15 |
| Python | v3.60 |
| VMware | v15.5.6 build-16341506 with the instance of Ubuntu 16.04 |

Besides, we observe that the presence of words in common use affects the log-normal distribution of the Lucene dataset.

Fig. 7 shows the occurrence of keywords in each dataset. It represents the number of the same keywords in certain files. As we can see, all datasets follow the rule that the number of the same keywords decreases with the increment in the number of files.

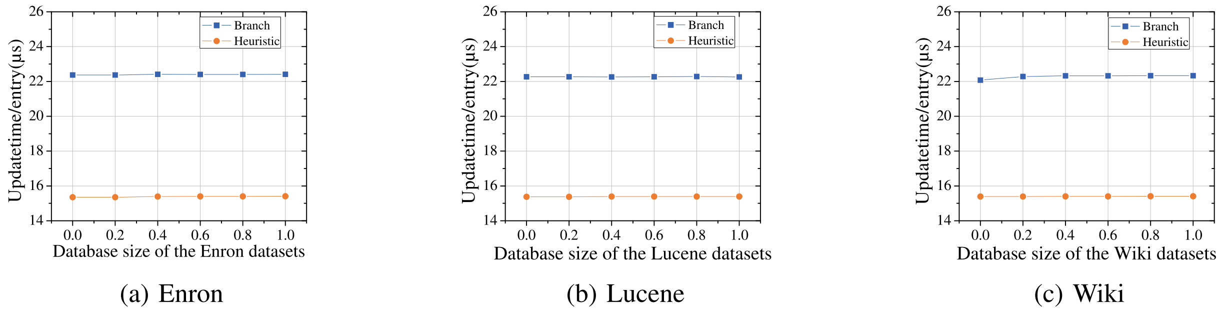
## B. Test Platform

Tab. III shows our test environment to simulate our test context. To build Owner, User, Cloud A, and Cloud B, we choose VMware Workstation and assign CPU cores/memory on each side. We allocate 16 cores and 64GB of memory for each Cloud. MongoDB [45] is a popular NoSQL database with high performance on query responses. We choose MongoDB to store the encrypted data, which fits the storage structure of the key-value pair.

When coding the instance Branch, we use SHA512 from OpenSSL [46] to construct the cryptographic hash function $H$, and we use the HMAC function from OpenSSL to construct the secure PRF $F$. The SHA256 parameter initializes the HMAC function. We use the Python object model to encapsulate this part of the OpenSSL code.

When coding the connection of databases, we use the MongoDB database in single-node mode to store search indexes, and we use the PYmongo package to connect the database on each side. In accessing the database, we use data slicing and thread parallelism to reduce the additional time overhead of accessing the database content. In order to make interactions between Cloud A and B, we use the Python socket package to transfer the encrypted data under ASCII encoding by LAN.

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.



(a) Enron (b) Lucene (c) Wiki

Fig. 8. Comparison of updating time overheads between Branch and the heuristic KS$^2$E instance in LAN.

### C. Evaluation Setup

Inspired by the previous evaluation setup in DSSE and the functionality of sharing search indexes, we confirm that the time expenses of operations represent the primary performance of each scheme. We record the expansion of plaintext and ciphertext databases and design experiments to test the time expenses of sharing and searching.

*1) Updating Time Overhead in Branch:* In the test of encryption time overhead, we use the protocol Update of Branch to encrypt entries of each dataset. In detail, we let Owner read and update all plaintexts and then write ciphertexts into Cloud A database in ciphertext collections. We record the entire updating time overhead three times and take the average overhead on each dataset. After that, we get the average updating time overhead from the percentage of encrypted data in the whole dataset.

*2) Sharing Time Overhead in Branch:* To test the sharing time overhead of Branch, we use the protocols Sharetoken and Share to share all indexes from Cloud A to B. Then we record the average sharing time overhead of User and Cloud B in the protocol Share, respectively. During this test, Owner generates the sharing tokens of each $id$ and sends them to User. After receiving them, User executes the protocol Share with Cloud B. We sample the sharing time overhead of documents with different numbers of keywords by the number of matching keywords in the document.

*3) Search Time Overhead in Branch:* We test the search time overhead from User and Cloud B running the protocol Search. During this test, User sends search tokens to Cloud B. Cloud B queries the database using the search token and returns search results. We search the top 30 keywords ordered by the highest frequency on the three datasets and sample the average search time overhead on each keyword.

*4) Compare With the Heuristic* **KS$^2$E** *Instance:* We construct a heuristic KS$^2$E instance from the DSSE instance Diana. Recall that Diana is a forward private scheme which is applied to the forward and backward secure DSSE [3], [22], [23], [24]. It uses the Constrained Pseudo Random Function (CPRF) to guarantee forward privacy. We instantiate Diana based on the tree-based GGM PRF [47], [48]. Then we use our datasets to test the heuristic KS$^2$E and compare it with Branch.

*5) Compare With MUSE and MKSE:* To evaluate the performance of searching for files shared by multiple Owners, we measure the time overheads of database generation, sharing, and keyword searching using MKSE and MUSE under LAN and WAN. We employ multiple Owners and outsource encrypted databases to Cloud A to adapt the setting. We compare the sub-linear search instance PKC18 [6], which uses the Perfect Hash Function, with MKSE. We choose the queue-based $\mu$SE instance (Q-uSE) in [11] for comparison. Both instances focus on efficient database generation, sharing, and search. The PKC18 instance uses the *DataKeyGen* algorithms for each Owner and the *QueryKeyGen* for one User. The queue-based $\mu$SE instances are separately initialized for multiple Owners, and each Owner can *Enroll* the identity of one User. We uniformly sample 2,000 files from our Enron email dataset to generate an appropriate dataset. The MUSE and MKSE instances are coded in Python, and they use the same construction of the secure PRF.

TABLE IV
COMPARISON OF STORAGE OVERHEAD BETWEEN BRANCH AND THE HEURISTIC KS$^2$E

| Dataset | Size (GB) | Branch | | Heuristic | |
|---|---|---|---|---|---|
| | | State(MB) | EDB(GB) | State(MB) | EDB(GB) |
| Enron | 0.72 | 21.46 | 8.47 | 10.66 | 1.97 |
| Lucene | 0.59 | 21.46 | 6.91 | 10.66 | 1.59 |
| Wiki | 10.81 | 320.33 | 126.27 | 166.64 | 29.14 |

### D. Comparison With the Heuristic $KS^2E$ Instance

Fig. 8 shows the updating time overhead of Branch and the heuristic KS$^2$E on each dataset. They roughly share a constant time overhead for updating a search index in a database, while they have updating overheads of about 22.5 $\mu s$ and 15.3 $\mu s$ per entry, respectively. Branch does spend more time in constructing the bi-directional index structure among searchable ciphertexts. In the updating stage, we see that Branch incurs more cost on calling the hash function than the heuristic KS$^2$E.

Tab. IV shows the storage overhead of Branch and the heuristic KS$^2$E. Branch consumes more memory to store the internal state for search and sharing than the heuristic KS$^2$E. Besides, the EDB storage cost by Branch is about 4.3 times that of the heuristic KS$^2$E. We note that Branch requires more storage cost to maintain the bi-directional index structure.

Fig. 9 shows the entire sharing time cost. The costs of Branch on Enron, Lucene, and Wikipedia are 614.87s, 563.81s, and 1.73 $\times 10^4$ s, respectively. Branch's cost increases by 28.0 times between Enron and Wikipedia datasets and 30.7 times between Lucene and Wikipedia datasets. The overheads of the heuristic are at least nine $times$ larger. We see that the bi-directional index accelerates the retrieval in the data-collection slice of MongoDB and speeds up the index sharing.

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.

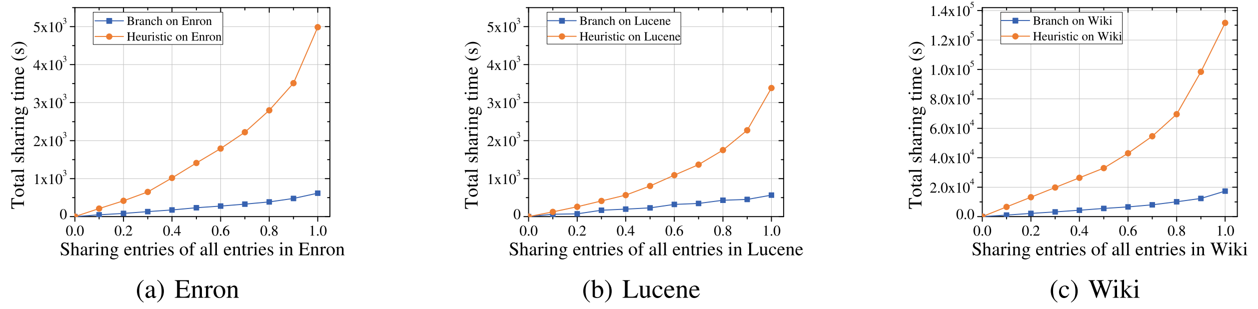



Fig. 9. Comparison of sharing time overheads between Branch and the heuristic KS$^2$E instance in LAN.

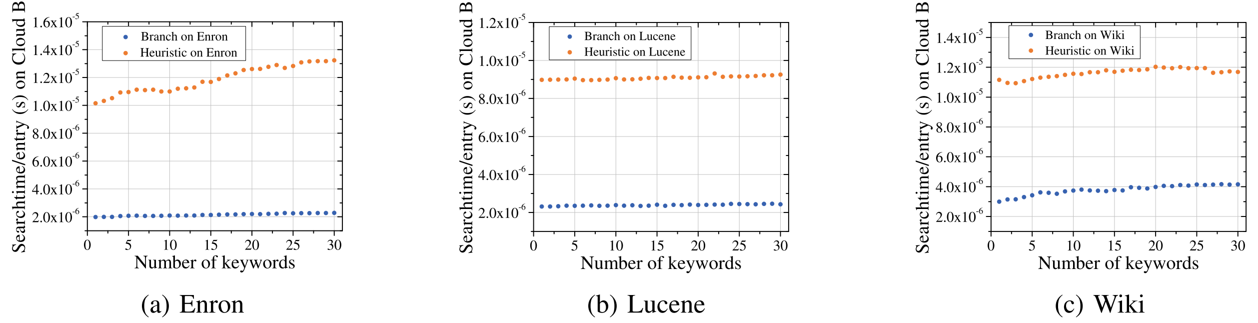



Fig. 10. Comparison of search performance between Branch and the heuristic KS$^2$E instance in LAN.

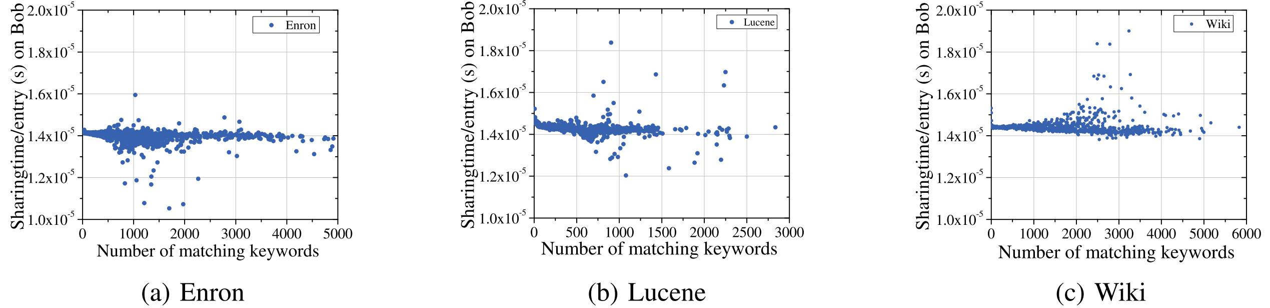



Fig. 11. Sharing performance on User in LAN.

Fig. 10 shows the search time overhead of Branch and the heuristic KS$^2$E on each dataset. From these search times of Branch, we observe that the search time overhead of each index is gradually increasing with the gradual reduction of keyword frequency. The frequency of the 30th keyword decreases by 0.37, 0.38, and 0.45 times that of the first keyword. At the same time, the keyword frequency significantly affects the search time on each dataset. The search time overheads of the 1st frequency keyword in Branch is separately $1.98 \times 10^{-6}$s, $2.31 \times 10^{-6}$s, $2.99 \times 10^{-6}$s, and that of the 30th frequency keyword is separately $2.27\times10^{-6}$s, $2.45\times10^{-6}$s, $4.15\times10^{-6}$s. The search performance of Branch is nearly 4.3 times faster than that of the heuristic KS$^2$E in our test.

### *E. The Sharing Performance of Branch*

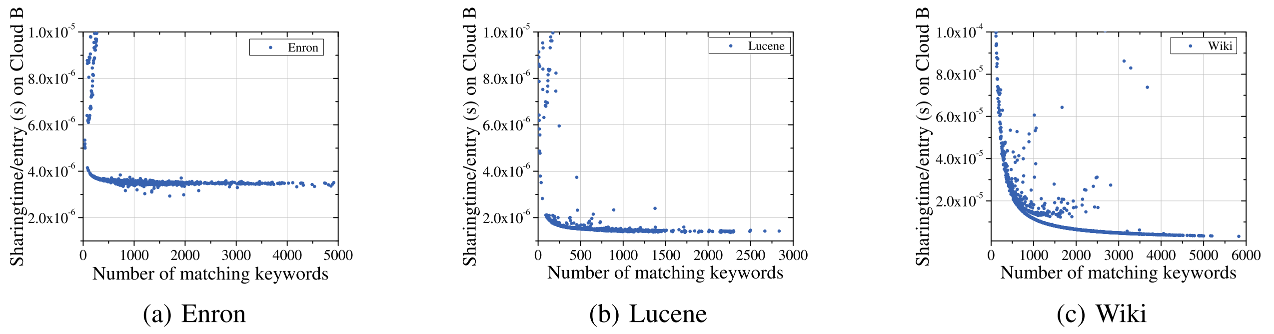
Fig. 11 shows the sharing time overhead of User on each dataset in detail. These graphs show the average cost of sharing an index according to the number of matches from one Share. From Fig. 11a, 11b, 11c, we observe that the sharing time presents a constant level. The average overhead of Fig. 11a, 11b, 11c, is separately $1.40 \times 10^{-5}$s, $1.43 \times 10^{-5}$s, $1.44 \times 10^{-5}$s, and the variance is separately $9.86 \times 10^{-14}$, $2.30 \times 10^{-13}$, $2.39 \times 10^{-13}$. Therefore, we confirm that the sharing time overhead on User is linear with the matching number of keywords in one Share.

Fig. 12 shows the sharing time overhead of Cloud B on each dataset in detail. Similarly, these graphs show the average cost of sharing an index according to the number of matches from one Share. We observe that each entry's average sharing time overhead decreases significantly with the increment of matching keywords. Fig. 12a, 12b, 12c show that the sharing time overhead drops to $3.44 \times 10^{-6}$, $1.37 \times 10^{-6}$, and $3.05\times10^{-6}$, respectively. Besides, the time overhead tends to be nearly constant on all datasets when the number of matching keywords exceeds 1,000. Therefore, we confirm that the sharing time overhead on Cloud B is sub-linear with the matching number of keywords in one Share.

### *F. Comparison With the Instances of MUSE and MKSE*

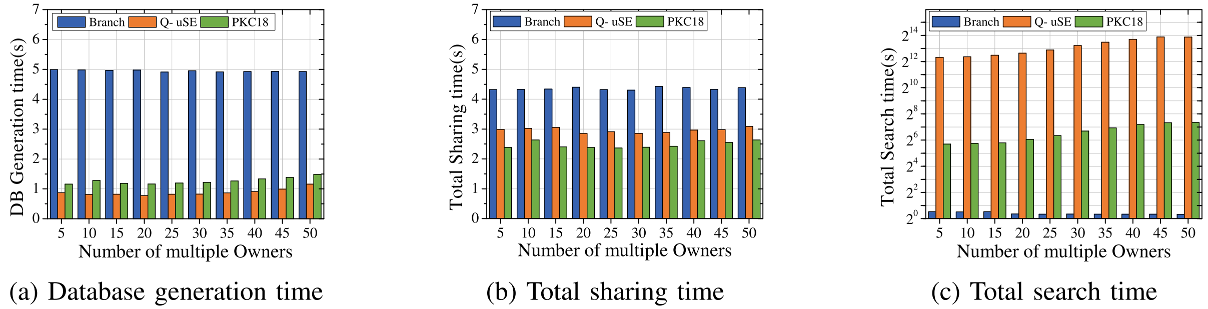
Fig. 13 shows the comparison results of KS$^2$E, MUSE, and MKSE instances in the setting of multiple Owners under LAN. We see that KS$^2$E is comparable to MKSE and MUSE on the time overheads of database generation and sharing. By Fig. 13a, we confirm that the database generation overheads are constant and they are not significantly affected by the number of Owners. In Fig. 13a, the overheads of Branch,

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.



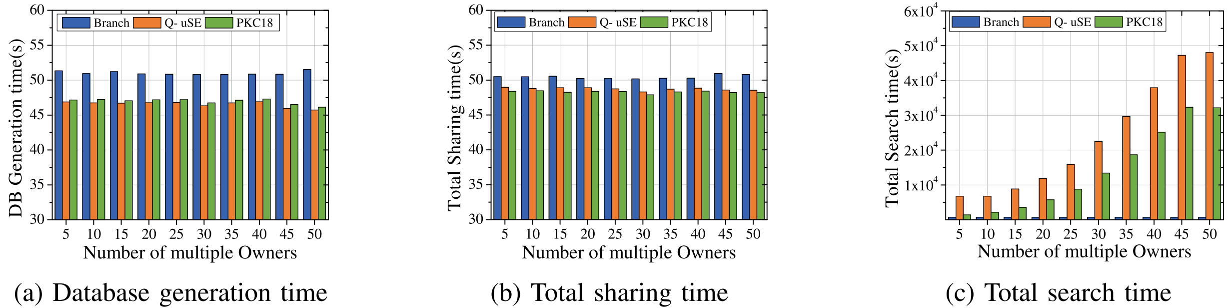

(a) Enron (b) Lucene (c) Wiki

Fig. 12. Sharing performance on Cloud B in LAN.




(a) Database generation time (b) Total sharing time (c) Total search time

Fig. 13. Comparison of performance between the instances of KS$^2$E, MUSE, and MKSE in LAN.




(a) Database generation time (b) Total sharing time (c) Total search time

Fig. 14. Comparison of performance between the instances of KS$^2$E, MUSE, and MKSE in WAN (average 23ms delay).

Q-$\mu$SE, and PKC18 generating the databases (with 2,000 files) are about 4.96s, 0.82s, and 1.14s, respectively.

From Fig. 13b, we see that the total overheads of KS$^2$E, MUSE, and MKSE instances are also constant which is not restricted by the number of Owners. In Fig. 13b, the sharing time overheads of Branch, Q-$\mu$SE, and PKC18 are about 4.33s, 2.95s, and 2.51s, respectively.

It should be noted that Fig. 13c shows our advantages over MUSE and MKSE in terms of search time overheads. Fig. 13c presents the total search overheads for searching all keywords from the database (with 2,000 files). The cost of Branch is much lower than that of Q-$\mu$SE and PKC18. Fig. 13c illustrates that the increasing number of Owners does harm the performance of MKSE and MUSE. We say that the KS$^2$E instance benefits from the bi-directional index to search the shared files efficiently.

Fig. 14 shows the comparison results for the KS$^2$E, MUSE, and MKSE instances in the context of multiple Owners under WAN with an average delay of 23ms. The instances also incur constant costs here. But it is clear that the gaps among them in Fig. 14a and 14b are smaller than those in Fig. 13. In Fig. 14c, the overhead of Branch is nearly 730.46s, while the costs of Q-$\mu$SE and PKC18 increase from 6,757.23s and 1,386.12s to 48,030.73s and 32,152.72s, respectively. We see that Branch still brings efficiency in the search.

## VI. Conclusion

In this paper, we propose the concept of KS$^2$E. By analyzing previous Searchable Encryption with sharing problems, we conclude that the forward and backward privacy and bi-directional index structure are the keys to our challenge. We propose the forward and backward privacy definitions of KS$^2$E and construct the instance Branch. From our security analysis and evaluation, we confirm that Branch can efficiently search and share search indexes while providing forward and backward privacy.

Although Branch provides all functions under the defined security model, the deletion operation still requires multiple instances consuming extra local users' computational costs in the sharing stage. We will aim to construct non-interactive KS$^2$E constructions with more efficient deletion.

## Appendix A

*Theorem 1 (Adaptive Security of KS$^2$E-Branch):* Let PRF $F$ with a specific key be modeled as the random oracle $\mathcal{H}_F$,

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.

**Game 3** $G_3$ Description

**Setup** $(\lambda)$ :
1: Generate empty maps $\mathrm{EDB}_\sigma, \mathrm{EDB}_{\sigma'}$ as encrypted database
2: Generate two timestamps $u \leftarrow 0, u' \leftarrow 0$

**Update** $(\sigma, (w, id, add); \mathrm{EDB}_\sigma)$ :
**Owner:**
1: $u \leftarrow u+1$
2: $(L, I_w, R_w, C_w, I_{id}, R_{id}, C_{id}) \xleftarrow{\$} \{0,1\}^\lambda \times \{0,1\}^{2\lambda} \times \{0,1\}^\lambda \times \mathcal{C}_w \times \{0,1\}^{2\lambda} \times \{0,1\}^\lambda \times \mathcal{C}_{id}$
3: $J_w \xleftarrow{\$} \{0,1\}^\lambda$, $J_{id} \xleftarrow{\$} \{0,1\}^\lambda$
4: If $\mathsf{Kid}[id]$ is $\perp$, then $k_{id} \xleftarrow{\$} \{0,1\}^\lambda$ and set $\mathsf{Kid}[id] \leftarrow k_{id}$, else $k_{id} \leftarrow \mathsf{Kid}[id]$
5: Set $\mathsf{Dec}[k_{id}, C_{id}] \leftarrow w$
6: Program $\mathcal{H}_F^1$ s.t. $\mathcal{H}_F^1(w||id) = L$
7: Program $\mathcal{H}_F^2$ s.t. $\mathcal{H}_F^2(w||id) = J_w, \mathcal{H}_F^2(id||w) = J_{id}$
8: Insert $(u, J_{id}, C_{w,id})$ to $\mathsf{UptIDHist}[id]$
9: $C_{w,id} \leftarrow (L, I_w, R_w, C_w, I_{id}, R_{id}, C_{id})$
10: **Send** $C_{w,id}$

**The view of Server:**
1: Parse $C_{w,id}$ as $(L, I_w, R_w, C_w, I_{id}, R_{id}, C_{id})$
2: $\mathrm{EDB}_\sigma[L] \leftarrow (I_w, R_w, C_w, I_{id}, R_{id}, C_{id})$

**Search** $(\sigma', w)$ :
**User:**
1: Read $\left((u^1, J_w^1, C_{w,id}^1), \ldots (u^j, J_w^j, C_{w,id}^j)\right)$ from $\mathsf{UpWHist}'[w]$
2: **for** $i = j$ to 1 **do**
3: Parse $C_{w,id}^i$ as $(L, I_w^i, R_w^i, C_w^i)$
4: **if** $\mathcal{H}\left(J_w^i, R_w^i\right) \neq \perp \wedge \mathcal{H}\left(J_w^i, R_w^i\right) \neq I_w^i \bigoplus \left(L^{i-1}||J_w^{i-1}\right)$ **then**
5: *set* **Break**
6: **else** Program $\mathcal{H}$ s.t. $\mathcal{H}\left(J_w^i, R_w^i\right) = I_w^i \bigoplus \left(L^{i-1}||J_w^{i-1}\right)$
7: **end if**
8: **end for**
9: Send search token $T_w \leftarrow \left(L^j, J_w^j\right)$ to Server

**The view of Server:**
1: $S \leftarrow \emptyset$
2: **while** $L$ is not $\perp$ **do**
3: Choose $(I_w, R_w, C_w)$ from $\mathrm{EDB}_{\sigma'}[L]$
4: $(L'||J_w') \leftarrow \mathcal{H}(J_w, R_w) \bigoplus I_w$
5: $S \leftarrow S \cup C_w$
6: Update $L \leftarrow L', J_w \leftarrow J_w'$
7: **end while**
8: Return $S$

**Sharetoken** $(\sigma, id)$ :
**Owner:**
1: $\left((u^1, J_{id}^1, C_{w,id}^1), \ldots (u^j, J_{id}^j, C_{w,id}^j)\right) \leftarrow \mathsf{UptIDHist}[id]$
2: Insert $u$ to $\mathsf{TimeST}[id]$
3: **for** $i = j$ to 1 **do**
4: Parse $C_{w,id}^i$ as $\left(L^i, I_w^i, R_w^i, C_w^i, I_{id}^i, R_{id}^i, C_{id}^i\right)$
5: **if** $\mathcal{H}\left(J_{id}^i, R_{id}^i\right) \neq \perp \wedge \mathcal{H}\left(J_{id}^i, R_{id}^i\right) \neq I_{id}^i \bigoplus \left(L^{i-1}||J_{id}^{i-1}\right)$ **then**
6: *set* **Break**
7: **else** Program $\mathcal{H}$ s.t. $\mathcal{H}\left(J_{id}^i, R_{id}^i\right) = I_{id}^i \bigoplus \left(L^{i-1}||J_{id}^{i-1}\right)$
8: **end if**
9: **end for**
10: Read $k_{id} \leftarrow \mathsf{Kid}[id]$
11: **Return** $\left(L^j, J_{id}^j, id, k_{id}\right)$

**Share** $(P_{id}, \sigma; \mathrm{EDB}_\sigma, \mathrm{EDB}_{\sigma'})$ :
**User:**
1: Parse $P_{id} = (L, J_{id}, id, k_{id})$
2: Send $D_{id} \leftarrow (L, J_{id})$

**The view of Server:**
1: $S \leftarrow \emptyset$
2: **while** $L$ is not $\perp$ **do**
3: Choose $(I_{id}, R_{id}, C_{id})$ from $\mathrm{EDB}_\sigma[L]$
4: $\left(L'||J_{id}'\right) \leftarrow \mathcal{H}(J_{id}, R_{id}) \bigoplus I_{id}$
5: $S \leftarrow S \cup C_{id}$
6: Update $L \leftarrow L', J_{id} \leftarrow J_{id}'$
7: **end while**
8: Return $S$

**User:**
1: $S' \leftarrow \emptyset$
2: **for** each symmetric ciphertext $C_{id}$ in $S$ **do**
3: Read $w \leftarrow \mathsf{Dec}[k_{id}, C_{id}]$
4: $u' \leftarrow max[\mathsf{TimeST}[id]]$
5: $(L, I_w, R_w, C_w) \xleftarrow{\$} \{0,1\}^\lambda \times \{0,1\}^{2\lambda} \times \{0,1\}^\lambda \times \mathcal{C}_w$
6: $J_w \xleftarrow{\$} \{0,1\}^\lambda$
7: Program $\mathcal{H}_F^{1'}$ s.t. $\mathcal{H}_F^{1'}(w||id) = L$
8: Program $\mathcal{H}_F^{2'}$ s.t. $\mathcal{H}_F^{2'}(w||id) = J_w$
9: $C_{w,id} \leftarrow (L, I_w, R_w, C_w)$, and insert $C_{w,id}$ to $S'$
10: Insert $(u', J_w, C_{w,id})$ to $\mathsf{UpWHist}'[w]$
11: **end for**
12: Send $S'$

**The view of Server:**
1: **for** each keyword ciphertext $C_{w,id}$ in $S'$ **do**
2: parse $C_{w,id}$ as $(L, I_w, R_w, C_w)$
3: $\mathrm{EDB}_{\sigma'}[L] \leftarrow (I_w, R_w, C_w)$
4: **end for**

and let cryptographic hash function $H$ be modeled as the random oracle $\mathcal{H}$. We define $\mathcal{L}_B = (\mathcal{L}_B^{Upt}, \mathcal{L}_B^{Shr}, \mathcal{L}_B^{Srch})$ as:

$$\mathcal{L}_B^{Upt}(w, id, op) = \emptyset$$
$$\mathcal{L}_B^{Shr}(id) = rp(id), \text{UpSHist}(id)$$
$$\mathcal{L}_B^{Srch}(w) = sp(w), \text{UpHist}(w)$$

Branch is $\mathcal{L}_B$-adaptively-secure.

*Proof:* We analyze the indistinguishability between the instance Branch and simulator $\mathcal{S}_{Branch}$, and we use the game hop method to analyze indistinguishability.

*Game $G_0$:* This game is identical with the real scheme Branch so that:

$$\mathbb{P}\left[\mathrm{Real}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\mathrm{KS^2E-Branch}}(\lambda) = 1\right] = \mathbb{P}[G_0 = 1]$$

*Game $G_1$:* In this game, we replace the PRF $F$ used in $G_0$ with a random oracle $\mathcal{H}_F$. The adversarial distinguishing advantage between $G_0$ and $G_1$ is exactly the distinguishability between PRF $F$ and $\mathcal{H}_F$. The adversary constructs a reduction $B_1$ making at most call on PRF $F$ with a specific key, then he can decide whether to call $F$ or get a random string from $\mathcal{H}_F$. In this condition, we have:

$$\mathbb{P}[G_0 = 1] - \mathbb{P}[G_1 = 1] \leq Adv_{F,B_1}^{prf}(\lambda)$$

*Game $G_2$:* In this game, we replace the symmetric ciphertexts $C_w, C_{id}$ with a random string from the space $\mathcal{C}_w, \mathcal{C}_{id}$. We can see that the distinguishability between game $G_1$ and game $G_2$ is from the symmetric ciphertexts $C_w, C_{id}$. The adversary constructs a reduction $B_2$ making at most call on CPA secure symmetric encryption (SE-CPA), then he can decide whether to call random string or symmetric ciphertexts. In this condition, we have:

$$\mathbb{P}[G_1 = 1] - \mathbb{P}[G_2 = 1] \leq Adv_{\mathcal{E},B_2}^{SE-CPA}(\lambda)$$

*Game $G_3$:* In this game, we replace $I_w, I_{id}$ with random string, then we use random oracle $\mathcal{H}$ to replace the cryptographic hash function $H$, which is programmed to hold the correctness of search and sharing. In this game, table $\mathsf{UptIDHist}[id]$ records the history of the $id$ when updating the search index. Table $\mathsf{UpWHist}'[w]$ records the history of the $w$ when re-encrypting search indexes. From this game, we can see that the distinguishability between game $G_2$ and game $G_3$ is from the cryptographic hash function value $I_w, I_{id}$. To break the program on oracle $\mathcal{H}$, the adversary constructs a reduction $B_3$ from a distinguisher $\mathcal{A}$ which inserts $N$ $(w, id)$ pairs in encrypted database. $B_3$ firstly guesses which $(w^*, id^*)$ pair will be set to **Break** for the first time among the $N$ pairs.

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.

For $(w^*, id^*)$, reduction $B_3$ will call its oracle $\mathcal{H}_F^{2^*}$ to evaluate the same $J_w^*$ or $J_{id}^*$ at $q$ times and try to make a **Break** on $\mathcal{H}$. If we assume the random $R_w, R_{id}$ are uniformly random, the probability of $\mathcal{H}$ called on $(J_w^*, R_w)$ and $(J_{id}^*, R_{id})$ is $2 \cdot q \cdot 2^{-\lambda}$ so that:

$$Pr\,[set\ \textbf{Break}] \leq Adv_{F,B_3}^{prf}(\lambda) + \frac{2q}{2^\lambda}$$

As guessing the pair $(w^*, id^*)$ has $N$ loss in the advantage of the reduction $B_3$, we have:

$$\mathbb{P}[G_2 = 1] - \mathbb{P}[G_3 = 1] \leq N \cdot Adv_{F,B_3}^{prf}(\lambda) + \frac{2Nq}{2^\lambda}$$

*The Simulator:* We slightly replace the $w$, $id$ used in $G_3$ with the *min* pattern $sp(w)$ and $rp(id)$ to hold Search and Share, and we can observe that UptIDHist$[id]$ leaks the UpSHist$(id)$ to recover the previous search index by $D_{id}$. Besides, the UpWHist$'[w]$ reveals the max[TimeST$[id]$] and historical entries left in the current database. When search query finds a record based on keyword $w$, the last sharing timestamp of $id$ is also revealed. In this condition, we can note UpWHist$'[w]$ as UpHist$(w)$, and we have:

$$\mathbb{P}[G_3 = 1] - \mathbb{P}\left[\text{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\text{KS}^2\text{E-Branch}}(\lambda) = 1\right] = 0$$

*Conclusion:* By combining all contributions from all games, there exists the adversary such that:

$$|\mathbb{P}[\text{Real}_{\mathcal{A}}^{\text{KS}^2\text{E-Branch}}(\lambda) = 1] - \mathbb{P}[\text{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\text{KS}^2\text{E-Branch}}(\lambda) = 1]| \leq Adv_{F,B_1}^{prf}(\lambda) + Adv_{\mathcal{E},B_2}^{SE-CPA}(\lambda) + N \cdot Adv_{F,B_3}^{prf}(\lambda) + \frac{2Nq}{2^\lambda}$$

□

## Appendix B

*Claim 1 (The correctness of KS$^2$E-Branch):* Based on the birthday problem, the Branch instance can make the collision resistance by making the $1 - e^{\frac{-\mu^2}{2^{\lambda+1}}}$ negligible, where $\mu$ represents the maximum number of inputs to the PRF $F$.

*Proof:* Based on the birthday problem, we see that the probability of all $\mu$ inputs occur collision after hashing is:

$$\mathbb{P}_{Coll} = 1 - 1 \cdot \left(1 - \frac{1}{2^\lambda}\right) \ldots \left(1 - \frac{\mu - 1}{2^\lambda}\right)$$

Base on the Taylor expansion for $e^x$, we have the following equation when $x$ is around 0:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \ldots \approx 1 + x$$

Following this Taylor expansion, we can approximately obtain:

$$\mathbb{P}_{Coll} \approx 1 - e^{-\frac{1}{2^\lambda}} \cdot e^{-\frac{2}{2^\lambda}} \ldots e^{-\frac{\mu-1}{2^\lambda}} = 1 - e^{-\frac{\mu(\mu-1)}{2 \cdot 2^\lambda}}$$

For simplicity, we can approximately obtain the collision probability when $\mu$ is small relative to $2^\lambda$:

$$\mathbb{P}_{Coll} = 1 - e^{\frac{-\mu^2}{2^{\lambda+1}}}$$

□


## Acknowledgment

The authors would like to thank the reviewers for their valuable suggestions that helped to improve the article greatly.


## References


[1] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. ACM Conf. Comput. Commun. Secur.*, Oct. 2012, pp. 965–976.

[2] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 72–75.

[3] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1465–1482.

[4] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *Advances in Cryptology—EUROCRYPT*. Berlin, Germany: Springer, May 2004, pp. 506–522.

[5] R. A. Popa and N. Zeldovich, "Multi-key searchable encryption," *IACR Cryptol. ePrint Arch.*, vol. 2013, p. 508, Aug. 2013.

[6] A. Hamlin, A. Shelat, M. Weiss, and D. Wichs, "Multi-key searchable encryption, revisited," in *Public-Key Cryptography—PKC*. Cham, Switzerland: Springer, Mar. 2018, pp. 95–124.

[7] D. G. Amalarathinam and J. M. Priya, "Survey on data security in multi-cloud environment," *Int. J. Pure Appl. Math.*, vol. 118, no. 6, pp. 323–334, 2018.

[8] T. Shi, H. Ma, G. Chen, and S. Hartmann, "Cost-effective web application replication and deployment in multi-cloud environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 8, pp. 1982–1995, Aug. 2022.

[9] R. Potharaju et al., "Hyperspace: The indexing subsystem of Azure synapse," *Proc. VLDB Endowment*, vol. 14, no. 12, pp. 3043–3055, Jul. 2021.

[10] L. Chen, W.-K. Lee, C.-C. Chang, K.-K.-R. Choo, and N. Zhang, "Blockchain based searchable encryption for electronic health record sharing," *Future Gener. Comput. Syst.*, vol. 95, pp. 420–429, Jun. 2019.

[11] J. G. Chamani, Y. Wang, D. Papadopoulos, M. Zhang, and R. Jalili, "Multi-user dynamic searchable symmetric encryption with corrupted participants," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 1, pp. 114–130, Jan. 2023.

[12] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in *Proc. 13th ACM Conf. Comput. Commun. Secur.*, Oct. 2006, pp. 79–88.

[13] S. Patel, G. Persiano, and K. Yeo, "Symmetric searchable encryption with sharing and unsharing," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2018, pp. 207–227.

[14] B. Cui, Z. Liu, and L. Wang, "Key-aggregate searchable encryption (KASE) for group data sharing via cloud storage," *IEEE Trans. Comput.*, vol. 65, no. 8, pp. 2374–2385, Aug. 2016.

[15] J. Wang and S. S. M. Chow, "Omnes pro uno: Practical multi-writer encrypted database," in *Proc. USENIX Secur. Symp.*, 2022, pp. 2371–2388.

[16] D. Xiaoding Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. IEEE Symp. Secur. Privacy*, May 2000, pp. 44–55.

[17] G. Chen, T.-H. Lai, M. K. Reiter, and Y. Zhang, "Differentially private access patterns for searchable symmetric encryption," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2018, pp. 810–818.

[18] L. Xu, X. Yuan, C. Wang, Q. Wang, and C. Xu, "Hardening database padding for searchable encryption," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2019, pp. 2503–2511.

[19] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 1–51, Jan. 2015.

[20] B. Fuller et al., "Sok: Cryptographically protected database search," in *Proc. IEEE Symp. Security Privacy*, May 2017, pp. 172–191.

[21] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *Proc. ACM CCS*, 2016, pp. 1143–1154.

[22] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1038–1055.

[23] S.-F. Sun et al., "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 763–780.

[24] S.-F. Sun et al., "Practical non-interactive searchable encryption with forward and backward privacy," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.


Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.

[25] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.

[26] T. Chen, P. Xu, W. Wang, Y. Zheng, W. Susilo, and H. Jin, "Bestie: Very practical searchable encryption with forward and backward security," in *Computer Security—ESORICS*. Cham, Switzerland: Springer, Oct. 2021, pp. 3–23.

[27] P. Xu et al., "ROSE: Robust searchable encryption with forward and backward security," *IEEE Trans. Inf. Forensics Security*, vol. 17, pp. 1115–1130, 2022.

[28] K. He, J. Chen, Q. Zhou, R. Du, and Y. Xiang, "Secure dynamic searchable symmetric encryption with constant client storage cost," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 1538–1549, 2021.

[29] J. Alderman, K. M. Martin, and S. L. Renwick, "Multi-level access in searchable symmetric encryption," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2017, pp. 35–52.

[30] Y. Wang and D. Papadopoulos, "Multi-user collusion-resistant searchable encryption with optimal search time," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, May 2021, pp. 252–264.

[31] A. Kiayias, O. Oksuz, A. Russell, Q. Tang, and B. Wang, "Efficient encrypted keyword search for multi-user data sharing," in *Computer Security—ESORICS*. Cham, Switzerland: Springer, Sep. 2016, pp. 173–195.

[32] S.-F. Sun, J. K. Liu, A. Sakzad, R. Steinfeld, and T. H. Yuen, "An efficient non-interactive multi-client searchable encryption with support for Boolean queries," in *Computer Security—ESORICS*. Cham, Switzerland: Springer, Sep. 2016, pp. 154–172.

[33] R. A. Popa et al., "Building web applications on top of encrypted data using mylar," in *Proc. NSDI*, 2014, pp. 157–172.

[34] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov, "Breaking web applications built on top of encrypted data," in *Proc. ACM CCS*, 2016, pp. 1353–1364

[35] Y. Su, J. Wang, Y. Wang, and M. Miao, "Efficient verifiable multi-key searchable encryption in cloud computing," *IEEE Access*, vol. 7, pp. 141352–141362, 2019.

[36] C. Hahn, H. Yoon, and J. Hur, "Multi-key similar data search on encrypted storage with secure pay-per-query," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 1169–1181, 2023.

[37] W. Wang, P. Xu, L. T. Yang, and J. Chen, "Cloud-assisted key distribution in batch for secure real-time mobile services," *IEEE Trans. Services Comput.*, vol. 11, no. 5, pp. 850–863, Sep. 2018.

[38] D. Cash et al., "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 23–26.

[39] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *Proc. USENIX Secur. Symp.*, 2016, pp. 707–720.

[40] I. Mironov, "Hash functions: Theory, attacks, and applications," Microsoft Res. Silicon Valley Campus, Mountain View, CA, USA, Nov. 2005. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2005/11/hash_survey.pdf

[41] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1449–1463.

[42] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, Mar. 1980.

[43] G. Attardi. (2016). *Wikipedia Extractor*. [Online]. Available: http://medialab.di.unipi.it/wiki/Wikipedia_Extractor

[44] N. Project. (2016). *Natural Language Toolkit*. [Online]. Available: http://www.nltk.org

[45] K. Banker, D. Garrett, P. Bakkum, and S. Verch, *MongoDB in Action: Covers MongoDB Version 3.0*. New York, NY, USA: Simon and Schuster, 2016.

[46] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 388–402, Jun. 2004.

[47] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *J. ACM*, vol. 33, no. 4, pp. 792–807, Aug. 1986.

[48] A. Kiayias, S. Papadopoulos, N. Triandopoulos, and T. Zacharias, "Delegatable pseudorandom functions and applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 669–684.

**Wei Wang** (Member, IEEE) received the B.E. and Ph.D. degrees in electronic and communication engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2006 and 2011, respectively. She is currently an Associate Professor with the Cyber-Physical-Social Systems Laboratory, Huazhong University of Science and Technology. She has authored 30 papers in international journals and conferences. Her current research interests include cryptography and data privacy.

**Dongli Liu** received the B.E. degree in information security from the Huazhong University of Science and Technology, Wuhan, China, in 2018, where he is currently pursuing the Ph.D. degree in cyberspace security with the School of Computer Science and Technology. His current research interests include cryptography, particularly encryption, encrypted database, and security protocols.

**Peng Xu** (Member, IEEE) received the Ph.D. degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2010. He was a Post-Doctoral Researcher with the Huazhong University of Science and Technology from 2010 to 2013 and an Associate Research Fellow with the University of Wollongong, Australia, from 2018 to 2019. He is currently a Full Professor with the Huazhong University of Science and Technology. He was a PI in ten grants, including four NSF projects. He has authored 50 research articles and three books and holds 20 patents. His current research interests include cryptography.

**Laurence Tianruo Yang** (Fellow, IEEE) received the B.E. degree in computer science and technology and the B.Sc. degree in applied physics from Tsinghua University, Beijing, China, in 1992, and the Ph.D. degree in computer science from the University of Victoria, Victoria, BC, Canada, in 2006. He is currently a Professor with the School of Computer Science and Technology, Huazhong University of Science and Technology, China; the School of Computer Science and Technology, Hainan University, China; and the Department of Computer Science, St. Francis Xavier University, Canada. His research has been supported by the National Sciences and Engineering Research Council, Canada, and the Canada Foundation for Innovation. His current research interests include parallel and distributed computing, embedded and ubiquitous/pervasive computing, and big data.

**Kaitai Liang** (Member, IEEE) received the Ph.D. degree in computer science from the Department of Computer Science, City University of Hong Kong, Hong Kong, in 2014. He is currently an Assistant Professor with the Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Delft, The Netherlands. His current research interests include applied cryptography and information security, particularly encryption, blockchain, postquantum crypto, privacy-enhancing technology, and security in cloud computing.

Authorized licensed use limited to: TU Delft Library. Downloaded on October 09,2023 at 08:14:18 UTC from IEEE Xplore. Restrictions apply.