Oblivious Dynamic Searchable Encryption via Distributed PIR and ORAM

Thang Hoang* Attila A. Yavuz* Betul F. Durak† Jorge Guajardo‡

Abstract

Dynamic Searchable Symmetric Encryption (DSSE) allows to delegate search/update operations over encrypted data via an encrypted index. However, DSSE is known to be vulnerable against statistical inference attacks, which exploits information leakages from access patterns on encrypted index and files. Although generic Oblivious Random Access Machine (ORAM) can hide access patterns, it has been shown to be extremely costly to be directly used in DSSE setting.

We developed a series of Oblivious Distributed DSSE schemes that we refer to as ODSE, which achieve oblivious access on the encrypted index with a high security and improved efficiency over the use of generic ORAM. Specifically, ODSE schemes are 3 - 57× faster than applying the state-of-the-art generic ORAMs on encrypted dictionary index in real network settings. One of the proposed ODSE schemes offers desirable security guarantees such as information-theoretic security with robustness against malicious servers. These properties are achieved by exploiting some of the unique characteristics of searchable encryption and encrypted index, which permits us to harness the computation and communication efficiency of multi-server PIR and Write-Only ORAM simultaneously. We fully implemented ODSE and conducted extensive experiments to assess the performance of our proposed schemes in a real cloud environment.

1 Introduction

Data outsourcing allows a client to store her data on the cloud to reduce data management and maintenance costs. Despite its merits, cloud services come with severe privacy issues. The client may encrypt the data with standard encryption to protect its privacy. However, standard encryption prevents the client from performing basic operations (e.g., search/update) over the outsourced encrypted data. This significantly degrades the benefits of cloud services.

1.1 State-of-the-art and Limitations

DSSE. The concept of searchable encryption was first proposed by Song et al. [27], which allows to perform search-only operations over encrypted data. Dynamic Searchable Symmetric Encryption (DSSE) was later introduced by Kamara et al. [19], which offers both search and

*Oregon State University, {hoangmin,attila.yavuz}@oregonstate.edu

†Rutgers University, fbdurak@cs.rutgers.edu

‡Robert Bosch Research and Technology Center, Jorge.GuajardoMerchan@us.bosch.com

This work was partially done while the first and third authors were visiting Robert Bosch RTC—LLC.

Table 1: Comparison of ODSE schemes and their ORAM-based counterparts when accessing the encrypted index **I**.

Scheme	Security				Delay (s)		Distributed Setting†	
	Forward privacy	Backward privacy	Hidden access pattern‡	Encrypted index*	Search	Update	Privacy level	Robustness
Standard DSSE [8]	✗	✗	✗	Computational	0.036	0.62	-	-
Path-ORAM[29]	✓	✓	Computational	Computational	160.6		-	-
Ring-ORAM [25]	✓	✓	Computational	Computational	137.4		-	-
$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$	✓	✓	Computational	Computational	**2.8**		$\ell - 1$	✗
$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$	✓	✓	Computational	Computational	**6.3**		t	✓
$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$	✓	✓	**Information theoretic**	**Information theoretic**	**7.1**		t	✓

This table compares the performance of ODSE schemes with standard DSSE and ORAM-based counterparts under real network setting with the encrypted index of 9×10^{10} keyword-file pairs. The index was stored on Amazon EC2 clouds (see Section 6 for detailed analysis). The delay of all schemes were measured in the average-case cost.
We simulated the use of generic ORAM on the DSSE encrypted index with the round-trip optimization in [12] for comparison. We selected Path-ORAM [29] and Ring-ORAM [25] since they are the most efficient ORAM schemes.
∗ The encrypted index in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is "encrypted" by SSS to be information-theoretically secure. In other schemes, it is IND-CPA encrypted and therefore, it is computationally secure.
‡ All ODSE schemes perform both search and update protocols to hide the actual operation type. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ achieve information-theoretic security for keyword search, and computational security for update, and therefore, their overall security is computational.
† ℓ is the total number of servers, $t < \ell$ is the number of colluding servers (privacy parameter).

update capabilities over encrypted data via an encrypted index **I** representing keyword-file relationships in encrypted data $\mathcal{F}$. Many DSSE schemes have been proposed, each offering various performance, functionality and security trade-offs (e.g., [18, 19, 8, 6, 22, 30, 23]).

Information leakage in DSSE. It is known that all DSSE schemes leak significant information through search and access patterns, which are vulnerable to statistical inference attacks [17, 20, 7, 31, 24]. These attacks can reveal sensitive information about encrypted queries and files [20, 7, 24]. There are two sources of information leakages in DSSE: (i) leakages through search and update on encrypted index **I**, (ii) leakages due to access on encrypted files $\mathcal{F}$. A notable attack by Zhang et al. [31] has indicated that, *future research on DSSE should focus on sealing access pattern leakages rather than accepting them by default.* Unless these leakages are prevented, a trustworthy deployment of DSSE for privacy-critical applications may not be possible.

Existing Approaches to Reduce Information Leakages in DSSE. Several attempts relying on trivial strategies [10, 16] are either impractical or unable to completely seal all leakages in DSSE access patterns. Generic Oblivious Random Access Machine (ORAM) [14][1] can hide access patterns, and therefore, it can prevent most of the information leakages in DSSE. Garg et al. [12] proposed TWORAM scheme, which optimizes the round-trip communication under $\mathcal{O}(1)$ client storage when using ORAM to hide *file access patterns*[2] in DSSE. Despite its merits, prior studies (e.g., [28, 8, 21]) showed that generic ORAM (e.g., [29]) is still costly to be used in DSSE due to its $\mathcal{O}(\log N)$ communication overhead. Although several ORAMs with $\mathcal{O}(1)$ bandwidth complexity have been introduced recently, they are still extremely costly due to the use of fully/partially Homomorphic Encryption (HE). The performance of such schemes has been shown to be worse than that of ORAM with $\mathcal{O}(\log N)$ bandwidth overhead [2].

[1] By generic ORAM, we mean oblivious access techniques that can hide the operation type (i.e., whether it is read or write), as opposed to PIR or Write-Only ORAM.

[2] It differs from the objective of this paper, where we focus on hiding access patterns on the encrypted index in DSSE (see Section 6 for clarification).

1.2 Research Objective

As mentioned previously, it is imperative to seal information leakages from accessing encrypted files $\mathcal{F}$ and encrypted index $\mathbf{I}$ to achieve a secure DSSE. Since $\mathcal{F}$ is unstructured and the size of individual files in $\mathcal{F}$ might be arbitrarily large, to the best of our knowledge, generic ORAM seems to be the only option for oblivious access on $\mathcal{F}$.

The objective of this paper is to design oblivious access techniques on encrypted index $\mathbf{I}$*, which are more efficient than the generic ORAM, by exploiting special properties of searchable encryption and* $\mathbf{I}$*.*

1.3 Our Contributions

We propose a series of Oblivious Distributed Encrypted Index $\mathbf{I}$ with the direct application on DSSE, which we refer to as ODSE. Our intuition is as follows.

Main idea. We first observe that, in DSSE, keyword search and file update on $\mathbf{I}$ are read-only and write-only operations, respectively. This observation permits us to leverage specific bandwidth-efficient oblivious access techniques for each operation such as multi-server Private Information Retrieval (PIR) (for search) and Write-Only ORAM (for update) rather than using generic ORAM.

The second observation is that, identifying an appropriate data structure for $\mathbf{I}$ is critical for the adaptation of the bandwidth-efficient cryptographic primitives mentioned above. We found that forward index and inverted index are the ideal choices for the file update and keyword search operations, respectively as proposed in [15]. However, doing search and update on two isolated indexes can cause an inconsistency, which requires the server to perform synchronization. The synchronization operation leaks significant information [15]. To avoid this problem, it is necessary to integrate both search index and update index in an efficient manner. Fortunately, this can be achieved by leveraging a two-dimensional index, which allows keyword search and file update to be performed in two separate dimensions without creating any inconsistency at their intersection. This strategy permits us to perform computation-efficient (multi-server) PIR on one dimension, and communication-efficient (Write-Only) ORAM on the other dimension to achieve oblivious search and update, respectively with a high efficiency. Note that this index has an extra security benefit by offering hidden size pattern via padding with a minimal storage overhead.

We introduce three ODSE schemes called $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$, each offering various desirable performance and security properties (see Table 1) as follows.

Desirable Properties.

- *Low end-to-end delay and bandwidth overhead:* The proposed ODSE schemes incur low-cost computation at the server side (e.g., XOR, basic modular arithmetic operations in finite fields). The experiments showed that ODSE schemes are 3 - 57× faster than the use of efficient generic ORAM schemes (e.g.,[29, 25]) on encrypted index with the round-trip optimization proposed in [12] under real network setting (see Table 1 and Section 6 for details).

- *Complete obliviousness and Information-theoretic security:* ODSE achieves the full obliviousness by sealing all information leakages including query types (search/update), access patterns and size patterns from accessing the encrypted index $\mathbf{I}$. More specifically, access patterns on the encrypted index $\mathbf{I}$ are computationally secure in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ schemes, in which

the search query is information-theoretically secure. The encrypted index $\mathbf{I}$ is computationally secure in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ schemes. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ offers *information-theoretic* security for encrypted index $\mathbf{I}$ and the access patterns on $\mathbf{I}$.

- *Robustness against malicious servers:* $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ can tolerate a certain number of malicious servers in the distributed system.

- *Full-fledged implementation and open-sourced framework:* We fully implemented all the proposed ODSE schemes and strictly evaluated their performance in the real-cloud setting. Our implementation accelerated server-side PIR computation by using multi-threading and highly optimized libraries (e.g., NTL). We will release ODSE for public use and wide adaptation.

It is clear that the standard DSSE constructions (e.g., [8]) are much faster, but also less secure than our proposed methods in the sense of leaking more information beyond the access patterns (e.g., forward-privacy, backward-privacy) over the encrypted index. ODSE schemes offer higher security by sealing all these leakages at the cost of performance, but are still more efficient than applying generic ORAM techniques, as shown in Table 1 and further analyzed in Section 6.

2 Preliminaries and Building Blocks

Notation. Operators $||$ and $(\cdot)^\top$ denote the concatenation and the transpose, respectively. $x \xleftarrow{\$} \mathcal{S}$ denotes that x is randomly and uniformly selected from $\mathcal{S}$. $|\mathcal{S}|$ denotes the cardinality of set $\mathcal{S}$. $\langle x \rangle_{\mathsf{bin}}$ denotes the binary representation of x. Given a security parameter θ, $\mathcal{E} = (\mathsf{Enc}, \mathsf{Dec}, \mathsf{Gen})$ denotes an IND-CPA symmetric encryption comprised of three algorithms: key generation $\kappa \leftarrow \mathcal{E}.\mathsf{Gen}(1^\theta)$; encryption of message M with key κ and counter c as $C \leftarrow \mathcal{E}.\mathsf{Enc}_\kappa(M, c)$; decryption as $M \leftarrow \mathcal{E}.\mathsf{Dec}_\kappa(C, c)$. KDF is a keyed derivation function. $\oplus$ denotes the XOR operation. We denote $\mathbf{u} \cdot \mathbf{v}$ as the inner product of two vectors of the same length. We denote a finite field as $\mathbb{F}_p$, where p is a prime. $I[i]$ denotes accessing i-th component of I. Given a matrix $\mathbf{I}$, $\mathbf{I}[*, j \dots j']$ denotes accessing columns j to j' of $\mathbf{I}$.

Shamir Secret Sharing. (t, ℓ)-threshold Shamir Secret Sharing (SSS) scheme [26] is presented in Algorithm 1. Given a secret $\alpha \in \mathbb{F}_p$ to be shared, the dealer generates a random t-degree polynomial f and evaluates $f(x_i)$ for party $\mathcal{P}_i \in \{\mathcal{P}_1, \dots, \mathcal{P}_\ell\}$, where x_i is a deterministic non-zero element of $\mathbb{F}_p$ to identify party $\mathcal{P}_i$ and this information is public (SSS.$\mathsf{CreateShare}$ Algorithm). We denote the share for $\mathcal{P}_i$ as $[\![\alpha]\!]_i$. The secret can be reconstructed by combining at least $t+1$ correct shares with Lagrange interpolation (SSS.$\mathsf{Recover}$ Algorithm). It is also possible to recover the secret from a number of incorrect SSS-shares via Reed-Solomon decode or list decoding algorithms [4, 13]. We use this property to extend our schemes into the malicious setting (see Appendix A.2).

SSS is a t-private secret sharing scheme, so any combinations of t shares leak no information about the value. SSS offers homomorphic properties including addition, scalar multiplication, and *partial* multiplication. We extend the notion of Shamir share of value to indicate the share of vector. Given a vector $\mathbf{v} = (v_1, \dots, v_n)$, $[\![\mathbf{v}]\!]_i = ([\![v_1]\!]_i, \dots, [\![v_n]\!]_i)$ indicates the share of $\mathbf{v}$ for party $\mathcal{P}_i$, in which components in $[\![\mathbf{v}]\!]$ are shares of components in $\mathbf{v}$.

Private Information Retrieval (PIR). PIR enables retrieval of a data item from a (unencrypted) public database server without revealing which item is retrieved. We present the

Algorithm 1 Shamir Secret Sharing (SSS) scheme

$(\llbracket\alpha\rrbracket_1, \ldots, \llbracket\alpha\rrbracket_\ell) \leftarrow \mathsf{SSS.CreateShare}(\alpha, t)$: Create t-private shares of α

1: $(a_1, \ldots, a_t) \xleftarrow{\$} \mathbb{F}_p$
2: **for** $k = 1, \ldots, \ell$ **do**
3: $\quad \llbracket\alpha\rrbracket_k \leftarrow \alpha + \sum_{u=1}^{t} a_u \cdot x_k^u$
4: **return** $(\llbracket\alpha\rrbracket_1, \ldots, \llbracket\alpha\rrbracket_\ell)$

$\alpha \leftarrow \mathsf{SSS.Recover}(\{\mathcal{A}\}, t)$: Recover the value

1: Randomly select $t+1$ shares $\{\llbracket\alpha\rrbracket_{x_i}\}_{i=1}^{t+1}$ among $\mathcal{A}$
2: $g(x) \leftarrow \mathsf{LagrangeInterpolation}\big(\{(x_i, \llbracket\alpha\rrbracket_{x_i})\}_{i=1}^{t+1}\big)$
3: **return** α, where $\alpha \leftarrow g(0)$

definition of PIR in the distributed setting as follows.

Definition 1 (multi-server PIR [13, 4])**.** *Let $\mathbf{b} = (b_1, \ldots, b_n)$ be a database consisting of n items being stored in ℓ servers. A multi-server PIR protocol consists of three algorithms as follows. Given an item b in $\mathbf{b}$ to be retrieved, the client creates queries $(\rho_1, \ldots, \rho_\ell) \leftarrow \mathsf{PIR.CreateQuery}(i)$ and distributes ρ_j to server $\mathcal{S}_j$. Each server $\mathcal{S}_j$ responds with an answer $r_j \leftarrow \mathsf{PIR.Retrieve}(\rho_j, \mathbf{b})$. Upon receiving ℓ answers, the client computes the value of item b by invoking the reconstruction algorithm $b \leftarrow \mathsf{PIR.Reconstruct}(r_1, \ldots, r_\ell)$.*
Correctness: *A multi-server PIR is* correct *if the client can obtain the correct value of b from ℓ answers via* $\mathsf{PIR.Reconstruct}$ *algorithm with the probability 1.*
t-privacy: *A multi-server PIR is* t-private *if $\forall j, j' \in \{1, \ldots, n\}$, $\forall \mathcal{L} \subseteq \{1, \ldots, \ell\}$ s.t. $|\mathcal{L}| \leq t$, the probability distributions of $\big\{\rho_{i\in\mathcal{L}} : (\rho_1, \ldots, \rho_\ell) \leftarrow \mathsf{PIR.CreateQuery}(j)\big\}$ and $\big\{\rho'_{i\in\mathcal{L}} : (\rho'_1, \ldots, \rho'_\ell) \leftarrow \mathsf{PIR.CreateQuery}(j')\big\}$ are identical.*

We recall two efficient multi-server PIR protocols as follows.

- *XOR-based PIR:* [9] (Algorithm 2). It relies on XOR to perform the private retrieval, in which the database $\mathbf{b}$ contains n items b_i, each being interpreted as a m-bit string.
- *SSS-based PIR:*[13, 4] (Algorithm 3). It relies on SSS to improve the robustness of multi-server PIR, in which the database $\mathbf{b}$ contains n items b_i, each being interpreted as an element of $\mathbb{F}_p$.

Algorithm 2 XOR-based PIR [9]

$(\rho_1, \ldots, \rho_\ell) \leftarrow \mathsf{PIR^{xor}.CreateQuery}(j)$: Create select query

1: Initialize binary string $e \leftarrow 0^m$ and set $e[j] \leftarrow 1$; $\rho_i \xleftarrow{\$} \{0,1\}^m$ for $1 \leq i < \ell$
2: $\rho_\ell \leftarrow \rho_1 \oplus \ldots \rho_{\ell-1} \oplus\ e$
3: **return** $(\rho_1, \ldots, \rho_\ell)$

$r_i \leftarrow \mathsf{PIR^{xor}.Retrieve}(\rho_i, \mathbf{b})$: Retrieve an item in the DB $\mathbf{b}$

1: $r_i \leftarrow \bigoplus_{j\in\mathcal{J}} b_j$ where $\mathcal{J} = \{j : \rho_i[j] = 1\}$
2: **return** r_i

$b \leftarrow \mathsf{PIR^{xor}.Reconstruct}(r_1, \ldots, r_\ell)$: Reconstruct the item

1: **return** b, where $b = \bigoplus_{k=1}^{\ell} r_k$

Algorithm 3 SSS-based PIR [13, 4]

$(\llbracket \mathbf{e} \rrbracket_1, \ldots, \llbracket \mathbf{e} \rrbracket_\ell) \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{CreateQuery}(j)$: Create select queries
1: Let $\mathbf{e} := (e_1, \ldots, e_n)$, where $e_j \leftarrow 1$, $e_i \leftarrow 0$ for $1 \le i \ne j \le n$
2: **for** $i = 1, \ldots, n$ **do** $(\llbracket e_i \rrbracket_1, \ldots, \llbracket e_i \rrbracket_\ell) \leftarrow \mathrm{SSS}.\mathsf{CreateShare}(e_i, t)$
3: $\llbracket \mathbf{e} \rrbracket_i \leftarrow (\llbracket e_1 \rrbracket_i, \ldots, \llbracket e_n \rrbracket_i)$, for $1 \le i \le \ell$
4: **return** $(\llbracket \mathbf{e} \rrbracket_1, \ldots, \llbracket \mathbf{e} \rrbracket_\ell)$

$\llbracket b \rrbracket_i \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Retrieve}(\llbracket \mathbf{e} \rrbracket_i, \mathbf{b})$: Retrieve the item
1: **return** $\llbracket b \rrbracket_i$, where $\llbracket b \rrbracket_i \leftarrow \llbracket \mathbf{e} \rrbracket_i \cdot \mathbf{b}$

$b \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\llbracket b \rrbracket_1, \ldots, \llbracket b \rrbracket_\ell, t)$: Recover the item
1: **return** b, where $b \leftarrow \mathrm{SSS}.\mathsf{Recover}(\llbracket b \rrbracket_1, \ldots, \llbracket b \rrbracket_\ell, t)$

ORAM. ORAM allows users to access their own data stored on the cloud without leaking to the storage provider which data blocks have been accessed. We give the security definition of ORAM as follows.

Definition 2 (ORAM security [29])**.** *Let $\vec{o} = (\mathsf{op}_i, u_i, \mathsf{data}_i)_{i=1}^{q}$ be a data request sequence, where $\mathsf{op}_i \in \{\mathit{read}(u_i, \mathsf{data}_i), \mathit{write}(u_i, \mathsf{data}_i)\}$, u_i is the logical address to be read/written and data_i is the data at u_i to be read/written. Let $\boldsymbol{AP}(\vec{o})$ be an access pattern observed by the server $\mathcal{S}$ given a data request sequence σ. An ORAM scheme is* secure *if for any two data request sequences $\vec{o}$ and $\vec{o}'$ of the same length, their access patterns $\boldsymbol{AP}_j(\vec{o})$ and $\boldsymbol{AP}_j(\vec{o}')$ are computationally indistinguishable.*

Write-Only ORAM: Blass et al. [5] proposed a Write-Only ORAM scheme in the context of hidden volume encryption. This scheme aims to only hide the write patterns, instead of both read/write operations as in the generic ORAM model. Intuitively, there are $2n$ memory slots that are used to store n blocks. For each block to be written, the client puts it into a stash and then reads λ slots chosen uniformly at random among $2n$ slots. The client decrypts λ slots, flushes data from the stash to empty slots, re-encrypts and write them back. By selecting λ sufficiently large (e.g., $\lambda = 80$), one can achieve negligible write failure probability without using the stash. In case the stash is used to achieve small λ (e.g., $\lambda = 4$), its size is proven to have an upper bound.

3 Definition and Models

System Model. Our system model comprises a client and ℓ servers $\boldsymbol{\mathcal{S}} = (\mathcal{S}_1, \ldots, \mathcal{S}_\ell)$, each storing a version of the encrypted index. In our system, the encrypted files are stored on a separate server different from $\boldsymbol{\mathcal{S}}$ (as in [16]). While encrypted files can be securely accessed via a generic ORAM (e.g., [29, 25]), this paper only focuses on oblivious access on distributed encrypted index $\mathcal{I}$ on $\boldsymbol{\mathcal{S}}$.

We give the definition of ODSE as follows.

Definition 3. *An Oblivious Distributed Dynamic Searchable Symmetric Encryption (ODSE) scheme is a tuple of one algorithm and two protocols ODSE =* $(\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ *such that:*

1. $(\mathcal{I}, \sigma) \leftarrow \mathsf{Setup}(\mathcal{F})$*: Given a set of files $\mathcal{F}$ as input, the algorithm outputs a distributed encrypted index $\mathcal{I}$ and a client state σ.*

2. $\mathcal{R} \leftarrow \mathsf{Search}(w, \mathcal{I}, \sigma)$: *The client inputs a keyword w to be searched and the state σ; the servers input the distributed encrypted index $\mathcal{I}$. The protocol outputs to the client a set $\mathcal{R}$ containing identifier of files in which w appears.*

3. $(\mathcal{I}', \sigma') \leftarrow \mathsf{Update}(f_{id}, \mathcal{I}, \sigma)$: *The client inputs the updated file f_{id} and a state σ; the servers input the distributed encrypted index $\mathcal{I}$. The protocol outputs a new state σ' and the updated index $\mathcal{I}'$ to the client and servers, respectively.*

Security Model. We define the security of ODSE in the semi-honest setting as follows.

Definition 4 (ODSE security)**.** *Let $\vec{o} = (\mathsf{op}_1, \ldots, \mathsf{op}_q)$ be an operation sequence, where $\mathsf{op}_i \in \{\mathsf{Search}(w, \mathcal{I}, \sigma), \mathsf{Update}(f_{id}, \mathcal{I}, \sigma)\}$, w is a keyword to be searched and f_{id} is a file with identifier id whose relationship with unique keywords in the distributed encrypted index $\mathcal{I}$ need to be updated, and σ denotes with a client state information. Let $\boldsymbol{ODSE}_j(\vec{o})$ represent the ODSE client's sequence of interactions with server $\mathcal{S}_j$, given an operation sequence $\vec{o}$.*
Correctness: *An ODSE is* correct *if, for any operation sequence $\vec{o}$, $\{\boldsymbol{ODSE}_1, \ldots, \boldsymbol{ODSE}_\ell\}$ returns data consistent with $\vec{o}$, except with a negligible probability.*
t-security*: An ODSE is* t-secure *if $\forall \mathcal{L} \subseteq \{1, \ldots, \ell\}$ s.t. $|\mathcal{L}| \leq t$, for any two operation sequences $\vec{o}$ and $\vec{o}'$ where $|\vec{o}| = |\vec{o}'|$, the views $\{\boldsymbol{ODSE}_{i \in \mathcal{L}}(\vec{o})\}$ and $\{\boldsymbol{ODSE}_{i \in \mathcal{L}}(\vec{o}')\}$ observed by a coalition of up to t servers are (perfectly, statistically or computationally) indistinguishable.*

ODSE operation obliviousness. As defined in Definition 3, keyword search and file update are the two main operations in searchable encryption. Given that these operations might incur different procedures, we can invoke both search and update protocols for any actual action to achieve the operation obliviousness according to Definition 4.

4 The Proposed ODSE Schemes

In this section, we present data structures of ODSE and two ODSE instantiations, each offering different performance and security levels.

4.1 ODSE Data Structures

Let f_{id} and w denote a file with unique identifier id and a (key)word in a file, respectively. Given an incidence matrix $\mathbf{I}$, the keyword-file relationships are represented via cell values $\mathbf{I}[i, j] \in \{0, 1\}$. Each keyword and file is assigned to a unique row and column index, respectively. Therefore, each row of $\mathbf{I}$ contains the search result of a keyword while each column contains the data (unique keywords) of a file. Since Write-Only ORAM is used for file update, the number of columns are doubled. Hence, given that there are M keywords and N files, the size of $\mathbf{I}$ will be $M \times 2N$.

We leverage two static hash tables T_w, T_f as in [30] to keep track of the location of keywords and files in $\mathbf{I}$, respectively. They are of structure: $T := \langle \mathsf{key}, \mathsf{value} \rangle$, where key is a keyword or file ID and value is the (row/column) index of key in $\mathbf{I}$, which can be retrieved as $\mathsf{value} \leftarrow T[\mathsf{key}]$. We denote $\mathcal{D}$ as the set of empty columns that are not assigned to any particular files.

4.2 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$: Fast ODSE

We introduce $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, an ODSE scheme that offers a low search delay by using XOR trick. We present $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ in Scheme 1 with the following highlights.

Scheme 1 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ scheme

$(\mathcal{I}, \sigma) \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Setup}(\mathcal{F})$: Generate distributed encrypted index $\mathcal{I}$

1: Let Π and Π' be a random permutation on $\{1, \ldots, 2N\}$ and $\{1, \ldots, M\}$ resp.
2: $\kappa \leftarrow \mathsf{Gen}(1^\theta)$; $\mathbf{I}'[*, *] \leftarrow 0$; $\mathbf{c} \leftarrow (c_1, \ldots, c_{2N})$ where $c_i \leftarrow 1$ for $1 \le i \le 2N$
3: Extract keywords $(w_1, \ldots, w_m)$ from files $\mathcal{F} = \{f_{id_1}, \ldots, f_{id_n}\}$
4: $T_f[id_j] \leftarrow \Pi(j)$ for $1 \le j \le n$; $T_w[w_i] \leftarrow \Pi'(i)$ for $1 \le i \le m$
5: $\mathcal{D} \leftarrow \Pi(j)$ for $n \le j \le 2N$
6: $\mathbf{I}'[x_i, y_j] \leftarrow 1$ if $w_i \in f_{id_j}$ where $x_i \leftarrow T_w[w_i]$, $y_j \leftarrow T_f[id_j]$ for $1 \le i \le m$, $1 \le j \le n$
7: **for** $i = 1, \ldots, M$ **do** $\tau_i \leftarrow \mathsf{KDF}_\kappa(i)$; $\mathbf{I}[i, j] \leftarrow \mathcal{E}.\mathsf{Enc}_{\tau_i}(\mathbf{I}'[i, j], j||c_j)$ for $1 \le j \le 2N$
8: Let $\mathcal{I}$ contain ℓ copies of $\mathbf{I}$ and $\sigma \leftarrow (\kappa, T_w, T_f, \mathbf{c}, \mathcal{D})$
9: **return** $(\mathcal{I}, \sigma)$ `# The client sends each copy of` $\mathbf{I}$ `to each server`

$\mathcal{R} \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Search}(w, \mathcal{I}, \sigma)$: Search keyword w

Client:
1: $j \leftarrow T_w[w]$; $(\rho_1, \ldots, \rho_\ell) \leftarrow \mathsf{PIR}^{\mathsf{xor}}.\mathsf{CreateQuery}(j)$
2: Send ρ_i to $\mathcal{S}_i$, for $1 \le j \le n$
Server: each $\mathcal{S}_i \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving ρ_i **do**
3: $\hat{I}_i \leftarrow \mathsf{PIR}^{\mathsf{xor}}.\mathsf{Retrieve}(\rho_i, \mathbf{I}_i)$; Send $\hat{I}_i$ to the client
Client: On receive $(\hat{I}_1, \ldots, \hat{I}_\ell)$ from ℓ servers
4: $I \leftarrow \mathsf{PIR}^{\mathsf{xor}}.\mathsf{Reconstruct}(\hat{I}_1, \ldots, \hat{I}_\ell)$
5: $\tau_j \leftarrow \mathsf{KDF}_\kappa(j)$; $I'[j'] \leftarrow \mathcal{E}.\mathsf{Dec}_{\tau_i}(I[j'], j'||c_{j'})$ for $1 \le j' \le 2N$
6: **return** $\mathcal{R}$, where $\mathcal{R} = \left\{T_f.\mathsf{getKey}(j') : (I'[j'] = 1) \wedge \left((j' \notin \mathcal{D}) \vee (I'[j] \in S)\right)\right\}$

$(\mathcal{I}', \sigma') \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Update}(f_{id}, \mathcal{I}, \sigma)$: Update file f_{id}

Client: Initialize $\hat{I}[i] \leftarrow 0$ for $1 \le i \le 2N$
1: **for** each keyword $w_i \in f_{id}$ **do** $\hat{I}[x_i] \leftarrow 1$, where $x_i \leftarrow T_w[w_i]$
2: $S \leftarrow S \cup \{(id, \hat{I})\}$; $\mathcal{D} \leftarrow \mathcal{D} \cup T_f[id]$
3: Let $\mathcal{J}$ contain λ random-selected column indexes, send $\mathcal{J}$ to a random server $\mathcal{S}_l$
Server: $\mathcal{S}_l$ receiving $\mathcal{J}$ **do**
4: Send $\mathbf{I}_l[*, j]$, for each $j \in \mathcal{J}$ to the client
Client:
5: **for** each $j \in \mathcal{J}$ **do** $\tau_i \leftarrow \mathsf{KDF}_\kappa(i)$; $\mathbf{I}'[i, j] \leftarrow \mathcal{E}.\mathsf{Dec}_{\tau_i}(\mathbf{I}_l[i, j], j||c_j)$ for $1 \le i \le M$
6: **for** each $\hat{j} \in \mathcal{J} \cap \mathcal{D}$ **do** Pick a $(id, \hat{I})$ from S; $\mathbf{I}'[*, \hat{j}] \leftarrow \hat{I}^\top$; $\mathcal{D} \leftarrow \mathcal{D} \setminus \{\hat{j}\}$; $T_f[id] \leftarrow \hat{j}$
7: **for** each $j \in \mathcal{J}$ **do** $\hat{\mathbf{I}}[i, j] \leftarrow \mathcal{E}.\mathsf{Enc}_{\tau_i}(\mathbf{I}'[i, j], j||{+}{+}c_j)$ for $1 \le i \le M$
8: Send $\{\hat{\mathbf{I}}[*, j]\}_{j \in \mathcal{J}}$ to ℓ servers
Server: each $\mathcal{S}_i \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ **do**
9: $\mathbf{I}_i[*, j] \leftarrow \hat{\mathbf{I}}[*, j]$, for each $j \in \mathcal{J}$
10: **return** $(\mathcal{I}', \sigma')$ where $\mathcal{I}', \sigma'$ are $\mathbf{I}_i$ and σ have been updated

Setup: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Setup}$ algorithm presents the construction of an encrypted index $\mathbf{I}$. We encrypt $\mathbf{I}$ bit-by-bit, meaning that each cell is encrypted by a unique row key and a column counter pair (step 7). Each file and keyword is randomly assigned to a unique column and row, respectively. The client state consists of a master key κ, two hash tables (T_w, T_f), a counter vector $\mathbf{c}$ and the set of empty columns $\mathcal{D}$. The client sends a replica of the encrypted index $\mathbf{I}$ to each server.

Search: We leverage Chor's PIR on the row dimension of $\mathbf{I}$ for private retrieval of search data ($\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Search}$ protocol), where the keyword index is obtained from T_w. Since the data is IND-CPA encrypted rather than being public as in the traditional PIR model, the client needs to decrypt the retrieved data to obtain the final search result (step 5).

Update: Recall that the content (i.e., keywords) of a file is represented by a column in $\mathbf{I}$. Given a file f_{id} to be updated, we leverage Write-Only ORAM on the column dimension of $\mathbf{I}$ to update keyword-file pairs in f_{id} into an empty column ($\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Update}$ protocol). Since each server stores a replica of the encrypted index $\mathbf{I}$, the client only reads λ random columns from a single server first (step 3). After updating columns with are updated (step 6), the client re-encrypts and writes them back to all servers (steps 7– 8).

Security properties: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ requires all ℓ servers to be semi-honest and to answer the client. If there exists a malicious server, meaning that it is either down or returns an incorrect value, the correctness of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ will be compromised. Therefore, we propose a more robust ODSE scheme in the following section.

4.3 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$: Robust ODSE

We introduce $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ scheme which is robust against malicious servers by harnessing SSS-based PIR. The robustness is due to the ability to recover the secret shared by SSS in the presence of incorrect shares, as presented in Section 2. We present $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ in Scheme 2 with the following highlights:

Setup: In $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$, we construct $\mathbf{I}$ with IND-CPA encryption. Thus, the setup algorithm of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ is identical to that of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$.

Scheme 2 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ scheme

$(\mathbf{I}, \sigma) \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}.\mathsf{Setup}(\mathcal{F})$: Generate encrypted index

1: $(\mathbf{I}, \sigma) \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Setup}(\mathcal{F}, \mathbf{I}')$
2: **return** $(\mathbf{I}, \sigma)$

$\mathcal{R} \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}.\mathsf{Search}(w, \mathbf{I}, \sigma)$: Search keyword w

Client:
1: $j \leftarrow T_w[w]$, $\mathcal{R} \leftarrow \emptyset$
2: $(\llbracket \mathbf{e} \rrbracket_1, \ldots, \llbracket \mathbf{e} \rrbracket_\ell) \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{CreateQuery}(j)$
3: Send $\llbracket \mathbf{e} \rrbracket_i$, to $\mathcal{S}_i$, for $1 \leq j \leq n$
Server: each $\mathcal{S}_i \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $\llbracket \mathbf{e} \rrbracket_i$ **do**:
4: **for** $k = 1 \ldots, 2N'$ **do**
5: **for** $j = 1, \ldots, M$ **do**
6: $\langle c_{kj} \rangle_{\mathsf{bin}} \leftarrow \mathbf{I}'[j, (k-1) \cdot \lfloor \log_2 p \rfloor + 1, \ldots, k \cdot \lfloor \log_2 p \rfloor]$
7: $\mathbf{c}_k \leftarrow (c_{k1}, \ldots, c_{kM})$
8: $\llbracket b_k \rrbracket_i \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Retrieve}(\llbracket \mathbf{e} \rrbracket_i, \mathbf{c}_k)$
9: Send $\llbracket b_k \rrbracket_i$ to the client
Client: On receive $(\llbracket b_k \rrbracket_1, \ldots, \llbracket b_k \rrbracket_\ell)_{k=1}^{M}$ from ℓ servers
10: **for** $k = 1 \ldots, 2N'$ **do**
11: $b_k \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\llbracket b_k \rrbracket_1, \ldots, \llbracket b_k \rrbracket_\ell, t)$
12: $I \leftarrow \langle b_1 \rangle_{\mathsf{bin}} || \ldots || \langle b_{2N'} \rangle_{\mathsf{bin}}$
13: $\tau_j \leftarrow \mathsf{KDF}_\kappa(j)$; $I'[j'] \leftarrow \mathcal{E}.\mathsf{Dec}_{\tau_i}(I[j'], j' || c_{j'})$ for $1 \leq j' \leq 2N$
14: **return** $\mathcal{R}$, where $\mathcal{R} = \left\{ T_f.\mathsf{getKey}(j') : (I'[j'] = 1) \wedge \left((j' \notin \mathcal{D}) \vee (I'[j] \in S) \right) \right\}$

$(\mathcal{I}', \sigma') \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}.\mathsf{Update}(f_{id}, \mathcal{I}, \sigma)$: Update a file

1: $(\mathcal{I}', \sigma') \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Update}(f_{id}, \mathcal{I}, \sigma)$
2: **return** $(\mathcal{I}', \sigma')$

Search: We leverage the SSS-based PIR protocol on the row dimension of $\mathbf{I}$ to conduct keyword search. Each server performs the inner product between the search query and the encrypted index $\mathbf{I}$ via scalar multiplication and additive homomorphic properties of SSS. This operation requires row data in $\mathbf{I}$ to be represented as elements in $\mathbb{F}_p$. This condition is not satisfied by default since each row in $\mathbf{I}$ is a uniformly random binary string of length $2N$ due to IND-CPA encryption, which cannot be represented in $\mathbb{F}_p$ when $\log_2 p < 2N$. Therefore, we split the binary representation of each row of $\mathbf{I}$ into equally-sized substrings s_i s.t. $|s_i| < \log_2 p$. The inner product is performed iteratively between the search query and divided chunks from all rows in $\mathbf{I}$ (steps 4–8, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}.\mathsf{Search}$ Algorithm).

Update: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ leverages Write-Only ORAM on the column of $\mathbf{I}$ to perform file update. Since $\mathbf{I}$ in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ is constructed similarly to $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ (their Setup algorithms are identical), the update protocol of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ is identical to that of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$.

Security properties: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ requires at least $t+1$ servers among ℓ to be available and give the answers correctly, instead of all servers. The SSS-based PIR protocol allows $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ to tolerate a certain number of malicious servers in the system, as discussed in Appendix A.2.

Since $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ relies on IND-CPA encryption, the encrypted index and update operation are therefore only computationally secure. In next section, we introduce another robust ODSE scheme which achieves a complete information theoretical (IT)-security.

4.4 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$: Robust and IT-Secure ODSE

We introduce a $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ scheme that offers robustness against malicious servers and information-theoretic security for not only $\mathbf{I}$ but also any operations (search and update) on it. The main idea is to create $\mathbf{I}$ via SSS, and harness SSS-based PIR to conduct private search. The robustness comes from the ability to recover the secret shared by SSS in the presence of incorrect shares (see Section 5). We describe $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ in Scheme 2 with the following highlights.

Setup: We first generate an unencrypted index $\mathbf{I}'$ as in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Setup}$ algorithm (steps 1–6). Instead of using IND-CPA encryption to encrypt $\mathbf{I}'$, we create shares of $\mathbf{I}'$ by SSS and distribute them to corresponding servers. Since SSS operates on elements in $\mathbb{F}_p$, we split the binary representation of each row of $\mathbf{I}'$ into $\lfloor \log_2 p \rfloor$-bit chunks, and compute SSS share for each. So, the encrypted index $\mathbf{I}_i$ is the SSS share of $\mathbf{I}'$ for server $\mathcal{S}_i$, which is a matrix of size $M \times N'$, where $\mathbf{I}_i[i,j] \in \mathbb{F}_p$ and $N' = N/\lfloor \log_2 p \rfloor$.

Search: We leverage the SSS-based PIR protocol on the row dimension of $\mathbf{I}$ to conduct the keyword search. As each $\mathbf{I}_i$ stored on $\mathcal{S}_i$ is a share matrix, the inner product of two shares (i.e., $\mathbf{I}_i$ and search query) results in a share represented by a $2t$-degree polynomial. Therefore, the

Scheme 2 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ scheme

$(\mathcal{I}, \sigma) \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}.\mathsf{Setup}(\mathcal{F})$: Generate distributed encrypted index $\mathcal{I}$

1: Execute steps 1–6, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Setup}$ algorithm to construct $\mathbf{I}'$
2: **for** $i = 1, \ldots, M$ **do**
3: **for** $j = 1, \ldots, 2N'$ **do**
4: $\langle b_{ij} \rangle_{\mathsf{bin}} \leftarrow \mathbf{I}'[i, (j-1) \cdot \lfloor \log_2 p \rfloor + 1, \ldots, j \cdot \lfloor \log_2 p \rfloor]$
5: $(\mathbf{I}_1[i,j], \ldots, \mathbf{I}_\ell[i,j]) \leftarrow \mathrm{SSS}.\mathsf{CreateShare}(b_{ij}, t)$
6: **return** $(\mathcal{I}, \sigma)$, where $\mathcal{I} \leftarrow \{\mathbf{I}_1, \ldots, \mathbf{I}_\ell\}$ and $\sigma \leftarrow (T_w, T_f, \mathcal{D})$

Scheme 2 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ scheme (continued)

$\mathcal{R} \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}.\mathsf{Search}(w, \mathcal{I}, \sigma)$: Search keyword w

Client:

$j \leftarrow T_w[w]$; $(\llbracket \mathbf{e} \rrbracket_1, \ldots, \llbracket \mathbf{e} \rrbracket_\ell) \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{CreateQuery}(j)$

Send $\llbracket \mathbf{e} \rrbracket_i$, to $\mathcal{S}_i$, for $1 \leq j \leq n$

Server: each $\mathcal{S}_i \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $\llbracket \mathbf{e} \rrbracket_i$ **do**

for $k = 1 \ldots, 2N'$ **do** $\llbracket b_k \rrbracket_i \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Retrieve}(\llbracket \mathbf{e} \rrbracket_i, \mathbf{I}_i[*, k])$

Send $(\llbracket b_1 \rrbracket_i, \ldots, \llbracket b_{2N'} \rrbracket_i)$ to the client

Client: Receive $(\llbracket b_k \rrbracket_1, \ldots, \llbracket b_k \rrbracket_\ell)_{k=1}^{2N'}$ from ℓ servers

for $k = 1 \ldots, 2N'$ **do** $b_k \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\llbracket b_k \rrbracket_1, \ldots, \llbracket b_k \rrbracket_\ell, 2t)$

$I' \leftarrow \langle b_1 \rangle_{\mathsf{bin}} || \ldots || \langle b_{2N'} \rangle_{\mathsf{bin}}$

return $\mathcal{R}$, where $\mathcal{R} = \left\{ T_f.\mathsf{getKey}(j') : (I'[j'] = 1) \wedge \left((j' \notin \mathcal{D}) \vee (I'[j] \in S) \right) \right\}$

$(\mathcal{I}, \sigma') \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}.\mathsf{Update}(f_{id}, \mathcal{I}, \sigma)$: Update file f_{id}

Client: Initialize $I'[i] \leftarrow 0$ for $1 \leq i \leq 2N$

Execute steps 1–2 in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Update}$ algorithm

Let $\mathcal{J}$ contain λ' random block indexes, send $\mathcal{J}$ to random $t+1$ servers

Server: each $\mathcal{S}_i \in \{\mathcal{S}_{x_1}, \ldots, \mathcal{S}_{x_t+1}\}$ receiving $\mathcal{J}$ **do**

Send $(\mathbf{I}_i[*, j], \ldots, \mathbf{I}_i[*, j])$ for each $j \in \mathcal{J}$ to the client

Client:

for $i = 1 \ldots, M$ **do**

  $b_{ij} \leftarrow \mathrm{SSS}.\mathsf{Recover}(\langle \mathbf{I}_1[i, j], \ldots, \mathbf{I}_\ell[i, j] \rangle, t)$ for each $j \in \mathcal{J}$

  $\mathbf{I}'[i, j \cdot \lfloor \log_2 p \rfloor + 1, ...(j+1) \cdot \lfloor \log_2 p \rfloor] \leftarrow \langle b_{ij} \rangle_{\mathsf{bin}}$

$\mathcal{J}' \leftarrow \bigcup_{j \in \mathcal{J}} \left\{ j \cdot \lfloor \log_2 p \rfloor + 1, \ldots, (j+1) \cdot \lfloor \log_2 p \rfloor \right\} \cap \mathcal{D}$

for each $\hat{j} \in \mathcal{J}'$ **do** Pick a $(id, \hat{I})$ from S; $\mathbf{I}'[*, \hat{j}] \leftarrow \hat{I}^\top$; $\mathcal{D} \leftarrow \mathcal{D} \setminus \{\hat{j}\}$; $T_f[id] \leftarrow \hat{j}$

for each $j \in \mathcal{J}$ **do**

  for $i = 1 \ldots, M$ **do**

    $\langle b_{ij} \rangle_{\mathsf{bin}} \leftarrow \mathbf{I}'[i, j \cdot \lfloor \log_2 p \rfloor + 1, \ldots, (j+1) \cdot \lfloor \log_2 p \rfloor]$

    $(\hat{\mathbf{I}}_1[i, j], \ldots, \hat{\mathbf{I}}_\ell[i, j]) \leftarrow \mathrm{SSS}.\mathsf{CreateShare}(b_{ij}, t)$

  Send $(\hat{\mathbf{I}}_i[*, j])$ to $\mathcal{S}_i$ for $1 \leq i \leq \ell$

Server: each $\mathcal{S}_i \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $(\hat{\mathbf{I}}_i[*, j])$, $j \in \mathcal{J}$ **do**

$\mathbf{I}_i[*, j] \leftarrow \hat{\mathbf{I}}_i[*, j]$, for each $j \in \mathcal{J}$

return $(\mathcal{I}', \sigma')$ where $\mathcal{I}', \sigma'$ are $\mathbf{I}_i$ and σ have been updated

---

client needs to call $\mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}$ algorithm with the privacy parameter of $2t$ (instead of t in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$) to obtain the correct search result.

Update: Similar to $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, we leverage Write-Only ORAM on the column dimension of the encrypted index for the file update. Recall that Write-Only ORAM will access λ random columns of the unencrypted index $\mathbf{I}'$. In $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$, each column of the encrypted index $\mathbf{I}_i$ on $\mathcal{S}_i$ contains the share of $\lfloor \log_2 p \rfloor$ successive columns of $\mathbf{I}'$. Therefore, the client needs to read $\lambda' = \lceil \frac{\lambda}{\lfloor \log_2 p \rfloor} \rceil$ random columns of $\mathbf{I}_i$ from $t+1$ servers to reconstruct λ columns of $\mathbf{I}'$. After the update, the client creates new shares for the retrieved columns and writes them back to ℓ servers. We present this strategy in the $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}.\mathsf{Update}$ protocol.

Security properties: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ requires at least $2t+1$ servers to be available and to answer honestly. The encrypted index and all search/update operations on it are unconditionally secure due to SSS. This property allows $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ to be robust against a number of malicious servers as discussed in Section 5.

5 Security

In this section, We present the security of proposed ODSE schemes. We provide the proofs in Appendix A.1.

Theorem 1. *ODSE^{wo}_{xor} scheme is correct and computationally $(\ell-1)$-secure by Definition 4.*

Proof. See Appendix. □

Theorem 2. *ODSE^{wo}_{ro} scheme is correct and computationally t-secure by Definition 4.*

Proof. See Appendix. □

Theorem 3. *ODSE^{wo}_{it} scheme is correct and unconditionally (statistically) t-secure according to Definition 4.*

Proof. See Appendix. □

Robustness against malicious servers. Since $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ relies on SSS as the building block, it can be robust against malicious servers. The extension is straightforward by using special techniques such as Reed Solomon [4] or list decoding [13] algorithms to handle incorrect shares returned by the server, given that the Lagrange interpolation in SSS.$\mathsf{Recover}$ algorithm does not return a consistent value. We give a more detailed description on the malicious setting extension in Appendix A.2.

6 Experimental Evaluation

Implementation details. We implemented all ODSE schemes in C++. Specifically, we used `Google Sparsehash` to implement T_f and T_w hash tables. We utilized `Intel AES-NI` library to implement AES-CTR encryption/decryption in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$. We leveraged `Shoup's NTL` library for pseudo-random number generator and arithmetic operations over finite field. We used `ZeroMQ` library for client-server communication. We used multi-threading technique to accelerate PIR computation at the server. Our implementation is publicly available on github at

https://github.com/thanghoang/IM-DSSE/

Hardware and network settings. We used Amazon EC2 with `r4.4xlarge` instance type to deploy servers. Each server was equipped with 16 vCPUs Intel Xeon E5-2686 v4 @ 2.3 GHz and 122 GB RAM. For client, we used a laptop equipped with Intel Core i5-5287U CPU @ 2.90 GHz and 16 GB RAM. All machines ran Ubuntu 16.04. The client established a network connection with the server via WiFi. We used a home network data plan, which offers the download/upload throughput of 27/5 Mbps.

Dataset. We used subsets of the `Enron` database to build the encrypted index with different sizes from millions to billions of keyword-file pairs.

Instantiations of compared techniques. We compared ODSE with a standard DSSE scheme [8], and the use of generic ORAM on top of the DSSE encrypted index. The performance of all schemes was measured in average-case cost. We instantiated ODSE schemes and their counterparts with the following settings:

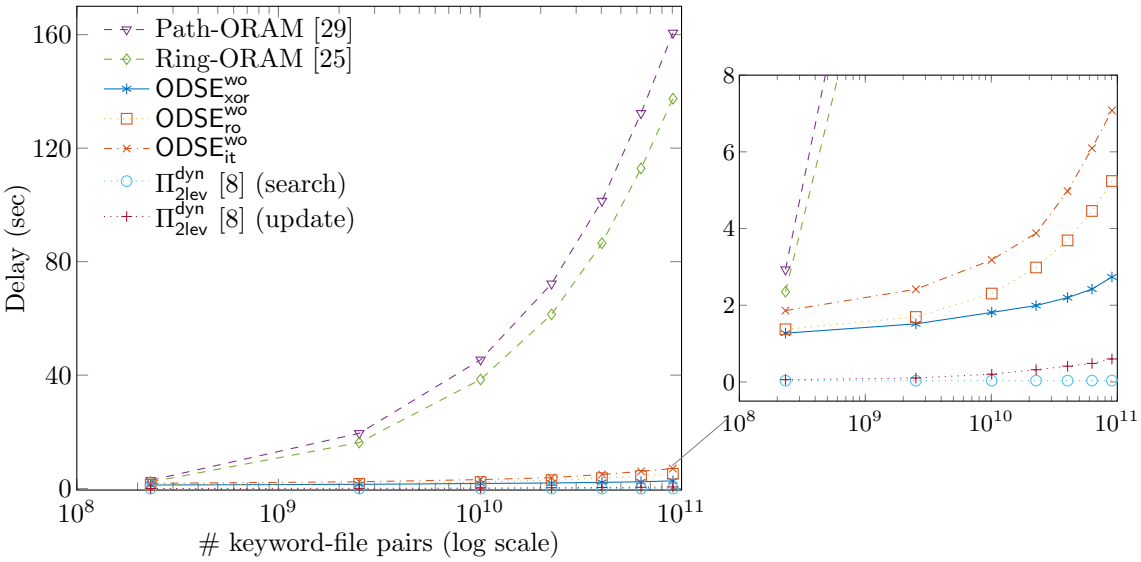


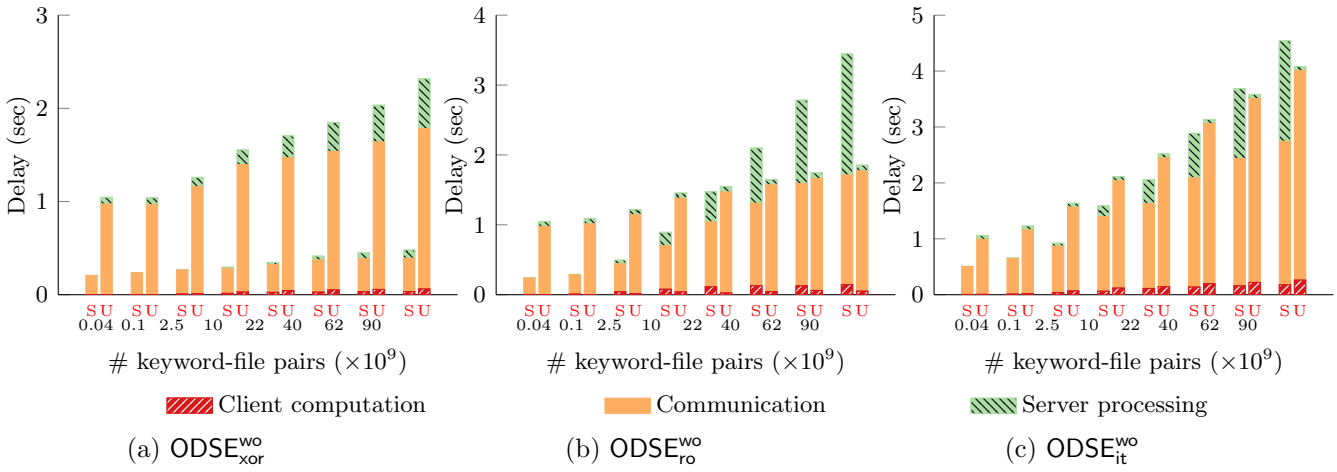
Figure 1: Latency of ODSE schemes and their counterparts.

• *ODSE*: We performed our experiments in the semi-honest setting, in which servers do not collude with each other. Therefore, we used two servers for ODSE^{wo}_{xor} and ODSE^{wo}_{ro} schemes, and three servers for ODSE^{wo}_{it} scheme. We selected $\lambda = 4$ for ODSE^{wo}_{xor} and ODSE^{wo}_{ro}, and $\lambda' = 4$ with $\mathbb{F}_p$ where p is a 16-bit prime for ODSE^{wo}_{it}. We note that selecting larger p (up to 64 bits) can reduce the PIR computation time, but also increase the bandwidth overhead as a trade-off. We chose 16-bit prime field to achieve a balance between computation and communication overhead according to the network constraints used in this study.

• *Standard DSSE*: We selected one of the most efficient DSSE schemes by Cash et al. in [8] (i.e., Π^{dyn}_{2lev} variant) to showcase the performance gap between ODSE and standard DSSE. We simulated the performance of Π^{dyn}_{2lev} using the same software/hardware environments and optimizations with ODSE (e.g., parallelization, AES-NI acceleration). Note that we did not use the Java implementation of this scheme available in Clusion library [1] for comparison due to its lack of hardware acceleration support (no AES-NI) and the difference between running environments (Java VM vs. native C). So for the simulation of Π^{dyn}_{2lev}, we used numbers that would be better than the Clusion Java implementation.

• *Simulation of using generic ORAM on DSSE data structure*: We selected Path-ORAM [29] and Ring-ORAM [25] as ODSE counterparts since they are the most efficient generic ORAM schemes being proposed to date. Note that we did not use other recent ORAMs (e.g.,[3, 11]) for comparison since their cost is higher than that of the selected ORAMs due to fully/partially homomorphic encryption. Moreover, there is a very limited numbers of studies focusing on hiding access patterns in DSSE except the TWORAM scheme in [12] which provides an oblivious access strategy to the encrypted files. This context is different from ours, where we only focus on oblivious access on the encrypted index. Therefore, we did not explicitly compare TWORAM with ODSE but instead, used one of their techniques to improve the performance of using generic ORAM on DSSE data structure. Specifically, we applied the selected ORAMs on the dictionary index containing (w_i, id_j) pairs suggested in [21] along with the round-trip optimization proposed in [12], to make the comparison as fair as possible.

Overall Results. Figure 1 presents the end-to-end delays of ODSE schemes and their counter-



(a) $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ (b) $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ (c) $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$

Figure 2: Detailed search (S) and update (U) costs of ODSE schemes.

parts, where both search and update are performed in ODSE schemes to hide the actual type of operation as discussed in Section 3. Clearly, ODSE offers a higher security than standard DSSE at the cost of a longer delay. However, ODSE schemes are still 3 - 57× faster if using generic ORAMs to hide the access patterns under different encrypted index sizes being experimented. Specifically, with an encrypted index containing ten billions of keyword-file pairs, $\Pi^{\mathsf{dyn}}_{\mathsf{2lev}}$ cost 36 ms and 600 ms to finish a search and update operation, respectively. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ took 2.8 seconds and 7.1 seconds respectively, to accomplish both keyword search and file update operations, compared with 160 seconds by using Path-ORAM with the round-trip optimization trick. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ is the most efficient in terms of search, whose delay was less than 1 second. This is due to the fact that $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ only computes XOR operations and the size of the search query is minimal (i.e., a binary string). $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ are more robust (e.g., malicious tolerant) and more secure (e.g., unconditional security) than $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ with the cost of higher search delay. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is slowest among the three ODSE schemes since it requires three servers and, therefore, the client needs to transmit more data. For file updates, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ achieved a similar delay since they have the same number of servers and incurred the same amount of data to be transmitted. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is slightly slower than $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ since the client transmitted data to three servers, instead of two. We can see that in many cases where it is not necessary to hide the operation types (search/update), using ODSE to conduct individual operations, especially the keyword search, is much more efficient than generic ORAM schemes. In the following, we dissect the total cost to investigate which factors contributed the most to the latency of ODSE schemes.

Detailed cost analysis. Figure 2 presents the total delays of separate keyword search and file update operations, as well as their detailed costs in ODSE schemes. Note that ODSE performs both search and update (one of them is dummy) to hide the actual type of operation performed by the client.

• *Client processing:* As shown in Figure 2, client computation contributed the least amount to the overall search delay (i.e., $< 10\%$) in all ODSE schemes. The client computation comprises the following operations: (1) Generate select queries (with SSS in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$, and PRG in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$); (2) SSS recovery (in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$) and IND-CPA decryption (in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$); (3) Filter dummy positions in the decrypted data. Note that it is possible to reduce the client-side delay in the online phase (by at least 50%-60%) via pre-computation of

some values such as row keys, select queries (only contain shares of 0 or 1). For the file update, the client performs decryption and re-encryption on λ columns (in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$), or SSS over λ' blocks (in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$). Since we used cryptographic acceleration (i.e., Intel AES-NI) and highly optimized number theory libraries (i.e., NTL), all these computations only contributed a small fraction of the overall delay as shown in Figure 2.

• *Client-server communication:* Data transmission is the dominating factor in the delay of ODSE schemes. The communication cost of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ is smaller than that of other ODSE schemes, since the size of search query and the data transmitted from servers are binary vectors. In $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$, these vectors are of 16-bit components $\in \mathbb{F}_p$. The communication overhead of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ can be reduced by using a smaller finite field, but with the cost of increased PIR computation at the server side.

• *Server processing:* The cost of PIR operations in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ is negligible as it only relies on XOR operations. The PIR computation of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is reasonable, as it operates on a bunch of 16-bit values. For update operations, the server-side cost is mainly due to memory accesses for column update. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ are highly memory access-efficient since we organized the memory layout for column-friendly access. This layout minimizes the memory access delay not only in update but also in search, since the inner product in PIR also accesses contiguous memory blocks by this organization. In $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, we stored the matrix for row-friendly access to permit efficient XOR operations during search. However, this requires file update to access non-contiguous memory blocks. Therefore, the file update in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ incurs a higher memory access delay than that of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ as shown in Figure 2.

Discussion and Limitations. In this paper, we focused on sealing information leakages from the access pattern on the DSSE encrypted index beyond using generic ORAMs. By leveraging bandwidth-efficient oblivious access techniques including multi-server PIR and Write-Only ORAM, ODSE achieves a high security with the following asymptotic costs. The communication of ODSE is $\mathcal{O}(M + N)$, where M and N are the (maximum) number of unique keywords and files in the database, respectively. Although the cost is linear, we have shown that, the actual communication overhead of ODSE is much lower than using generic ORAM to access the DSSE encrypted index, whose asymptotic bandwidth cost is poly-logarithmic but is hidden by large constant factors in reality. In practice, this results in up to a 57× improvement in performance. For instance, the transmission cost of using Path-ORAM on the DSSE encrypted index is $\mathcal{O}(r \log^2(N \cdot M))$, where r is the size of search/update results, which ,in many cases, might be worse than downloading the entire database [21]. The client and server computation costs are $\mathcal{O}(M)$ and $\mathcal{O}(N \cdot M)$, respectively. Despite the PIR computation over the entire encrypted index, we showed experimentally that, its delay was reasonable in practice, assuming that all possible optimizations (e.g., parallelization, multi-threading, assembly optimization) have been taken into account.

The main limitation of ODSE is the encrypted index size, which is $\mathcal{O}(N \cdot M)$. Given the database containing 300,000 files and 300,000 keywords used in this experiment, the size of encrypted index is 21 GB.

7 Conclusions

We proposed a new set of Oblivious Distributed DSSE schemes called ODSE, which achieve a full obliviousness, hidden size pattern, and low end-to-end delay simultaneously. Specifically,

$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ achieves the lowest end-to-end delay with the smallest communication overhead among all of its counterparts with the highest resiliency against colluding servers. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ achieves the highest level of privacy with information-theoretic security for access patterns and the encrypted index, along with the robustness against malicious servers. Our experiments demonstrated that ODSE schemes are one order of magnitude faster than the most efficient ORAM techniques over DSSE encrypted index. We fully implemented ODSE schemes and their counterparts on an actual cloud environment, and will open-source our software for a public use and wide adaptation.

References

[1] The clusion library. Available at `https://github.com/encryptedsystems/Clusion/`, 2017.

[2] I. Abraham, C. W. Fletcher, K. Nayak, B. Pinkas, and L. Ren. Asymptotically tight bounds for composing oram with pir. In *IACR International Workshop on Public Key Cryptography*, pages 91–120. Springer, 2017.

[3] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable oblivious storage. In *International Workshop on Public Key Cryptography*, pages 131–148. Springer, 2014.

[4] A. Beimel and Y. Stahl. Robust information-theoretic private information retrieval. In *International Conference on Security in Communication Networks*, pages 326–341. Springer, 2002.

[5] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 203–214. ACM, 2014.

[6] R. Bost. Sophos forward secure searchable encryption. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*. ACM, 2016.

[7] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 668–679. ACM, 2015.

[8] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. *IACR Cryptology ePrint Archive*, 2014:853, 2014.

[9] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[10] S. Cui, M. R. Asghar, S. D. Galbraith, and G. Russello. Obliviousdb: Practical and efficient searchable encryption with controllable leakage.

[11] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*, pages 145–174. Springer, 2016.

[12] S. Garg, P. Mohassel, and C. Papamanthou. Tworam: Round-optimal oblivious ram with applications to searchable encryption. *IACR Cryptology ePrint Archive*, 2015:1010, 2015.

[13] I. Goldberg. Improving the robustness of private information retrieval. In *2007 IEEE Symposium on Security and Privacy (SP'07)*, pages 131–148. IEEE, 2007.

[14] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[15] F. Hahn and F. Kerschbaum. Searchable encryption with secure and efficient updates. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 310–320. ACM, 2014.

[16] T. Hoang, A. Yavuz, and J. Guajardo. Practical and secure dynamic searchable encryption via oblivious access on distributed data structure. In *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*. ACM, 2016.

[17] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Annual Network and Distributed System Security Symposium – NDSS*, volume 20, page 12, 2012.

[18] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security*, pages 258–274. Springer, 2013.

[19] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 965–976. ACM, 2012.

[20] C. Liu, L. Zhu, M. Wang, and Y.-a. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[21] M. Naveed. The fallacy of composition of oblivious ram and searchable encryption. Technical report, Cryptology ePrint Archive, Report 2015/668, 2015.

[22] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. In *Security and Privacy (S&P), 2014 IEEE Symposium on*, pages 639–654. IEEE, 2014.

[23] G. S. Poh, J.-J. Chin, W.-C. Yau, K.-K. R. Choo, and M. S. Mohamad. Searchable symmetric encryption: Designs and challenges. *ACM Computing Surveys (CSUR)*, 50(3):40, 2017.

[24] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*. ACM, 2016.

[25] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. *IACR Cryptology ePrint Archive*, 2014:997, 2014.

[26] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[27] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 44–55, Washington, DC, USA, 2000. IEEE Computer Society.

[28] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *Annual Network and Distributed System Security Symposium – NDSS*, volume 14, pages 23–26, 2014.

[29] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications security*, pages 299–310. ACM, 2013.

[30] A. A. Yavuz and J. Guajardo. Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. In *Selected Areas in Cryptography – SAC 2015*, Lecture Notes in Computer Science. Springer International Publishing, August 2015.

[31] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 707–720, Austin, TX, 2016. USENIX Association.

A Appendix

A.1 Proofs

Proof of Theorem 1. We argue the correctness and security of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ in the semi-honest setting as follows:
Correctness: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ is correct iff the keyword search operation returns correct result and the file update operation is consistent. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ leverages Chor's PIR protocol for keyword search, which was proven to be correct in [9]. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ leverages Write-Only ORAM for file update, which was proven to achieve negligible inconsistent probability in [5].
Security: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ leverages Chor's PIR and therefore, achieves $(\ell - 1)$-privacy for keyword search as proven in [9]. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ leverages Write-Only ORAM for the file update, which incurs random access patterns over the encrypted index [5]. Since the index in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ is IND-CPA encrypted, such file update operations achieve the computational security. Notice that Write-Only ORAM is originally proposed in the single-server setting. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ performs Write-Only ORAM on ℓ server with an identical procedure (e.g., memory accesses are the same in ℓ servers), and therefore, the server coalition does not affect the security of Write-Only ORAM. For each actual search/update operation, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ performs both search and update protocol. Therefore, given the security of each search and update operations as discussed above, for any actual access operations, the views of $\ell-1$ colluding servers in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ are computationally indistinguishable. □

Proof of Theorem 2. We argue the correctness and security of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ in the semi-honest setting as follows:
Correctness: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ leverages the SSS-based PIR protocol for keyword search, which was proven to be correct in [4]. Similar to $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ uses Write-Only ORAM for file update, which was proven to achieve a negligible inconsistent probability in [5].
Security: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ leverages a SSS-based PIR protocol and therefore, achieves t-privacy for keyword search due to the t-privacy property of SSS, as shown in [4, 13]. Similar to $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ leverages Write-Only ORAM over IND-CPA encrypted database, which offers computational security as shown in [5]. For each actual operation, the client performs both search and update protocols. Therefore, the access pattern in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ is a computationally indistinguishable in the presence of t colluding servers. □

Proof of Theorem 3. We argue the correctness and security of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ in the semi-honest setting as follows:
Correctness: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ leverages a SSS-based PIR protocol over the shares of the encrypted index for keyword search. In this case, the answer from each server is the share represented by a $2t$-degree polynomial. There are more than $2t + 1$ honest servers in the system and therefore, the client can be able to reconstruct the secret correctly to obtain the search result. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ also leverages Write-Only ORAM for file update, which was proven to achieve negligible inconsistent probability in [5]. In overall, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is correct except with a negligible probability.
Security: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ leverages a SSS-based PIR protocol and therefore, achieves t-privacy for keyword search due to the t-privacy property of SSS. Recall that Write-Only ORAM incurs random memory access on the encrypted index. Meanwhile, this index in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is information-theoretically secure since it is shared by SSS. Therefore, any access patterns incurred by update operation in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ are information-theoretically (statistically) indistinguishable, in the presence of t colluding servers. Given that search and update are individually information-theoretically secure, we conclude that $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is information-theoretically secure in the coalition of up to t servers. □

A.2 Malicious Setting Extension

$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ scheme uses SSS as the building block and therefore, it can tolerate with a number of incorrect shares to recover the secret shared by SSS scheme. The main idea is to leverage error correction techniques such as Reed Solomon Decoding in [4] or List Decoding algorithms in [13] as follows:

Let α be the value and $[\![\alpha]\!]_i$ be the share of α for server $\mathcal{P}_i$ using $(t-\ell)$-SSS. For simplicity, we assume that all ℓ servers respond to the client, some of which return malicious answers. The Reed Solomon Decode in [4] can recover the α correctly from ℓ shares $[\![\alpha]\!]$, given the number of incorrect shares is:

$$t_m \leq t < \frac{\ell}{3}. \tag{1}$$

Goldberg et al. in [13] proposed to use List decoding algorithm with verification to tolerate more malicious servers. With this technique, the number of malicious servers is:

$$t_m \leq t < \ell - \lceil \sqrt{\ell t} \rceil. \tag{2}$$

To extend $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ schemes presented in Section 4 into the malicious setting, we insert the Reed Solomon decode [4] or List Decoding in [13] into the SSS.$\mathsf{Recover}$ Algorithm presented in Section 2 after the Lagrange interpolation, and verify if the recovered secret is consistent as follows. If there exists a incorrect answer among ℓ answers due to the malicious server, the Lagrange interpolation might return an inconsistent value, given that the incorrect answer is selected to participate in the reconstruction. To prevent this, the client will invoke correction techniques presented above to detect malicious servers and to obtain the correct value in case the interpolation function does not return a consistent result when performing on $t+1$ random shares several times. As indicated in Eq. (1) and (2), the minimum number of servers in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is also increased, compared with the semi-honest setting.

Notice that in the semi-honest setting, the update protocol of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ presented in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}.\mathsf{Update}$ Algorithm is communication-optimized, in which the client only communicates with $2t+1$ random servers to read and recover k columns correctly (e.g., step 2). In the malicious setting, the client needs to read from *all* ℓ servers to verify the consistency of columns and to recover them correctly.