MUSES: Efficient Multi-User Searchable Encrypted Database

Tung Le
Virginia Tech

Jorge Guajardo
Robert Bosch LLC — RTC

Rouzbeh Behnia
University of South Florida, Sarasota

Thang Hoang
Virginia Tech

ABSTRACT

Searchable encrypted systems enable privacy-preserving keyword search on encrypted data. Symmetric Searchable Encryption (SSE) achieves high security (e.g., forward privacy) and efficiency (i.e., sublinear search), but it only supports single-user. Public Key Searchable Encryption (PEKS) supports multi-user settings, however, it suffers from inherent security limitations such as being vulnerable to keyword-guessing attacks and the lack of forward privacy. Recent work has combined SSE and PEKS to achieve the best of both worlds: support multi-user settings, provide forward privacy while having sublinear complexity. However, despite their elegant design, the existing hybrid scheme inherits some of the security limitations of the underlying paradigms (e.g., patterns leakage, keyword-guessing) and might not be suitable for certain applications due to costly public-key operations (e.g., bilinear pairing).

In this paper, we propose MUSES, a new multi-user encrypted search scheme that addresses the limitations in the existing hybrid design, while offering user efficiency. Specifically, MUSES permits multi-user functionalities (reader/writer separation, permission revocation), prevents keyword-guessing attacks, protects search/result patterns, achieves forward/backward privacy, and features minimal user overhead. In MUSES, we demonstrate a unique incorporation of various state-of-the-art distributed cryptographic protocols including Distributed Point Function, Distributed PRF, and Secret-Shared Shuffle. We also introduce a new oblivious shuffle protocol for the general L-party setting with dishonest majority, which can be of independent interest. Our experimental results indicated that the keyword search in our scheme is two orders of magnitude faster with 13× lower user bandwidth overhead than the state-of-the-art.

KEYWORDS

Privacy-enhancing technologies; encrypted search; data privacy.

1 INTRODUCTION

Data outsourcing services (e.g., Dropbox, Google Drive, MS OneDrive) have been increasingly prevalent because of their accessibility and convenience. Commodity cloud storage (e.g., AWS IAM, Google Cloud IAM) not only can provide users with data storage facilities, but also support a fine-grained access control for data sharing across a large number of users. Nevertheless, outsourcing data to external clouds might lead to privacy concerns, especially for sensitive data (e.g., health and financial records). This is because a compromised cloud provider can access and exploit data illegitimately. Although end-to-end encryption can enable data confidentiality, it also prevents data utility (e.g., querying, analytics), thereby invalidating the benefits of data outsourcing services.

To address the data utilization and privacy dilemma, Searchable Encryption (SE) was proposed to enable keyword search on encrypted data while respecting the confidentiality of the data and the search query. There are two main lines of SE research including Symmetric SE (SSE) [10, 25, 27, 37, 42, 46, 67] and Public-Key SE (PEKS) [5, 8, 77]. While SSE offers high security guarantees (e.g., forward privacy [10, 37, 53, 68, 72], backward privacy [37, 53, 68, 72]), efficiency (e.g., sublinear search), and diverse query functionalities (e.g., range query [26, 50, 72]), it mainly supports the single-user setting, in which the data can only be searched by its owner. This strictly limits its practicality when adopted in real-world settings, where the data can be contributed by multiple users (e.g., emails). On the other hand, PEKS enables encrypted search in the multi-user setting, in which one user (the reader) can search on encrypted documents sent/shared by the other users (the writers). Unfortunately, PEKS is known to suffer from various inherent security flaws including the lack of forward privacy and vulnerability to dictionary attacks. Meanwhile, forward privacy has been shown (via practical attack demonstrations [79]) to become a de facto requirement in SE for long-term security.

Recently, Wang et al. [73] proposed the notion of Hybrid SE (HSE), which elegantly combines SSE and PEKS to achieve the benefits of both SE paradigms: forward privacy and sublinear complexity in SSE, and multi-user functionalities in PEKS. Despite its merits and elegant design, the proposed HSE scheme accidentally inherits the security weaknesses of both worlds, including the keyword-guessing attack (KGA) vulnerabilities and search/result pattern leakage. In addition, the proposed HSE instantiation does not achieve backward privacy [11], which is necessary to prevent extra information leakage during search. Many devastating attacks (e.g., [4, 14, 44, 48, 52, 54, 56, 62, 63]) have shown that search/access pattern leakage reveals significant information about the query and the data even though they are both encrypted. While some techniques such as Oblivious RAM (oram) [66] can hide the search/result pattern in SE, they incur high bandwidth cost to the user(s) [61]. Given that HSE is still in the early stages and there is a lack of privacy-focused designs, we raise the following challenging and practically relevant question:

Can we design a new SSE scheme that not only supports multi-writer similar to HSE, but also addresses the inherent security limitations of the existing paradigms (i.e., in both SSE and PEKS), while achieving concrete efficiency?
# Servers

O(\(\log m\)) when \(L = 2\), or \(O(\sqrt{n})\) when \(L \geq 3\).

We assume document identifiers can be represented using a constant number of bits to skip this quantity in search and update complexity.

‡ Our FP-HSE is designed with user efficiency in mind, and therefore, it is highly favorable to thin users with limited computing and network resources (e.g., mobile). In MUSES, the reader only performs lightweight operations (e.g., modular additions), and the search bandwidth in MUSES is proportional to the number of matched documents, compared with linear w.r.t. entire database in prior oblivious SE schemes (e.g., [25, 32, 42]). Evaluation results indicate that MUSES achieves up to 13x lower reader-bandwidth than state-of-the-art oblivious SE designs. On the other hand, the writer in MUSES can revoke the reader’s access efficiently by offloading entire re-encryption task securely to the servers. This is more efficient than prior systems that do not naturally support revocation and require writers to re-encrypt the index themselves, which incurs high bandwidth for index transmission and computation cost.

Low server processing overhead: In MUSES, the servers only perform low-cost operations (e.g., modular addition/multiplication, rounding over small modulus). Therefore, it is more efficient than prior designs that incur costly public-key operations (e.g., pairing [5, 6, 8, 30, 73, 77, 78]).

• **Fully-fledged implementation and evaluation:** We fully implemented our MUSES scheme and evaluated its performance on commodity servers. Experimental results demonstrate that our technique performs search 129.1×–137.2× faster than the state-of-the-art multi-user SE technique (i.e., [73]). MUSES is also faster than single-user SE counterparts (around 1.6×–1.7× faster than [25]) under limited bandwidth settings. Our implementation is available and ready to be publicly released for reproducibility, comparison, and adaptation (see the attached artifact).

**Technique: Our Multi-party secret-shared shuffle.** To construct a new multi-user SE with user-driven efficiency, we come up with a new oblivious shuffle technique that can be of independent interest and can lead to other interesting applications. Specifically, we construct a generic multi-party secret-shared shuffle extended from the specific two-party one in [18], which permits multiple parties to obtain randomly shuffled data from their additive shares with dishonest majority.

Table 1 compares MUSES with the state-of-the-art SE designs in terms of security, functionality and complexity. To our knowledge, we are the first to propose a multi-user SE scheme that can achieve small leakage and high efficiency (optimal user bandwidth, low reader and writer overhead) simultaneously.

**Why distributed servers?** One might wonder why MUSES makes use of multiple servers rather than a single server as commonly used in SE platforms (e.g., [8, 15, 73]). This is due to the fact that in the multi-user setting with separate reader and writer roles, there are inherent security vulnerabilities that cannot be prevented with a single server. For instance, in the “rollback” attacks [33, 43, 47, 55, 59, 71], the malicious server can omit some update of the writer, and present the old version of the writer’s data to the reader. Preventing such realistic attacks requires coordination and communication between the reader and the writer, or a separate system for integrity such as blockchain [43, 71]. Given that it is costly to deploy such a dedicated system, we use multiple servers to not only prevent these attacks but also gain performance benefits of underlying cryptographic building blocks (e.g., Distributed Point Function, distributed PRF) used in our scheme. Also note that although we only present the semi-honest MUSES scheme in the main body, it

---

### Table 1: Comparison of MUSES with prior encrypted search systems.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>#Servers</th>
<th>Search Leakage</th>
<th>Update Leakage</th>
<th>Multi-Writer</th>
<th>Search Complexity</th>
<th>Update Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE [15]</td>
<td>1</td>
<td>({s_p(w), r_p(w), s_v(w)})</td>
<td>({u_p(w)})</td>
<td></td>
<td>(O(n_s))</td>
<td>(O(\lambda))</td>
</tr>
<tr>
<td>PERS [8]</td>
<td>1</td>
<td>({w, s_p(w), r_p(w), s_v(w)})</td>
<td>({u_p(w)})</td>
<td></td>
<td>(O(\sqrt{n_s}N))</td>
<td>(O(\lambda))</td>
</tr>
<tr>
<td>DORY [25]</td>
<td>2</td>
<td>({u_p(w)})</td>
<td></td>
<td></td>
<td>(O(mN))</td>
<td>(O(m))</td>
</tr>
<tr>
<td>FF-HSE [73]</td>
<td>1</td>
<td>({w, W, s_p(w), r_p(w), s_v(w)})</td>
<td>({u, u_p(w)})</td>
<td></td>
<td>(O(W^t))</td>
<td>(O(W))</td>
</tr>
<tr>
<td>Our MUSES</td>
<td>≥ 2</td>
<td>({W, s_v(w)})</td>
<td>({u, u_p(w)})</td>
<td></td>
<td>(O(\sqrt{n_s}N))</td>
<td>(O(m))</td>
</tr>
</tbody>
</table>

- \(L_{\text{Search}}\): Search leakage function with \(w\) as the input keyword and \(W\) as the writer subset when a search happens; \(L_{\text{Update}}\): Update leakage function with \(i\) as the identifier of the writer, \(op\) as the operation type (add/delete), \(u\) as the updated document, and \(w\) (resp. \(w\)) is the keyword to be added/deleted (resp. a vector of keywords in the updated document). \(s_p\): Search pattern; \(r_p\): Result pattern; \(w\): Search volume; \(u\): Document update pattern; \(\Omega\) is a sequence of operations and \((t, w)\) as a search on keyword \(w\) at the timestamp \(t\) (see §3 for more details). \(\lambda\): Security parameter; \(N\): Total number of documents; \(d_w\): Number of keywords per document. \(W\): Total number of unique keywords over all documents (keyword universe). \(m\): Size of keyword representation per document; \(n_s\): Number of documents matched per keyword search. In practice, \(d_w < m \ll W, n_s \ll N\). Number of updated keywords (added/deleted). Number of servers \(L\) is considered as a constant number in complexity analysis.

1.1 Our Contributions

We answer the above question affirmatively by proposing MUSES, a new distributed multi-user SE database scheme that achieves a high level of security with concrete efficiency simultaneously. MUSES achieves the following desirable properties.

- **Multi-user functionalities:** MUSES allows multiple users with reader and writer(s) separation similar to multi-user PEKS/HSE schemes (e.g., [6, 8, 30, 73, 78]). MUSES enables the writers to update their data that can be searched by the reader. MUSES also permits the writers to revoke the reader’s search permission if necessary with writer-efficiency.

- **High security:** MUSES is secure against KGA, offers forward and backward privacy, and hides search/result patterns simultaneously. Thus, it offers a much higher security guarantee than most prior multi-user SE schemes ([8, 49, 73, 75, 77]). Our MUSES offers semi-honest security with dishonest majority. It can also support any number \(L\) of servers with \(L - 1\) privacy degree guarantee, meaning the confidentiality of users’ data and queries is protected as long as one (out of \(L\)) server is honest. To our knowledge, there is no prior PEKS/HSE scheme that achieves all the aforementioned security properties.

- **User-driven efficiency:** MUSES is designed with user efficiency in mind, and therefore, it is highly favorable to thin users with limited computing and network resources (e.g., mobile). In MUSES, the reader only performs lightweight operations (e.g., modular additions), and the search bandwidth in MUSES is proportional to the number of matched documents, compared with linear w.r.t. entire database in prior oblivious SE schemes (e.g., [25, 32, 42]). Evaluation results indicate that MUSES achieves up to 13x lower reader-bandwidth than state-of-the-art oblivious SE designs. On the other hand, the writer in MUSES can revoke the reader’s access efficiently by offloading entire re-encryption task securely to the servers. This is more efficient than prior systems that do not naturally support revocation and require writers to re-encrypt the index themselves, which incurs high bandwidth for index transmission and computation cost.

- **Low server processing overhead:** In MUSES, the servers only perform low-cost operations (e.g., modular addition/multiplication, rounding over small modulus). Therefore, it is more efficient than
is straightforward to make it secure against rollback attacks (by using more servers). We present such an extension in Appendix C.

1.2 Technical Highlights

We present the technical highlights of our construction. MUSES is inspired by DORY [25], a symmetric SE scheme, and HSE [73], a framework that provides a generic transformation guideline to adopt symmetric SE in the multi-user setting. We begin by giving DORY’s overview, outline the challenges when transforming DORY to multi-user setting, and then present our high-level idea to address these challenges.

**Brief Overview of DORY.** DORY is a (single-client) encrypted search scheme that supports oblivious search and update capabilities. Its high-level idea is to instantiate the index for searchable keyword representation with a table data structure, in which the keyword search operation occurs in one dimension (e.g., column), while the document update occurs in the other dimension (i.e., row).

To reduce the index size, Bloom Filter (BF) is employed to compress the keyword representation in the document, resulting in the total index size of $O(N \cdot m)$, where $N$ and $m$ is the number of documents and the BF representation size, respectively. For confidentiality, the search index is row-wise encrypted for efficient update. To search for a keyword $w$, the user computes the BF representation of $w$, and uses Private Information Retrieval (PIR) based on Distributed Point Function (DPF) to retrieve $K$ encrypted columns in the search index that are indicated by the BF representation (i.e., $K$ is the number of elements “1” in the BF representation). To update a document, the user replaces the corresponding row in the search index with a new (encrypted) BF representation for all the keywords in the updated document.

In principle, DORY can be extended to support multi-user setting (separate reader/writer) by incorporating public-key cryptography (as suggested in [73]) to distribute the key that is used to encrypt the search index to the reader. However, it incurs inherent performance limitations due to its underlying cryptographic building blocks.

Specifically, the reader incurs a linear complexity with respect to the document collection size ($O(N \cdot K)$) in terms of both network bandwidth and computation overhead (due to the PIR, aggregation, and decryption sequence) to obtain the search result. This overhead is significant especially in the context of thin reader (as suggested in [73]) to distribute the key that is used to encrypt the separate reader/writer by incorporating public-key cryptography processing tasks to the server in a privacy-preserving manner. Specifically, we utilize multiple servers and develop a new protocol that is well-coordinated with Key-Homomorphic Pseudorandom Function (KH-PRF) [9] and DPF-based PIR [12, 13, 38] together, which permits each server to “partially” decrypt the encrypted columns and perform secure aggregation, respectively. At the end of our protocol, each server obtains a share of the final search result and therefore, they can exchange their shares together to reconstruct the result and return it to the reader. Our protocol ensures the servers do not learn anything (e.g., what columns are being aggregated/decrypted, decryption keys) apart from the final search result, given that all of them do not collude with each other.

While this strategy reduces the reader’s processing and bandwidth overhead, it permits the server to learn the particular document identifiers that match the search query. These so-called result patterns permit an adversarial server to infer the search pattern (e.g., whether the same/different keywords are being searched). **Can we hide such result patterns while still maintaining the reader efficiency?**

**Idea 2: Conceal result/search patterns via random shuffling.** To hide result patterns, our idea is to perform a random shuffling on the shares of the (aggregated decrypted) search result across multiple servers before they come together to open the final output.

We construct a generic L-party secret-shared shuffling technique with dishonest majority based on [18], which enables $L$ parties to randomly shuffle secret-shared data in a way that one party learns the final shuffled result, while each other party learns a permutation in a composition of $L−1$ permutations. We apply our L-party shuffle protocol to obfuscate the order of the search result, where one server learns the obfuscated document identifiers and each of the other servers learns a permutation in a permutation composition. Finally, the servers can individually send the obfuscated list and the permutation information to the reader so that he can obtain the final search result by computing the permutation inverse on the obfuscated list. This strategy slightly increases the computation overhead for the reader due to permutation inversion; however, the communication complexity $O(n_s)$ (where $n_s \ll N$ is the number of matched documents) is still maintained.

**Idea 3: Minimize writer overhead in revoking reader’ permission via “key rotation” on the servers.** To revoke the reader’s search ability, we re-encrypt the writer’s search index on the servers with fresh keys unknown to the reader. At a high level, we incorporate the homomorphic property of KH-PRF with random masking techniques, which enables the servers to “rotate” the index that is currently encrypted by the old KH-PRF keys to the new ones on behalf of the writer in a privacy-preserving manner. The writer only needs to share the old and the new fresh KH-PRF keys with the servers, and does not need to stay involved in the later process.

2 PRELIMINARIES

**Notation.** $||$ denotes the concatenation operator. We denote by $\lambda$ the security parameter and by $\mathbb{Z}_p$ the ring of integers modulo $p$.

We denote by $[n]$ the set $\{1, \ldots, n\}$. $x \leftarrow [n]$ means $x$ is selected uniformly at random from the set $\{1, \ldots, n\}$. For integers $q$ and $p$ where $q \geq p \geq 2$, we define $\lceil \cdot \rceil_p : \mathbb{Z}_q \rightarrow \mathbb{Z}_p$ as a rounding function as $\lfloor x \rfloor_p = i$ where $i \cdot [q/p]$ is the largest multiple of $[q/p]$ that

\footnote{A permutation composition is formed by $L−1$ separate permutations $\pi_1, \ldots, \pi_{L−1}$ applied in sequence to a data vector $d$ to be shuffled as: $\pi_{L−1}(\pi_{L−2}(\cdots(\pi_1(d))\cdots))$.}
does not exceed $x$. Bold small letters denote vectors, i.e., $a \in \mathbb{Z}_p^n$. We denote by $(a, b)$ the dot product of two vectors $a$ and $b$. Capitalized bold letters denote matrices, i.e., $M \in \mathbb{Z}_p^{m \times n}$. Given a matrix $M$, $M[i, \ast]$ and $M[\ast, j]$ denote accessing the row $i$ and column $j$ of $M$, respectively. $M[i, j]$ denotes accessing the cell indexed at row $i$ and column $j$. We denote $\pi$ as a permutation and $\pi^{-1}$ as its inverse such that $\pi^{-1}(\pi(x)) = x$. We denote the execution of protocol $A$ by $L$ parties $(\alpha_1, \alpha_2, \ldots, \alpha_L) \leftarrow A(i_1, i_2, \ldots, i_L)$, where the input/output of each party is separated by a semicolon ($;$).

Let $E = (Gen, Enc, Dec)$ be an IND-CPA symmetric encryption scheme: $\kappa \leftarrow Gen(\lambda)$ generating a key with security parameter $\lambda$; $c \leftarrow E. Enc(k, ct, m)$ encrypting plaintext $m$ with key $k$ and counter $ct$; $m \leftarrow E. Dec(k, ct, c)$ decrypting ciphertext $c$ with key $k$ and counter $ct$. Let $\Pi = (Gen, Enc, Dec)$ be a public-key encryption scheme: $(pk, sk) \leftarrow Gen(\lambda)$ generating a public and private-key pair with security parameter $\lambda$; $c \leftarrow \Pi. Enc(pk, m)$ encrypting plaintext $m$ with public key $pk$; $m \leftarrow \Pi. Dec(sk, c)$ decrypting ciphertext $c$ with private key $sk$. Let $BF = (Gen, Priv, Eval)$ be a Bloom Filter (BF) [7]: $(H_1, \ldots, H_k) \leftarrow BF. Init(m, K)$: generating $K$ mappings $H_k : S \rightarrow [m] \forall k \in [K]$ with two parameters $m$ (BF size) and $K$; $u \leftarrow BF. Gen(S)$: computing the BF representation $u \in \{0, 1\}^m$ of a given set $S$; $(0, 1) \leftarrow BF. Priv(u, S)$: checking whether an element $s$ belongs to the set represented by $BF$ vector $u$.

**Secret Sharing.** Secret sharing enables a secret to be shared among $L$ parties. We denote $x^{(i)}$ as the additive share of a secret $x \in \mathbb{Z}_p$ to party $i$ such that $x = \sum_{i=1}^{L} x^{(i)} \mod p$.

**Bit Operations.** We denote $\oplus$ and $\otimes$ as the bitwise XOR and AND operations, respectively. $x \ll t$ and $x \gg t$ denote left-shift and right-shift operations by $t$ bits of value $x$.

### 2.1 Distributed Point Function

Distributed Point Function (DPF) [12, 13, 38] permits $L$ parties to jointly evaluate a point function. For $a, b \in \{0, 1\}^*$, let $P_{ab} : \{0, 1\}^{|a|} \rightarrow \{0, 1\}$ such that $P_{ab}(a) = b \wedge P_{a,b}(d') = 0|b| \forall d' \neq a$. A DPF scheme contains the following PPT algorithms.

- $(k^{(1)}, \ldots, k^{(L)}) \leftarrow DPF. Gen(\lambda, a, b)$: Given security parameter $\lambda$, and values $a, b \in \{0, 1\}^*$, it outputs $L$ keys $k^{(1)}, \ldots, k^{(L)} \in \mathcal{K}$.

- $y^{(i)} \leftarrow DPF. Eval(k^{(i)}, x_i)$: Given a key $k^{(i)} \in \mathcal{K}$ and $x_i \in \{0, 1\}^{|a|}$, it outputs $y^{(i)}$ as the (arithmetic/binary) share of $P_{a,b}(x_i)$.

An application of DPF is to implement efficient private information retrieval (PIR). We present an $L$-party DPF-based PIR scheme to retrieve an item $b_j$ in $\mathbb{B}$ by $(b_1, b_2, \ldots, b_m)$ as follows. The client creates $L$ keys $(k^{(1)}, \ldots, k^{(L)}) \leftarrow DPF. Gen(\lambda, 1, 1)$ for $L$ parties, where $k^{(1)}, \ldots, k^{(L)} \in \{0, 1\}^n (n = O(\lambda \log m) \text{ for } L = 2, \text{ or } n = O(\lambda \sqrt{m}) \text{ for } L \geq 3)$ and $k^{(i)}$ is sent to party $P_{i}$ ($t \in [L]$). Each party $P_{i}$ returns $r^{(i)} \leftarrow \sum_{i=1}^{L} DPF. Eval(k^{(i)}, i) \times b_i$, and the client reconstructs the retrieved item $b_j \leftarrow \sum_{i=1}^{L} r^{(i)}$.

### 2.2 Key-Homomorphic PRF (KH-PRF)

KH-PRF [9] enables distributed evaluation of a secure PRF function $F^* : \mathcal{K} \times X \rightarrow Y$ such that $(\mathcal{K}, * )$ and $(Y, \circ)$ are both groups and for every $k_1, k_2 \in \mathcal{K}$, $F^*(k_1 + k_2, x) = F^*(k_1, x) \circ F^*(k_2, x)$. We define an $L$-party KH-PRF scheme as a tuple of PPT algorithms $KH-PRF = (Gen, Share, Eval)$ as follows.

- $k \leftarrow KH-PRF.Gen(\lambda)$: Given a security parameter $\lambda$, it outputs a secret key $k \in \mathcal{K}$.

- $(k^{(1)}, \ldots, k^{(L)}) \leftarrow KH-PRF.Share(k)$: Given a key $k \in \mathcal{K}$, it outputs $L$ keys $k^{(1)}, \ldots, k^{(L)} \in \mathcal{K}$ such that $k^{(1)} + \cdots + k^{(L)} = k$.

- $y \leftarrow KH-PRF.Eval(k, s)$: Given a key $k \in \mathcal{K}$ and a seed $s \in \{0, 1\}^*$, it outputs the evaluation $y = F^*(k, s) \in Y$.

We present the extension of the 2-party (almost) KH-PRF scheme based on Learning with Rounding (LWR) under Random Oracle Model (ROM) by Boneh et al. [9] into $L$-party setting. Let $\mathcal{H}_2 : \{0, 1\}^* \rightarrow \mathbb{Z}_p^n$ be a hash function modeled as a random oracle. The KH-PRF function $F^* : \mathbb{Z}_q^n \times \{0, 1\}^* \rightarrow \mathbb{Z}_p$ is defined as $F^*(k, s) = \langle \mathcal{H}_2(s, k) \rangle_p$, where $k^{(1)} + \cdots + k^{(L)} = k$. It is almost key homomorphic in the sense that $F^*(k, s) = e + \sum_{i=1}^{L} F^*(k^{(i)}, s) \mod p$ where $e$ is a small error, i.e., $e \in \{0, \ldots, L\}$.

### 2.3 Secret-Shared Shuffle

Secret-shared shuffle permits multiple parties to obliviously shuffle a set and get additive secret shares of the result. We recall a Two-party Secret-shared Shuffle (TSS) [18], which permits two parties $P_1, P_2$ to jointly shuffle a set and obtain two additive shares. TSS scheme in [18] is a tuple of PPT algorithms $TSS = (Gen, ShrTrns, Shfl)$ defined as follows.

- $\pi \leftarrow TSS.Gen(\lambda, n)$: Given a security parameter $\lambda$ and a set size $n$, it outputs a pseudorandom permutation $\pi$ for $n$ elements.

- $(\Delta, a, b) \leftarrow TSS. ShrTrns(\pi, i, \lambda)$: Given a permutation $\pi$ for $n$ elements to $P_1$, and a security parameter $\lambda$ to $P_2$, it outputs $\Delta = b - \pi(a) \in \mathbb{Z}_p^n$ to $P_1$ and $a, b \in \mathbb{Z}_p^n$ to $P_2$.

- $(x', b) \leftarrow TSS. Shfl(\pi, \Delta, a, b)$: Given a permutation $\pi$ and its corresponding $\Delta$ from $P_1$, a set $x$ and masks $a$ from $P_2$, it outputs $x' = \pi(x) + b \in \mathbb{Z}_p^n$ as a masked permutation of $x$ to $P_1$, and the mask value $b$ to $P_2$.

### 3 MODELS

**System Model.** Our system consists of a reader, $n_w$ independent writers, and $L$ servers. WLOG, we identify each writer as a member of $[n_w]$, so that $W = [n_w]$. Each writer $i \in W$ owns a separate collection of $N$ documents and would like to share it with the reader. We identify each document in the database as a member of $[N]$. We consider the reader would like to perform encrypted keyword search over all document collections of a writer subset $W' \subseteq W$. On the other hand, the writer can revoke the permission of the reader if needed. The reader and writers are independent parties and they do not have to communicate directly with each other. Our scheme is a Multi-User SE scheme (MUSES) defined as follows.

**Definition 1 (Multi-User SE).** A MUSES scheme is a tuple of PPT algorithms defined as follows:

- $(pk, sk) \leftarrow RSetup(\lambda)$: Given a security parameter $\lambda$, it outputs a public and private key pair $(pk, sk)$.

- $(\kappa_{w_i}, EIDX_{w_i}, st_i, STK_{w_i}, PTK_{w_i}) \leftarrow WSetup(\lambda, i, pk)$: Given a security parameter $\lambda$, a writer identifier $i$, and the public key $pk$, it outputs a writer key $\kappa_{w_i}$, an encrypted search index $EIDX_{w_i}$, a state $st_i$, a secret token $STK_{w_i}$ encrypted under $\kappa_{w_i}$, and a private token $PTK_{w_i}$ encrypted under $pk$. 

Tung Le, Rouzbeh Behnia, Jorge Guajardo, and Thang Hoang
• \( s \leftarrow \text{SearchToken}(w, \mathcal{W}') \): Given a keyword \( w \) and a subset of writers \( \mathcal{W}' \), it outputs a search token \( s \).

• \( O \leftarrow \text{Search}(s, \mathcal{S}, \{(i, st_i, EIDX_i, PTkn_i)\}_{i \in [n_w]} \) : Given a search token \( s \), the reader’s private key \( \mathcal{S} \), a set of tuples \( \{(i, st_i, EIDX_i, PTkn_i)\}_{i \in [n_w]} \) as the identifier, the state, the encrypted search index and the private token, respectively, of writers \( i \in [n_w] \), it outputs the search result \( O \).

• \( u \leftarrow \text{UpdateToken}(w, u, i, \omega, \mathcal{S}^{K_{SE}}, \mathcal{S}^{K_{DE}}, \mathcal{S}^{K_{ST}}) \in [n_w] \) : Given a set of keywords \( w \) for a document \( u \), a writer identifier \( i \), a writer key \( \mathcal{S}^{K_{SE}} \), states \( \mathcal{S}^{K_{DE}} \), and secret tokens \( \mathcal{S}^{K_{ST}} \) of writers \( i \in [n_w] \), it outputs an update token \( u \).

• \( (EIDX'_i, st'_i) \leftarrow \text{Update}(u, \{EIDX_i, st_i\}_{i \in [n_w]} \) : Given an update token \( u \), encrypted indices \( EIDX_i \) and states \( st_i \) of writers \( i \in [n_w] \), it outputs updated search index \( EIDX'_i \) and updated state \( st'_i \).

• \( (EIDX'_i, STkn'_i) \leftarrow \text{RevokePerm}(i, \mathcal{S}^{K_{SE}}, \{EIDX_i, st_i, STkn_i\}_{i \in [n_w]} \) : Given a writer identifier \( i \) and the corresponding writer’s secret key \( \mathcal{S}^{K_{SE}} \), encrypted search indices \( EIDX_i \), states \( st_i \), and secret tokens \( STkn_i \) of writers \( i \in [n_w] \), it outputs updated search index \( EIDX'_i \) and updated secret token \( STkn'_i \).

Definition 2 (Correctness of MUSES). For all \( \lambda \), (sk, pk) \leftarrow \text{RSetup}(1^\lambda) \), \((\mathcal{S}^{K_{SE}}, \mathcal{S}^{K_{DE}}, \mathcal{S}^{K_{ST}}) \leftarrow \text{WSetup}(1^\lambda, \mathcal{I}'_{\mathcal{I}} ) \), \( i \in \mathcal{W} \), and all sequences of \text{Search}, \text{Update} operations over \( \{EIDX_i\}_{i \in [n_w]} \) using tokens generated respectively from \text{SearchToken}(w, \mathcal{W}') \) and \text{UpdateToken}(w, u, i, \mathcal{S}^{K_{SE}}, \mathcal{S}^{K_{DE}}, \mathcal{S}^{K_{ST}}) \in [n_w] \) as well as \text{RevokePerm} operations over \( i \in \mathcal{W} \) \( \subseteq \mathcal{W} \), \text{Search} returns the correct results w.r.t the inputs \( (w, u, i) \) of \text{UpdateToken} when \( i \in \mathcal{W}' \setminus \mathcal{W} \), except with negligible probability in \( \lambda \).

Threat and Security Models. We assume the adversary can corrupt up to \( L - 1 \) out of the \( L \) servers, and an arbitrary number of writers. We assume the adversary is semi-honest, meaning that it is curious about the query of other honest writers/reader but follows the protocols faithfully. We concentrate on the security of the search index and its related operations. Let \( EIDX_i = (I_{1,i}, I_{2,i}, \ldots, I_{\mathcal{I}_i,i}) \) be an encrypted search index of writer \( i \), where \( I_{ij,i} \) contains information about keywords of the \( j \)-th document. Let \( \Omega \) be a sequence of operations on \( EIDX_i \). We denote \( t \) as the timestamp when an operation happens, \( \Omega_t \) records \( s_t \) for a search on keyword \( w \), and \( (t, u, w) \) for an update of document \( u \) with its new keywords \( w \) on \( EIDX_i \).

Definition 3 (Search Pattern [23, 73]). The search pattern \( \mathcal{S} \) indicates the frequency of search operations on some keywords, i.e., \( sp(w) = \{t : (t, w) \in \Omega_t\} \).

Definition 4 (Result Pattern). The result pattern \( rp(w) \) reveals what documents match the queried keyword \( w \), i.e., \( rp(w) = \{u_1, \ldots, u_{N'} : w \in I_{ju} \land i \in [N'] \subseteq [N]\} \).

Definition 5 (Search Volume). The search volume \( sv(w) \) indicates the number of documents matching the queried keyword, i.e., \( sv(w) = N' \) s.t. \( w \in I_{ju} \land u \in [N'] \subseteq [N] \).

Definition 6 (Document Update Pattern). The update pattern \( up(w) \) records the update frequency on documents, i.e., \( up(w) = \{t : (t, u, w) \in \Omega_t\} \).

The adversary can issue a sequence of queries to the MUSES oracle for any of the following: (i) writer corruption query, which returns the secret key of a specific writer; (ii) search query, which returns the search token of the queried keyword under a writer subset; (iii) update query, which returns the update token for a document of a specific writer; and (iv) revoke query, which returns the updated search index and secret tokens of a specific writer. The adversary can issue queries based on prior outcomes. To define security, we define the notion of history that captures a sequence of queries issued by the adversary into MUSES as follows.

Definition 7 (History). A history of MUSES is a sequence of queries \( \mathcal{H} = \{\text{Hist}_t\} \), where sequence number \( t \) denotes the timestamp when the query happens and \( \text{Hist}_t \in \{(\text{CorruptWriter}, i), (\text{Search}, w, \mathcal{W}'), (\text{Update}, i, u, w), (\text{Revoke}, i)\} \).

We introduce a leakage function family \( \mathcal{L}_H = \{\mathcal{L}_H^{\text{Setup}}, \mathcal{L}_H^{\text{Search}}, \mathcal{L}_H^{\text{Update}}, \mathcal{L}_H^{\text{Revoke}}, \mathcal{L}_H^{\text{CorruptWriter}}\} \) to cover the information of the history \( \mathcal{H} \) leaked during setup, search, update, permission revocation, and writer corruption, respectively. When an oracle is queried for the \( t \)-th operation, any function in \( \mathcal{L}_H \) is initialized with \( \mathcal{H} \), which is the history consisting of the previous \( (t - 1) \) operations and the \( t \)-th operation as the function input. It captures the leakage incurred by the current operation and all historical operations. Before any query (i.e., \( \mathcal{H} = \emptyset \)), \( \mathcal{L}_H = \mathcal{L}_H^{\emptyset} \). We also implicitly assume that MUSES histories are non-singular as defined in [46, 73].

Definition 8 (Adaptive Security of MUSES). For all PPT adversary \( \mathcal{A} \) and the game \( \text{IND}^{b}_{\text{MUSES}, \mathcal{A}, \mathcal{L}}(1^\lambda) \) in Figure 1, MUSES is
\( L_{MUSES} \)-adaptively-secure if:

\[
\Pr[\text{IND}_{MUSES, \Delta}(1^t) = 1] - \Pr[\text{IND}_{MUSES, \Delta}(1^t) = 1] \leq \text{negl}(t).
\]

Corruption leakage. To capture corruption leakage, we introduce the following function:

- UpdateBy(i): This function lists all updates by the writer i in the history. Formally, UpdateBy(i) = \{Hist1 : Hist2 = \{Update, i, u, w\} ∈ H\}.

Definition 9 (Forward Privacy of MUSES). An \( L_{MUSES} \)-adaptively-secure MUSES is forward-private if the update leakage \( L_{MUSES}^{\text{Update}}(i, u, w) \) of any update (i, u, w) by writer i s.t. (CorruptWriter, i) ∉ H can be written as \( L'(i, u) \) or \( L''(i), \) where \( L', L'' \) are stateless functions.

Definition 10 (Backward Privacy of MUSES). An \( L_{MUSES} \)-adaptively-secure MUSES is backward-privacy if the search and update leakage functions \( L_{MUSES}^{\text{Search}}, L_{MUSES}^{\text{Update}} \) can be written as \( L_{MUSES}^{\text{Update}}(i, u, w) = L'(i, u) = L''(i), \) where \( L', L'' \) are stateless functions.

Document retrieval/update leakage. In this paper, we focus only on the security of the search index component in encrypted search. Retrieving/updating actual documents from the writer’s database is out-of-scope. Sealing leakage from actual document retrieval/update is an independent study and some oblivious techniques (e.g., PIR [57, 58], ORAM [19–21]) can be applied orthogonally to our scheme to achieve system-wide end-to-end security.

4 OUR PROPOSED SCHEME

We introduce our new building block for L-party oblivious shuffle. We then present our proposed MUSES scheme.

4.1 New Building Block: L-party Shuffle

We present a generic L-party shuffle protocol extended from the two-party shuffle in [18] that permits L parties to randomly shuffle their L-additive shares of a data vector together in a way that one party learns the output of the permuted data while each of the other parties learns a permutation of a permutation composition. Note that this input/output of our scheme is opposite to that of the two-party protocol in [18], wherein one party has a plaintext data vector \( x \) as input, and the output is the shares of permuted data, i.e., \( P \) has \( \pi(x) + b \) and \( P_2 \) has \( b \).

Recall that in the two-party shuffle in [18], there is a preprocessing phase, where one party \( P_2 \) generates two random masking vectors \( a \) and \( b \) and interacts with the other party \( P \) owning a random permutation \( \pi \) in a way that \( P \) learns a translation function \( \Delta \) demonstrating the relation of \( b \) and the permutation of \( a \) with respect to \( \pi \), i.e., \( \Delta = b - \pi(a) \).

To extend into L-party shuffle setting, our high-level idea for precomputation is to have each party \( P_i \) with \( 1 \leq i < L \) obtain the translation function \( \Delta_i \) w.r.t its chosen random permutation \( \pi_i \), each party \( P_i \) with \( 2 \leq i \leq L \) keeps the share of the mask vector \( a_i \), and the last party \( P_L \) keeps a mask vector \( b_L \) such that:

\[
\pi_{i-1} \ldots (\pi_i(\Delta_i) + a_i) \ldots + \Delta_{L-1} = \pi_{i-1} \ldots (\pi_i(\sum_{j=0}^{i-1} a_j)) \ldots + b_L.
\]

Specifically, each party \( P_i, (1 \leq i < L) \) first generates independent permutation \( \pi_i \) then interacts with each other (as well as party \( P_L \)) to compute the translation function \( \Delta_i \) such that:

\[
\Delta_i = \pi_{i+1} - \pi_i(x_i)
\]

where \( a_i \) and \( b_i \) are the shares of random masks known by party \( P_i \), while its corresponding translation function \( \Delta_i^{(i)} = b_i^{(i)} - \pi_i(a_i^{(i)}) \) is known by party \( P_i \). To achieve this, we execute the two-party share translation protocol between \( P_i \) and \( P_{i+1} \), for \( j = i + 1, \ldots, L \), to generate \( \Delta_i^{(i)} = b_i^{(i)} - \pi_i(a_i^{(i)}) \) for \( P_i \), and \( a_i^{(i)}, b_i^{(i)} \) for \( P_{i+1} \) (Figure 2, LSS.Shift algorithm, lines 1–4). Then, for each party \( P \) for \( 2 \leq i < L \), each party \( P_j \) for \( j = i + 1, \ldots, L \), sends \( \Delta_j^{(i)} = b_j^{(i-1)} - a_j^{(i)} \) to \( P_i \) (lines 5–7). From \( \Delta_i^{(i)} \) and \( \Delta_j^{(j)} \), each party \( P_i, (1 \leq i < L) \) can obtain \( \Delta_i \) as shown in (1) (lines 8–11).

Upon completing the above precomputation, the parties can shuffle their secret-shared data as follows. Let \( d(i) \) be the share of data \( d \) held by \( P_i \). First, each party \( P_i (i \geq 2) \) sends its mask value \( z_i \leftarrow d(i) + a_i^{(1)} \) to \( P_1 \) (Figure 2, LSS.Shift algorithm, line 1). Then \( P_1 \) computes \( a_1 \leftarrow d(1) + \sum_{i=2}^{L} z_i \), and forwards \( a_2 \leftarrow \pi_1(a_1) + \Delta_1 = \pi_i(d) + \sum_{i=2}^{L} b_i^{(1)} \) to \( P_2 \) (line 2), who in turn computes \( a_3 \leftarrow \pi_2(a_2) + \Delta_2 = \pi_2(d) + \sum_{i=3}^{L} b_i^{(2)} \) and forwards it to \( P_3 \), and so on (lines 3–4). The above process continues until the final party \( P_L \) receives \( z_L = \pi_{L-1}(\ldots (\pi_1(d) \ldots )} + b_L^{(L-1)} \) from \( P_1 \). Finally, as \( P_L \) holds the mask \( b_L^{(L-1)} \), it can compute \( r \leftarrow a_L - b_L^{(L-1)} = \pi_1(\ldots (\pi_{L-1}(d))) \), which is the permutation of \( d \) (line 5).

As each party \( P_i \) holds an independent permutation, and the final vector (when sent to \( P_2 \)) is shuffled with \( L-1 \) independent permutations and masked with a random vector, it is easy to see our L-party shuffle achieves dishonest majority security in the sense that the collusion of \( L-1 \) parties does not learn the information of the whole permutation sequence applied on the data vector.

Figure 2: Our L-party Oblivious Secret-Shares Shuffle (LSS).

\[
\Delta_i = x_{i+1} - \pi_i(x_i)
\]
It contains information for the reader to execute when adding vectors, which can be interpreted as a binary matrix of size $K \times 1$. For each column $i$, the writer generates a KH-PRF key $\kappa_{wi}$ for such a decryption process. We first present necessary data structures followed by detailed operations.

### Data Structures

*Search Index.* Similar to [73], each writer $i \in \mathcal{W}$ has an independent search index representing keyword-document relationship in her document collection. We make use of BF to create an efficient search index for each writer $i \in [n]$. Suppose that there are $N$ documents, the writer extracts a set of unique keywords $\mathcal{V}_d$ for each document $u \in [N]$ and computes its BF representation as $w_u \leftarrow BF.Gen(\mathcal{V}_d) \in \{0, 1\}^m$. The search index contains $N$ BF vectors, which can be interpreted as a binary matrix of size $N \times m$ as $IDX = [w_1, w_2, \ldots, w_N] \in \{0, 1\}^{N \times m}$. By this representation, searching a keyword incurs checking its membership with BF by reading $K$ columns in $IDX_i$, where $K$ is the BF parameter. On the other hand, updating a document $u$ incurs creating a new BF representation of the updated keywords in $u$ and writing it to the corresponding row $IDX_i[u, \ast]$.

For confidentiality, $IDX_i$ needs to be encrypted. For reader efficiency, the writer needs to encrypt her index in a way that the reader can delegate the decryption task securely to the servers during search. MUSES makes use of KH-PRF for such a decryption delegation, where the writers’ indices are encrypted with almost KH-PRF function denoted as $F^*$.

First, the writer $i$ interprets the search index as a matrix $IDX_i = [d_1, d_2, \ldots, d_m] \in \{0, 1\}^{N \times m}$, where $d_u \in \{0, 1\}^N \forall u \in [m]$. For each column $m \in [m]$, the writer generates a KH-PRF key as $r_m \leftarrow KH-PRF.Gen(1^N)$, where $r_m \in \{0, 1\}^N$. Our work emphasizes on proper handling “almost” attribute of KH-PRF to achieve user efficiency. Notice that since $F^*$ is an almost KH-PRF, there exists a small error $\varepsilon$ as shown in [2.2] during the KH-PRF evaluation. Thus, it is necessary to reserve several bits for the error in the column data before being encrypted with KH-PRF so that such error can be “ruled out” after KH-PRF decryption to obtain the original data. Moreover, since the server will also perform secure addition of $K$ columns after KH-PRF evaluations for BF membership check, we need to reserve enough space for the aggregated error. Let $z = \lceil \log_2(\varepsilon \cdot K) \rceil$ be the number of bits for the aggregated error when adding $K$ columns encrypted by KH-PRF together. For each $u \in [N]$, the writer encrypts the element $d_u[u]$ as

$$\hat{d}_u[u] \leftarrow (d_u[u] \ll z) + F^*(r_u, u \parallel st_{i,u}) \pmod{p}$$

where $st_{i,u}$ is the update state of document $u$ (initialized with 0). The final encrypted index is $EIDX_i = [\hat{d}_1, \hat{d}_2, \ldots, \hat{d}_m] \in \mathbb{Z}_p^{N \times m}$, where $\hat{d}_u \in \mathbb{Z}_p^N \forall u \in [m]$.

#### Auxiliary Information.

In MUSES, each writer $i$’s encrypted index $EIDX_i$ is associated with three auxiliary components as follows.

- **Private token $PTkn_i$**: It contains information for the reader to search on $EIDX_i$, which are the KH-PRF keys $r_u$ that the writer $i$ uses to encrypt $IDX_i$ as discussed above. Let $(pk, sk) \leftarrow KH.Gen(1^N)$ be the public-key and private-key pair of the writer. The private token is $PTkn_i = (PTkn_{i,1}, \ldots, PTkn_{i,m})$, where $PTkn_{i,o} \leftarrow KH.PrivEnc(pk, r_o)$ for $o \in [m]$.

- **Secret token STkn_i**: It contains information for the writer $i$ to update her index, which is also the KH-PRF keys $r_u$ for $u \in [N]$, but encrypted with the writer’s secret key. Let $\kappa_{wi} \leftarrow KH.Gen(1^N)$ be the secret key of the writer $i$. The secret token is $STkn_i = (STkn_{i,1}, \ldots, STkn_{i,m})$, where $STkn_{i,o} \leftarrow KH.PrivEnc(\kappa_{wi}, r_o)$ for $o \in [m]$. The secret token is stored at the servers to achieve stateless writers.

- **Update state $st_i$**: It contains information concatenated with document identifier as seed value for KH-PRF evaluation. Specifically, the update state is $st_i = (st_{i,1}, \ldots, st_{i,N})$, where $st_{i,u}$ is the counter value (initialized with 0) for document $u$ that is incremented after each update operation on that document happens.

#### Setup protocol. Figure 3 presents the setup algorithms for the reader and the writers in MUSES with the following highlights.

- **Reader**: The reader executes $RSetup$ algorithm to generate a public and private key pair $(pk, sk)$. The reader keeps $sk$ private, and distributes $pk$ to all writers to setup necessary components.

- **Writer**: Each writer $i$ executes $WSetup$ algorithm on reader’s public key $pk$ to generate four components including the encrypted search index $EIDX_i$, a secret token $STkn_i$, a private token $PTkn_i$, and the update state $st_i$ as discussed above. Specifically, the writer first generates a secret key $\kappa_{wi}$ (line 1), and $K$-PRF keys $\{r_{i,1}, \ldots, r_{i,m}\}$ to encrypt $m$ columns of the search index (line 3). These $K$-PRF column keys are then encrypted under the writer’s secret key $\kappa_{wi}$ as the secret token $STkn_i$, and also encrypted under the reader’s public key $pk$ as private token $PTkn_i$ (line 5). The writer initializes counter values as 0 stored in $st_i$ (line 6). Finally, the writer encrypts an empty search index cell-by-cell by evaluating the almost KH-PRF function $F^*$ with KH-PRF column keys $r_{i,o}$ and the seeds formed by the update counters and row indices (line 7). To this end, the writer sends the encrypted index (EIDX) and auxiliary components ($st_i$, $STkn_i$, $PTkn_i$) to $L$ servers, while keeping $\kappa_{wi}$ private.

#### Key search protocol.

We present the search protocol of MUSES in Figure 4. Specifically, to search for a keyword $w$ on the index of a writer subset $\mathcal{W}'$, the reader first executes SearchToken algorithm to compute its BF representation as column indices $(\ell_1, \ldots, \ell_k)$, then creates corresponding DPF keys $\{q^{(I)}_k\}_{k \in [K]}$ (lines 1–4), and sends them to servers $P_1, \ldots, P_L$ as search tokens (lines 5–6). Where server $P_\ell$ receives the corresponding token $s_\ell = (\mathcal{W}', \{q^{(I)}_k\}_{k \in [K]})$. Next, upon receiving the search token, the
servers execute Search algorithm as follows. Each server $\mathcal{P}_\ell$ performs DPF evaluation on its received DPF keys with search indices $\text{EIDX}_i$ to privately retrieve the additive shares of requested columns $\text{EIDX}_i[x, k_1]$ as $(\tilde{d}_1^{(i)}, \ldots, \tilde{d}_L^{(i)})$, and with private tokens $\text{PTkn}_{I_k}$ to retrieve the additive shares of requested private tokens $(\text{PTkn}_{I_{0}}^{(L_1)}, \ldots, \text{PTkn}_{I_{0}}^{(L_k)})$ as $(\text{PTkn}_{I_{0}}^{(L_1)}, \ldots, \text{PTkn}_{I_{0}}^{(L_k)})$, for $i \in [K]$, and $i \in W'$.

$$
\tilde{d}_l^{(i)} [u] \leftarrow \tilde{d}_l^{(i)} [u] \leftarrow F_\lambda^{(i)} [u] \mid \text{st}_{i, u} \text{ (mod $p$)}
$$

When using BF, the servers have to merge retrieved columns to obtain the final search output. Otherwise, unwrapping each column (even when being shuffled) during search process may reveal the patterns of retrieved columns, which leads to search pattern leakage. Therefore, each server $\mathcal{P}_\ell$ computes its share of the aggregated column as $d^{(i)}_l \leftarrow \sum_{k=1}^{K} d^{(i)}_l \in \mathbb{Z}_p^N$. After this step, the servers can broadcast $d^{(i)}_l$ to reconstruct $d^{(i)}_l \sum_{l=1}^{L} d^{(i)}_l \in \mathbb{Z}_p^N$. However, it also permits the servers to obtain document identifiers matching the search query, which is known as result pattern leakage. We show how to further hide result pattern leakage on the search index in our scheme as follows. We utilize $L$-party secret-shared shuffle (LSS) (Figure 2) to securely permute the document identifiers in the search output. Our high-level idea is that after shuffling, each server $\ell \in 1, \ldots, L - 1$ holds an independent permutation, and the other server $L$ holds shuffled opened (unmasked) output which is permuted by a permutation composition of servers $1, \ldots, L - 1$. As a result, only the reader who receives all permutations and the shuffled opened output can recover the actual data. So after obtaining the share of the aggregated column $d^{(i)}_l$ above, $L$ servers $\mathcal{P}_1, \ldots, \mathcal{P}_L$ interact to perform oblivious shuffle (line 19) such that at the end of this process, each server $\mathcal{P}_\ell$, for $\ell \in [L - 1]$, holds an independent permutation $\pi_\ell$, and server $\mathcal{P}_L$ holds $d^{(i)}_l$, which is shuffled opened (unmasked) $d^i$. From (2) and (3), we can see that $d^{i}_l[u]$ contains an aggregated error as

$$
\tilde{d}_l^{(i)} [u] \leftarrow \sum_{i=1}^{K} d^{(i)}_l [u] + \sum_{i=1}^{K} e_{i, L, \pi_\ell} (u) \text{ (mod $p$)}
$$

where $\pi_\ell = \pi_{\ell-1} \cdot \pi_{\ell-2} \cdots \pi_1$ is the permutation composition constituted by $L - 1$ permutations $\pi_1, \ldots, \pi_{L - 1}$, and $e_{i, L, \pi_\ell} (u)$ is the error during the KH-PRF evaluation of each $d^{(i)}_l [u]$ in (2). Therefore, the server $L$ can perform $d^{i}_l[u] \leftarrow d^{i}_l[u] \mid \Rightarrow z$ for each $u \in [N]$ to remove $z$ bits of the small aggregated errors in (4), thereby obtaining the final search result as $\mathcal{O}_I = \{ u \in [N] : d^{i}_l[u] = K \}$ (lines 20-22).

Finally, the server $\mathcal{P}_\ell$, for $\ell \in 1, \ldots, L - 1$ sends $\pi_\ell^2$, while $\mathcal{P}_L$ sends $\mathcal{O}_I$ to the reader, for $i \in W'$, and the reader can recover the actual search output by reversing each permutation sequentially (lines 23-25).

4.2.4 Access revocation protocol. MUSES permits a writer to revoke access permission of the reader on her search index. The idea is to re-encrypt the writer’s index with refreshed (column) KH-PRF keys unknown to the reader. Figure 5 presents our revocation protocol, where the re-encryption operation is delegated securely to the servers for writer efficiency. Its high-level idea is as follows.

To re-encrypt EIDX, the writer $i$ first retrieves and decrpts STkn to obtain current column keys $r_{t, o}$ (line 2). Also, the writer generates new secret column keys $t_{l, o}^{(i)}$ and encrypts them as the updated secret token STkn$^{(i)}_{l, o}$ (line 3). The writer creates secret-shares of these keys as $(t_{1, o}^{(i)}, \ldots, t_{L, o}^{(i)})$, $(r_{t, o}^{(i)}, \ldots, r_{t, o}^{(i)})$ (lines 4-5), and sends $\{ \text{STkn}^{(i)}_{l, o}, r_{t, o}^{(i)}, t_{l, o}^{(i)} \}$ to server $\mathcal{P}_\ell$, for $\ell \in [m]$ and $\ell \in [L]$. This can be done efficiently by sending a PRP seed.
RevokePerm(i, κ_wj, {EIDX_k, st_i, STkn_i}) \in \{[w]\}:

Writer i → Server P_i: i
Server P_i → Writer i: secret tokens STkn_i
Writer i:
1. for v = 1 to m do
   2. τ_v = E.Dec(κ_wj, STkn_jv, τ_v)
   3. STkn_i = E.Enc(κ_wj, τ_v, STkn_i)
   4. (τ_v, {τ_v}) = KH-PRF.share(τ_v)
   5. (τ_v, {τ_v}) = KH-PRF.share(τ_v)
   6. STkn_i = STkn_i, STkn_j

Writer i → Server P_i': (STkn', i', τ', STkn'), for v ∈ [m] and f ∈ [L]
Server P_i:
7. STkn' = STkn_i
8. for u = 1 to N do
   9. for v = 1 to m do
      10. x_u,v \leftarrow \mathbb{Z}_p, M^{v}(u,v) \leftarrow x_u,v \leftarrow z
      11. T^{1}(u,v) \leftarrow M^{v}(u,v) \leftarrow F^{1}(τ', u) \leftarrow \cdot \cdot \cdot x_u,v + \max_e (\text{mod } p)
      12. T^{1}(u,v) \leftarrow M^{v}(u,v) \leftarrow F^{1}(τ', u) \leftarrow \cdot \cdot \cdot x_u,v + \max_e (\text{mod } p)

∀ j ∈ [L] and j ≠ i: Server P_i → Server P_i': T^{1}(u,v), T^{1}(u,v) for v ∈ [L]
Server P_i:
13. for u = 1 to N do
   14. for v = 1 to m do
      15. T^{1}(u,v) \leftarrow EIDX(u,v) + \sum_{z_{i,v}} T^{1}(u,v)
      16. T^{1}(u,v) \leftarrow \leftarrow T^{1}(u,v) \leftarrow \cdot \cdot \cdot x_u,v \leftarrow z
      17. EIDX'(u,v) \leftarrow \leftarrow T^{1}(u,v) + \sum_{z_{i,v}} T^{1}(u,v) + \max_e (\text{mod } p)
      18. return (EIDX', STkn_i')

Figure 5: Our MUSES permission revocation.

Each server P_i replaces STkn_i by STkn_i' (line 7), computes T^{1}(u,v) and T^{1}(u,v), where T^{1}(u,v) is the masked component to remove the shared encryption computed by the secret-shared key r^{1}_L, and T^{1}(u,v) is the component to unmask the value M^{v}(u,v) added by T^{1}(u,v) and add the shared encryption computed by the new secret-shared key r^{1}_L (lines 8–12). The server P_i then distributes T^{1}(u,v) and T^{1}(u,v) to the other servers. Next, each server P_i obtains the masked value with error denoted as T' (lines 13–15), where T' \leftarrow (T^{1}(u,v) = \sum_{z_{i,v}} M^{v}(u,v) \leftarrow z) + \sum_{z_{i,v}} M^{v}(u,v) + \epsilon_{u,v}. The random mask M^{v}(u,v) generated by each server P_i is to hide the plaintext data (IDX(u,v) \leftarrow z) when the servers remove the encryption by adding EIDX with T' \leftarrow T^{1}(u,v) \leftarrow T^{1}(u,v) to obtain T'. By clearing z LSBs of each value T^{1}(u,v) to 0, the error value part \epsilon_{u,v} is removed, and the servers now hold the masked plaintext value T^{1}(u,v) (line 16). Finally, to retrieve the final EIDX_i encrypted by the new secret-key, each server P_i computes EIDX_i(u,v) based on T^{1}(u,v) and \sum_{z_{i,v}} T^{1}(u,v). The value \max_e = L is the max error when using LWR-based KH-PRF (line 17). It is necessary for decryption in keyword search later. At the end of this protocol, all servers hold the same updated EIDX_i', which is the search index encrypted with the new columns, as well as updated secret tokens STkn_i'.

4.2.5 Document update protocol. Given an updated document with identifier u and a list of its keywords w, the writer i retrieves the state value st_i, corresponding to the document u, and secret tokens STkn_i = (STkn_i, ..., STkn_i,m) (Figure 6). Then, the writer uses her κ_wj to decrypt STkn_i (lines 1–2) and obtain the secret keys of data columns as r_i,v \leftarrow E.Dec(κ_wj, STkn_i,v, r_i,v), for v ∈ [m]. Next, the writer computes the new BF representation u \in [0, 1]^m of the updated document with input keywords w (lines 3–5). Finally,

UpdateToken(w, u, i, k_{w}, \{st_i, STkn_i\}) \in \{[w]\}:

Writer i → Server P_i: (i, u)
Server P_i → Writer i: state st_i,u, secret tokens STkn_i
Writer i:
1. for v = 1 to m do
   2. r_o = E.Dec(κ_w_j, STkn_i,v)
   3. u \leftarrow (0)^m, u \leftarrow (0)^m
   4. for each w_j ∈ w, j = 1 to K do
      5. c_{ij,k} = H_{i,j}(w, u, c_{ij,k}) ← 1
   6. for u = 1 to m do
      7. u'[v] \leftarrow (u[v] \leftarrow z) + F^{1}(r_o, u) \leftarrow (st_i,u + 1) \leftarrow (mod p)
      8. return u ← (i, u', u')

Update(u, EIDX_i, st_i, st_i,M):
Each server P_i: parse u = (i, u, u')
9. EIDX_i[u,v] ← u'[v], st_i,u ← st_i,u + 1
10. return (EIDX_i, st_i)

Figure 6: Our MUSES document update.

the writer encrypts the updated row u with column keys and the incremented counter value for KH-PRF evaluation as u'[v] ← u[v] + F^{1}(r_o, u) \leftarrow (st_i,u + 1) for each v ∈ [m] (lines 6–7). Finally, the writer sends the update token u = (i, u, u') to the servers to update the search index of the writer accordingly as EIDX_i[u, v] ← u' and st_i,u ← st_i,u + 1 (lines 9–10).

4.3 Analysis

4.3.1 Complexity. We analyze the online3 asymmetric cost of MUSES.

We consider the number of servers L as a small constant and omit it in the following analysis. Let m, K be the BF parameters and N be the number of documents. To search for a keyword w in a writer’s database, the reader creates a query of size O(K \cdot \lambda \cdot \tau) \sim O(\lambda \tau), where \tau = O(\log m) for L = 2, and \tau = O(\sqrt{\lambda}) for L ≥ 3 (as K is a constant BF parameter), and with the corresponding computation complexity O(\lambda \cdot \tau) \sim O(\lambda \tau). Also, for each writer in \{W\}, the reader sends shares of shared BF keys to L servers, which costs O(L \cdot K \cdot n \cdot \log q) \sim O(\lambda) in total (as n, q \sim O(\lambda) are the LWR parameters of KH-PRF). Let n be the bound on the size of the search output. To obtain the search result, the reader receives seed values from P_1, ..., P_{L-1}, and shuffled output from P_L, while re-generates the permutations \sigma_{1}, ..., \sigma_{L-1}, and reverses permutations to obtain the final search output, which incurs O(L \cdot (L - 1) \cdot n) \sim O(\lambda \cdot n) communication and O(L \cdot (L - 1) \cdot N) \sim O(\lambda \cdot N) computation cost. Overall, to search for a keyword on a writer subset \{W\}, the computation complexity at the reader is O(\tau + |W| \cdot N) \sim O(\lambda \cdot N), and the bandwidth cost between the reader and the servers is O(\lambda \cdot \tau + |W| \cdot (\lambda + n)).

To update a document, the writer retrieves and decrypts secret tokens to obtain the secret keys, which is O(m \cdot n \cdot \log q) \sim O(m \cdot \lambda) in bandwidth and O(m) in computation cost. The writer retrieves the counter value of size O(\lambda) of the updated document, creates a new BF representation of size O(m), and re-encrypts it with the secret keys and the incremented counter value. Therefore, the total writer’s bandwidth cost and computation cost per document update is O(m \cdot \lambda) and O(m), respectively.

For keyword search, each server incurs (K \cdot N \cdot m) \sim O(N \cdot m) modulo additions and multiplications. Each server performs O(K \cdot N) \sim O(N) KH-PRF evaluation invocations and the oblivious shuffle incurs O(N) arithmetic additions. The overall server computation cost per search on the writer set \{W\} is O((\lambda \cdot |W| - (N \cdot m + N)) -

3We do not consider the precomputational cost in this analysis.

Conference ’17, July 2017, Washington, DC, USA
$O(|W'| \cdot N \cdot m)$. For document update, the servers replace one row in EIDX and increment a counter value, and thus do not incur computation.

For permission revocation, the writer’s bandwidth and computation cost is similar to the overhead of document update, which is $O(m \cdot \lambda)$ and $O(m)$, respectively. Since the servers are responsible for updating the encrypted search index with new secret keys on behalf of the writer, the computation and inter-server communication costs are $O(N \cdot m)$.

For the storage, the reader and each writer stores a private/secret key of size $O(\lambda)$. For each writer, the servers store an index of size $O(\log p \cdot N \cdot m) = O(N \cdot m)$, a state of size $O(\lambda \cdot N)$, private tokens $PT_k$ and secret tokens $ST_k$, both are of the same size $O(m \cdot n \cdot \log q) \sim O(m \cdot \lambda)$. The total server cost is $O(n_w \cdot N \cdot m + n_w \cdot \lambda + N + n_w \cdot m \cdot \lambda)$, where $n_w$ is the number of writers.

4.3.2 Security Analysis.

Theorem 1. Assuming that the adversary can statically corrupt at most $L = 1$ out of the $L$ servers and some writers, MUSES is $\mathcal{L}_H$-adaptively-secure by Definition 8 with forward privacy by Definition 9, and backward privacy by Definition 10, where $\mathcal{L}_\text{Setup}(1^k) = \{i, N, m\}_{i \in [n_w]}$, $\mathcal{L}_\text{CorruptWriter}(i) = \{\text{UpdateBy}(i)\}$, $\mathcal{L}_\text{Search}(W', \omega) = \{\omega', st(\omega)\}$, $\mathcal{L}_\text{Update}(i, u, w) = \{i, \text{up}(u)\}$, and $\mathcal{L}_\text{Revoke}(i) = \{i\}$, where $W'$ is a writer subset.

We present the proof in Appendix A.

5 EXPERIMENTAL EVALUATION

Implementation. We fully implemented all our proposed techniques in C++ consisting of approximately 2,500 lines of code. We used standard cryptographic libraries, including OpenSSL [1] for IND-CPA encryption and hash functions, 1ibsecp256k1 [76] for public-key encryption in our scheme, and EMP-Toolkit [74] for IKNP OT protocol. We implemented KH-PRF and OS3 from scratch. We used 1ibzeromq [2] to implement network communication between servers and client. Our implementation is available and ready for public release (see the attached artifact).

Hardware and network. We used two EC2 r5n.4xlarge instances each equipped with 8-core Intel Xeon Platinum 8375C CPU @ 2.90 GHz and 128 GB memory as servers. For the reader, we used a laptop with an Intel i7-6820HQ CPU @ 2.7 GHz and 16 GB RAM. The bandwidth between servers is 3 Gbps and the bandwidth between the servers and the client is 20 Mbps with 10 ms RTT.

Dataset. We used the Enron email dataset [3] which includes about 28,229 documents, we let $N = 2^{13}$ be the bound on the number of documents. We choose BF parameters such that $N \times \text{FP rate} < 1$. For $K = 7$, we choose $m = 2000$ to achieve FP rate $\approx 3 \times 10^{-5}$. To evaluate the performance of permission revocation with various database sizes, we run experiments with the number of documents $N$ from $2^{20}$ to $2^{19} \sim 500K$. For the number of documents from $2^{20}$ to $2^{19}$, the corresponding BF parameter $m$ is from 1120 to 3120 (with $K = 7$) to satisfy the condition of low FP rate above.

• FP-HSE [73]: We selected the originally suggested parameters with 96-bit security level, where PRFs and keyed hash functions are instantiated with HMAC-SHA-256, and MNT224 curve for pairings. We measure the latency of FP-HSE in document update in two cases. The worst case is when all keywords of the updated document are new (FP-HSE-new), and the best case is when all keywords have appeared (FP-HSE-exist). Each folder in the dataset is considered as a separate writer.

• DORY [25]: We run experiments with DORY in the semi-honest setting similar to our MUSES and FP-HSE. We configure BF parameters of DORY similar to our scheme because DORY uses DPF-based PIR scheme for oblivious search, and 128-bit keys for IND-CPA encryption, and PRG seeds.

5.1 Overall Results

5.1.1 Keyword search.

Reader’s Bandwidth. Figure 7a shows the bandwidth between the reader and the servers of our MUSES, DORY and FP-HSE. The network overhead in MUSES increases from 0.3 MB to 2.1 MB, corresponding to the cases of 25 to 150 writers mostly due to transmitting KH-PRF key shares. For DORY, it incurs 4.6 MB–27.6 MB network overhead per search operation depending on the writer subset size, which is 12.8x–13.0x larger than the communication cost of MUSES. FP-HSE incurs the lowest bandwidth as the reader only sends a search token of 65 B to the server and receives the results. Although FP-HSE achieves the minimum bandwidth overhead among all schemes, it suffers from many security vulnerabilities and leaks more information than the others.

Keyword Search. Figure 7b illustrates the end-to-end delay in keyword search of our scheme with DORY and FP-HSE for different numbers of writers. The latency of all schemes grows almost linearly to the number of writers. MUSES is about $129.1x \sim 137.2x$.
Figure 8a demonstrates the bandwidth cost when a writer wants to update secret column keys to revoke access downloading and uploading the whole search index.

Just needs to transmit secret-shares of KH-PRF keys, together with corresponding to the cases from 1K to 500K documents as the writer

MUSES: Efficient Multi-User Searchable Encrypted Database Conference’17, July 2017, Washington, DC, USA

Three factors contributing to the delay include reader processing, communication latency, and server processing. Our scheme incurs a low reader processing cost in search, in which it only takes 12.7ms–64.3ms, attributing 0.5%–0.7% to the total delay. By contrast, the server cost in MUSES to search a keyword takes about 1.8s–9.9s, corresponding to 91.3%–91.7% of the total delay. The communication overhead during search in MUSES is about 0.1s–0.9s, which attributes about 7.6%–8.1% to the total delay.

5.1.2 Permission revocation. We evaluate the performance of MUSES when a writer wants to update secret column keys to revoke access permission of the reader on her database, and compare it with other schemes. For DORY and FP-HSE, as these schemes do not offer access revocation function for a user/reader’s database by offloading re-encrypting work to the servers as ours, we measure their latency to re-encrypt a user/reader’s search index on the user/reader side. For FP-HSE, the writer only re-encrypts her underlying SSE with another secret key and ignores updating encrypted search tokens to stop sharing her database with the reader.

Writer’s Bandwidth. Figure 8a demonstrates the bandwidth cost of all schemes in permission revocation. The bandwidth overhead of MUSES grows slightly when increasing BF size (from 1120 to 3120 corresponding to the cases from 1K to 500K documents) as the writer just needs to transmit secret-shares of KH-PRF keys, together with current and updated secret tokens while DORY and FP-HSE requires downloading and uploading the whole search index. MUSES incurs faster than FP-HSE, and 1.6×–1.7× faster than DORY. With 25 writers, MUSES takes approximately 1.8s to accomplish a search, and increases to about 10.9s for 150 writers. The overhead of FP-HSE mainly comes from pairing operations, in which decrypting each encrypted search token needs two pairing operations, while the overhead of our scheme mainly stems from KH-PRF evaluation. By contrast, the overhead of DORY is mostly due to network overhead.

Figure 9: E2E permission revocation delay (log scale on y-axis). Faster than FP-HSE, and 1.6×–1.7× faster than DORY. With 25 writers, MUSES takes approximately 1.8s to accomplish a search, and increases to about 10.9s for 150 writers. The overhead of FP-HSE mainly comes from pairing operations, in which decrypting each encrypted search token needs two pairing operations, while the overhead of our scheme mainly stems from KH-PRF evaluation. By contrast, the overhead of DORY is mostly due to network overhead.

Three factors contributing to the delay include reader processing, communication latency, and server processing. Our scheme incurs a low reader processing cost in search, in which it only takes 12.7ms–64.3ms, attributing 0.5%–0.7% to the total delay. By contrast, the server cost in MUSES to search a keyword takes about 1.8s–9.9s, corresponding to 91.3%–91.7% of the total delay. The communication overhead during search in MUSES is about 0.1s–0.9s, which attributes about 7.6%–8.1% to the total delay.

5.1.2 Permission revocation. We evaluate the performance of MUSES when a writer wants to update secret column keys to revoke access permission of the reader on her database, and compare it with other schemes. For DORY and FP-HSE, as these schemes do not offer access revocation function for a user/reader’s database by offloading re-encrypting work to the servers as ours, we measure their latency to re-encrypt a user/reader’s search index on the user/reader side. For FP-HSE, the writer only re-encrypts her underlying SSE with another secret key and ignores updating encrypted search tokens to stop sharing her database with the reader.

Writer’s Bandwidth. Figure 8a demonstrates the bandwidth cost of all schemes in permission revocation. The bandwidth overhead of MUSES grows slightly when increasing BF size (from 1120 to 3120 corresponding to the cases from 1K to 500K documents) as the writer just needs to transmit secret-shares of KH-PRF keys, together with current and updated secret tokens while DORY and FP-HSE requires downloading and uploading the whole search index. MUSES incurs faster than FP-HSE, and 1.6×–1.7× faster than DORY. With 25 writers, MUSES takes approximately 1.8s to accomplish a search, and increases to about 10.9s for 150 writers. The overhead of FP-HSE mainly comes from pairing operations, in which decrypting each encrypted search token needs two pairing operations, while the overhead of our scheme mainly stems from KH-PRF evaluation. By contrast, the overhead of DORY is mostly due to network overhead.

Three factors contributing to the delay include reader processing, communication latency, and server processing. Our scheme incurs a low reader processing cost in search, in which it only takes 12.7ms–64.3ms, attributing 0.5%–0.7% to the total delay. By contrast, the server cost in MUSES to search a keyword takes about 1.8s–9.9s, corresponding to 91.3%–91.7% of the total delay. The communication overhead during search in MUSES is about 0.1s–0.9s, which attributes about 7.6%–8.1% to the total delay.
bandwidth is higher, in which DORY takes 99.6s–25.8s, and FP-HSE takes 402.6s–67.2s, corresponding to the bandwidths 5 Mbps–30 Mbps. By contrast, MUSES incurs a delay of 31.4s–15.9s to finish a permission revocation, where most communication overhead is for transmitting encrypted KH-PRF keys and their secret-shares to the servers.

Varying numbers of servers. To achieve higher security level, more servers can be added to the system. Figure 11 illustrates the end-to-end delay of keyword search and permission revocation with different numbers of servers increasing from 2 to 6. In MUSES, adding more servers does not significantly increase the online computation work of each server. Instead, it just requires more communication rounds between the servers to forward and open the shuffled search output. In addition, a small amount of extra overhead in search operations is put on the reader to create and send more DPF keys, as well as secret-shares of KH-PRF keys when there are more servers joining the system. In particular, with 2 to 6 servers, the total delay to search for a keyword over the databases of 100 writers are 7.3s, 7.6s, 7.9s, 8.2s and 8.4s, respectively. Similarly, in permission revocation, the additional computation overhead on each server, when adding more counterparts, is insignificant as the servers have to broadcast their masked results to each other. Also, the writer has to create more secret-shares of KH-PRF keys, and upload secret tokens to more servers, which slightly increases communication overhead on the writer. Specifically, for the typical case of 2^16 documents, the end-to-end latencies to finish a permission revocation corresponding to the number of servers growing from 2 to 6 are 17.5s, 19.4s, 21.2s, 23.1s and 24.9s, respectively.

5.1.4 Document update. Figure 12 presents the update delay of MUSES and FP-HSE. For each new keyword, FP-HSE-new needs two pairings to generate a new token, thus its cost is linear to the number of new keywords, while DORY, FP-HSE-exist, and our scheme remain nearly unchanged. MUSES takes about 0.9s to finish a document update, while FP-HSE-new takes 202.6ms-1.2s for the cases increasing from 25 to 150 keywords, and DORY takes about 0.5ms–0.8ms. Most of the overhead in MUSES is due to downloading secret tokens and decrypting them to obtain secret column keys of the search index. The update latency of FP-HSE-exist slightly grows from 43.9ms to 47.1ms as it does not incur pairing operations.

5.1.5 Storage overhead. In MUSES, each writer stores a 128-bit secret key and the reader stores a 256-bit private key. As MUSES utilizes LWR-based KH-PRF in [9], each bit in Zp is encrypted to ciphertext in Zp with p = 210 by 10 bits. As a result, for each writer with an EIDX including 32,768 documents and BF size of 2000, the size of EIDX is ≈ 78.1 MB. Each column of EIDX is encrypted by a separate key, in which each key is of 1 KB. In total, the database size of each writer including EIDX, PTkn, STkn, and st is ≈ 82.5M, which goes up to ≈ 12.4 GB for 150 writers.

6 RELATED WORK

SSE. Song et al. [65] were the first to propose and formalize searchable encryption, followed by a line of SSE schemes proposed that offer secure search over encrypted data plus dynamic update via an encrypted index [10, 11, 16, 17, 27, 37, 39, 45, 46, 50, 53, 67–69]. However, these schemes might be susceptible to many types of leakage-abuse attacks [4, 14, 44, 48, 52, 54, 56, 62, 63] and mainly support single-user setting. The multi-client SE exploited in [49] uses KH-PRF as a mechanism to share a secret key between the trusted and “helping” users to protect the confidentiality of search tokens, as well as to remove the interaction with the data owner in search operations. However, the proposed system might be impractical as each search operation requires the presence of certain “helping” users. In addition, the “almost” property of current KH-PRF schemes has been ignored in that design, which might require proper and subtle treatment in many circumstances.

PEKS. PEKS schemes such as [6, 8, 30, 77] can support multi-writer setting, but they do not adapt to forward privacy, which might lead to injection attacks [79]. Although hybrid-based architecture (e.g., FP-HSE [73]) can ensure forward privacy, it requires each writer to be stateful and present to rebuild encrypted tokens periodically. In addition, most PEKS systems are vulnerable to KGA. PEKS in 2-server setting [51] can protect the privacy of the given trapdoor to prevent KGA, but it still incurs a large number of pairing operations in search (which runs linear to the number of keyword-document pairs). Single-key SSEs [25, 32] require that all users are trusted or assume a secure deployment environment with authenticated mechanisms. Multi-key SSE [49, 75] can provide decentralization between users but does not prevent pattern leakages.

Oblivious platforms. Some oblivious storage platforms employ ORAM and/or PIR primitives to hide access patterns during data operations (e.g., data sharing/access [19–21, 25, 57, 58], search [28, 32, 36, 42, 61]). However, these schemes incur a large communication overhead, which costs O(N) with N is the number of documents in the database. Differential Privacy-based technique [64] can obfuscate search access patterns, but it incurs high computation and latency.

Hardware-assisted SE. Trusted hardware (e.g., Intel SGX [22]) has been used to build practical oblivious platforms with diverse functionalities [24, 31, 34, 35, 40, 41, 60, 70]. These platforms require a strong security assumption on the hardware (e.g., isolation, tamper-free, side-channel resistance).
ACKNOWLEDGMENT

Thang Hoang is supported by an unrestricted gift from Robert Bosch, and the Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber R&D, innovation, and work force development. For more information about CCI, visit www.cyberinitiative.org. Rouzbeh Behnia is supported by the USF Sarasota-Manatee campus Office of Research through the Interdisciplinary Research grant program.

REFERENCES

A SECURITY PROOF OF MUSES (THEOREM 1)

We derive a \((t + 1)\)-hybrid sequence starting from \(\text{Hybrid}_0 = \text{IND}_0\text{MUSES}\) and the last hybrid \(\text{Hybrid}_{t+1}\) is exactly \(\text{IND}_{t+1}\text{MUSES}\). For \(t \in \{0, \ldots, t-1\}\), the only difference between \(\text{Hybrid}_0\) and \(\text{Hybrid}_{t+1}\) is that the oracle responds to the \((t + 1)\)-th query in \(\text{Hybrid}_{t+1}\) with input \(b = 0\) instead of \(b = 1\).

We prove that \(\mathcal{A}\) cannot distinguish \(\text{IND}_{t+1}\text{MUSES}\) from \(\text{IND}_t\text{MUSES}\) with non-negligible probabilistic by showing that each hybrid (except the first) is indistinguishable from its previous.

For \(t \in \{0, \ldots, t-1\}\), \(\text{Hist}_{t+1}\) can fall into four cases:

1. **CorruptWriterO** on \((i_0, i_1)\): It will only be answered when identifier \(i = i_0 = i_1\). As the update token provided by the corrupted writer is revealed, it requires that the update tuples provided by the writer \(i\) (i.e., UpdateBy\(i\)) are the same for either \(b = 0\) or \(b = 1\). We have \(\text{Hybrid}_t = \text{Hybrid}_{t+1}\) as the views of \(\mathcal{A}\) are identical.

2. **SearchO** on \((W_{t+1}, W_{t+1} \text{key}\{0,1\})\): As the target writers of any search leaks, it will only be answered when \(W_t = W_{t+1} = W_f\). Because at most \(L - 1\) out of \(L\) servers are corrupted, \(\mathcal{A}\) cannot learn the search output unless there are corrupted writers. If servers \(P_1, \ldots, P_{L-1}\) are corrupted, permutations \(\pi_1, \ldots, \pi_{L-1}\) are the only leaked information. Otherwise, if server \(P_t\) is included in the corrupted servers together with \(L - 2\) remaining servers who hold permutations, the search volume of \(w_k\) (i.e., \(sw(w_k)\)) is revealed,
and the oracle only answers when \(sv(w_0) = sv(w_1)\) in this case. The adversary still lacks a permutation of the honest (uncorrupted) server to completely recover the search output.

Depending on whether any writer in \(W\) has been corrupted, there are two following cases:

- If \(\exists \text{ Writer } i : i \in \mathcal{W} \land (\text{CorruptWriter}, i) \in \mathcal{H}, \mathcal{A}\) will have knowledge of the correct search output of the corresponding writer, and the oracle only answers when \(w = w_0 = w_1\) in this case.
- Otherwise, because the adversary only gets access to at most \(L - 1\) out of \(L\) servers, the adversary cannot tell the difference between the indices corresponding to the search queries, as well as whether the search output is correct or not.

The indistinguishability between \(\text{Hybrid}_L\) and \(\text{Hybrid}_{L+1}\) is guaranteed by the computationally indistinguishable security of \(\text{DPF}\)-based \(\text{PIR}\), \(\text{KH-PRF}\), and \(\mathcal{L}\) schemes.

3. **UpdateO** on \((\{i_k, u_k, w_k\}_{k \in \{0,1\}})\): The oracle answers the queries when \(i = i_0 = i_1\) and \(u = u_0 = u_1\) (i.e., \(\text{up}(u)\)), as the writer identifier and the position of the updated row (i.e., \(\text{EIFD}[u, *]\)) as well as the position of the updated state (i.e., \(\text{st}[u]\)) will be leaked during update.

Obviously, if \(\text{CorruptWriter}, i) \in \mathcal{H}, \mathcal{A}\) will have the knowledge of the update token. In particular, if writer \(i\) has been corrupted, the BF representation of keywords in \(w_k\), as well as the position \(u_k\) is divulged. Thus, the oracle only answers when two update tuples are similar in this case.

The indistinguishability between \(\text{Hybrid}_L\) and \(\text{Hybrid}_{L+1}\) is guaranteed by IND-CPA security of the symmetric encryption scheme \(\mathcal{E}\) and computationally indistinguishable security of \(\mathcal{KH-PRF}\).

4. **RevokeO** on \((\{i_k\}_{k \in \{0,1\}})\): The oracle answers the queries when \(i = i_0 = i_1\), since the writer identifier will be leaked in permission revocation. In addition, if \(\text{CorruptWriter}, i) \in \mathcal{H}, \mathcal{A}\) will still have the knowledge of the updated secret tokens. Specifically, if writer \(i\) has been corrupted, \(\mathcal{A}\) who holds \(\kappa_{w_0}\) can still decrypt to obtain secret keys contained in secret tokens of the writer \(i\) even after they are updated.

The indistinguishability between \(\text{Hybrid}_L\) and \(\text{Hybrid}_{L+1}\) is guaranteed by IND-CPA security of the symmetric encryption scheme \(\mathcal{E}\) and computationally indistinguishable security of \(\mathcal{KH-PRF}\).

By repeating the above procedure for \(l \in \{1, \ldots, t - 1\}\), we conclude that \(\mathcal{A}\) cannot distinguish \(\text{Hybrid}_L = \text{IND}^0_{\text{MUSES}}\) from \(\text{Hybrid}_L = \text{IND}^1_{\text{MUSES}}\). Thus, MUSES is \(\mathcal{L}\)-adaptively-secure.

---

**B QUERY-INDEPENDENT PREPROCESSING**

Figure 13 presents the time for preprocessing of our MUSES to run \(\text{LSS.ShrTrns}\) in the two-sever setting, which shows that it is linear to the number of writers. The overhead of the preprocessing phase mostly comes from Share Translation protocol between two servers: \((\Delta; a, b) \rightarrow \text{TS.ShrTrns}(\pi; 1^t)\), which lets one server obtain \(a, b\), and the other server learn the corresponding translation function \(\Delta \rightarrow b = \pi(a)\) without revealing the permutation \(\pi\). However, as the preprocessing phase is independent with the search queries, it can be performed in the offline phase, then the results are stored in a temporary memory and used when a search happens. We implement the primitive version of Share Translation protocol in [18] for our evaluation. The advanced version of it uses special structure of Benes permutation network to accelerate and achieve better performance [18].

**C PREVENTING ROLLBACK ATTACKS**

We present a (simple) extension to our semi-honest MUSES construction to prevent rollback attacks, where a corrupt server may omit some updates from the writers or use a tampered writer’ index to process the reader’s request. The high-level idea is to employ additional servers and perform “pair-wise” checking of the responses from different subsets of servers in processing the reader’s request. For simplicity, assume our original MUSES scheme uses two servers. We add one more server to detect rollback attacks as follows. Let \(W^*\) be the writer subset and \(w\) is the keyword that the reader wants to search. For each pair of servers \(P_1, P_2 \in \{P_1, P_2, P_3\}\), the reader executes the following:

- \(s_{ij} = (s_i, s_j) \rightarrow \text{SearchToken}(w, \mathcal{W}^*)\)
- \(O_{ij} \leftarrow \text{Search}(s_{ij}, \text{sk}, \{(i, st_i, \text{EIFD}_i, \text{PTK}(ni))\} \in [\pi_{\mathcal{W}^*})\)

\(\text{WLOG, assume } (P_2, P_3)\) are honest and \(P_1\) is corrupt, in which it uses a mutated search index \(\text{EIFD}'_i = \text{EIFD}_i + \epsilon\) (compared with \(\text{EIFD}_i\) in \(P_2\) and \(P_3\)). As \(\text{DPF}\) and \(\text{KH-PRF}\) are computed on \(\text{EIFD}'_i\), there will be an error in \(P_1\)'s computation, making the responses \(O_{12}, O_{13} \neq O_{23}\).

By checking the consistency of \(O_{12}, O_{13},\) and \(O_{23}\), the reader can tell whether there is a rollback attack happens, and abort the protocol accordingly. Note that this strategy can only detect the rollback attack, a special case of malicious behavior on integrity, but is unable to tell which server is corrupted. Meanwhile, a malicious adversary can deviate from the protocol to not only compromise the integrity but also the privacy of the reader’s query. That requires a more comprehensive investigation to completely achieve malicious security, which we leave as our future work.