A Multi-server Oblivious Dynamic Searchable Encryption Framework

Thang Hoang [a,*], Attila A. Yavuz [a], F. Betül Durak [b,**], Jorge Guajardo [b]
[a] *Department of Computer Science and Engineering, University of South Florida, Tampa, FL, USA*
E-mails: hoangm@mail.usf.edu, attilaayavuz@usf.edu
[b] *Robert Bosch LLC, Research and Technology Center, Pittsburgh, PA, USA*
E-mails: betul.durak@us.bosch.com, jorge.guajardomerchan@us.bosch.com

Abstract. Data privacy is one of the main concerns for data outsourcing on the cloud. Although standard encryption can provide confidentiality, it prevents the client from searching/retrieving meaningful information on the outsourced data thereby, degrading the benefits of using cloud services. To address this data utilization versus privacy dilemma, Dynamic Searchable Symmetric Encryption (DSSE) has been proposed. DSSE enables encrypted search and update functionality over the encrypted data via a secure index. However, the state-of-the-art DSSE constructions leak information from the access pattern, making them vulnerable against various attacks. While generic Oblivious Random Access Machine (ORAM) can hide the access pattern, it incurs a heavy communication overhead, which was shown costly to be directly used in the DSSE setting.

In this article, by exploiting the multi-cloud infrastructure, we develop a comprehensive Oblivious Distributed DSSE (ODSE) framework that allows oblivious search and updates on the encrypted index with high security and improved efficiency over the use of generic ORAM. Our framework contains a series of ODSE schemes each featuring different levels of performance and security required by various types of real-life applications. ODSE offers desirable security guarantees such as information-theoretic security and robustness in the presence of a malicious adversary. We fully implemented ODSE framework and evaluated its performance in a real cloud environment (Amazon EC2). Our experiments showed that ODSE schemes are $3\times$-$57\times$ faster than using generic ORAMs on a DSSE encrypted index under real network settings.

Keywords: Searchable encryption, Write-only ORAM, Multi-server PIR, Privacy-preserving clouds

1. Introduction

The concept of storage-as-a-service provides a comprehensive storage architecture for companies or individuals to store data on the cloud, thereby reducing the data management and maintenance cost. Despite its usefulness, recent data breach incidents on such systems have shown the importance of preserving the confidentiality of sensitive data stored on the cloud. Although standard encryption (e.g., AES) can preserve data privacy, it also prevents the users from searching or retrieving meaningful information from outsourced data, which invalidates some benefits of using cloud storage services. To address this data utilization *vs.* privacy dilemma, the concept of searchable symmetric encryption (SSE) was introduced [1]. Since then, many SSE schemes have been developed in attempts to offer practical query functionality while, at the same time, preserving user privacy and data confidentiality. In the following sections, we first review the ongoing research on SSE and outline the security limitations of state-of-the-art approaches.

*Corresponding author. E-mail: hoangm@mail.usf.edu.
**Work done while the third author was at EPFL, Lausanne, Switzerland.

1.1. State-of-the-arts and Limitation

SSE. Song et al. were the first to propose the concept of SSE [1]. Later, Curtmola et al. [2] defined indistinguishability under the adaptive chosen keyword attack (IND-CKA2) as a formal security notion for SSE, and presented an IND-CKA2-secure scheme supporting single keyword search. The security is achieved by constructing a secure index ($\mathbf{I}$) representing the relationship between keywords and encrypted files ($\mathcal{F}$), both of which ($\langle \mathbf{I}, \mathcal{F} \rangle$) are outsourced to the cloud. Several refinements based on this index model have been proposed to offer more functionality and query diversity such as ranked query [3] and/or multi-keyword search [4, 5]. The main limitation of these constructions is that they are only static, meaning that they can only perform a search on the encrypted data with no update allowed after the setup. Kamara et al. were among the first to propose Dynamic Searchable Symmetric Encryption (DSSE) [6], which enables both search and update functionalities on encrypted files. After their studies, many DSSE schemes have been proposed, each offering distinct performance, functionality and security trade-offs [6–12].

Information Leakages in SSE and Limitations of Other Approaches. SSE without relying on the encrypted index has been shown to be vulnerable against many attacks [13, 14]. On the other hand, although the encrypted index-based SSE is known to be more secure, it still leaks a lot of information that the adversary can exploit to conduct statistical attacks [15–17]. For instance, when the client performs a search on the encrypted index, the search token in DSSE might reveal the content files to be updated in the future as well as all of the historical updates on files matching with the token. These leakages are defined as forward-privacy and backward-privacy, respectively [18]. Zhang et al. showed that it is possible to learn which keywords have been searched in forward-insecure DSSE schemes through file-injection attacks [17]. Most efficient DSSE schemes [6, 7, 19, 20] do not provide forward- and backward-privacy when searching on $\mathbf{I}$. Although there are some forward- and backward-private DSSE schemes being proposed recently (e.g., [8, 21]), they rely on costly public key operations [22]. More severely, since the search, update and retrieval queries in DSSE are deterministic, all standard DSSE schemes leak *access patterns* on both $\mathbf{I}$ and $\mathcal{F}$. In particular, the client leaks the *file-access pattern* when updating a file or when retrieving a set of files matching with the search query performed on the encrypted index. Similarly, the client leaks the *index-access pattern* when performing the search or update on the encrypted index. Liu et al. [16] demonstrated a practical attack that can determine which keywords being searched by observing the search pattern.

To seal most of the access pattern leakage in DSSE, one can use a *generic*[1] Oblivious Random Access Machine (ORAM) technique [23] to conduct oblivious access on $\mathbf{I}$ and $\mathcal{F}$. Garg et al. [24] proposed TWORAM, which optimizes the use of ORAM to hide *file access patterns* in DSSE[2]. Despite its merits, prior studies such as [7, 25] stated that generic ORAM [23] is expensive to be used in DSSE setting due to its logarithmic client-bandwidth overhead. Although ORAM schemes with a constant bandwidth complexity have been introduced recently [26], they rely on costly cryptographic protocols (i.e., homomorphic encryption [27]), whose performance was shown worse than ORAMs with logarithmic bandwidth overhead [28]. Alternative solutions trying to avoid generic ORAM are either very costly or unable to seal access pattern leakage in DSSE [29, 30].

[1] By generic ORAM, we mean the technique that can hide whether the access is to read or to write as opposed to read-only Private Information Retrieval 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 §8 for clarification).

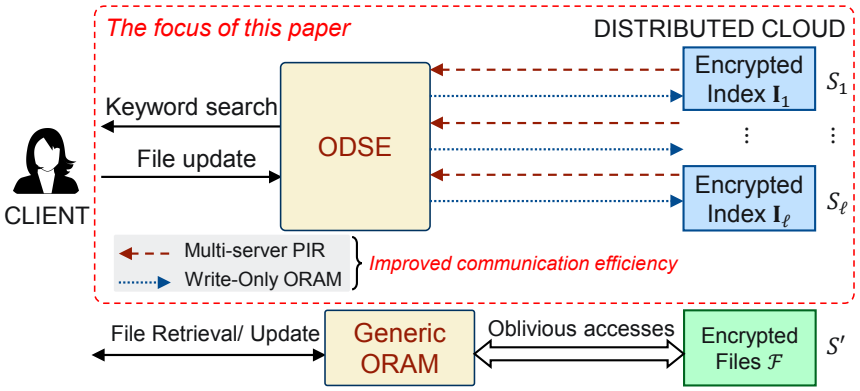


Fig. 1. Our research objective and high-level approach.

1.2. Our Contributions

In DSSE, it is highly desirable to seal access pattern leakages when accessing the encrypted index ($\mathbf{I}$) and encrypted files ($\mathcal{F}$). Since the size of individual files in $\mathcal{F}$ can be arbitrarily large and each search/update query might involve with a different number of files, to the best of our knowledge, generic ORAM seems to be the only option for oblivious access on $\mathcal{F}$. In this paper, we focus more on oblivious access techniques on the index ($\mathbf{I}$) that are more bandwidth-efficient than using generic ORAM (Figure 1). Specifically, we propose ODSE, a comprehensive oblivious encrypted index framework in the multi-server setting with the application to DSSE. The framework contains three ODSE schemes including $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ each offering various performance and security properties as follows.

- **Full obliviousness with information-theoretic security:** ODSE seals information leakages when accessing the encrypted index that might lead into statistical attacks. Our constructions hide the index-access pattern, and therefore provide forward- and backward-privacy and secrecy of the query types (search/update). $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ offers computational security for the encrypted index as well as access operations on it. On the other hand, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ provides information-theoretic statistical security (see §5).
- **Low end-to-end delay:** All ODSE schemes offer low end-to-end-delay, which are $3\times$-$57\times$ faster than using generic ORAM atop the DSSE encrypted index (with optimization [24]) under real network settings (see §8).
- **Robustness against malicious adversary:** In the present work, we provide secure methods not only in the honest-but-curious setting but also in the malicious environment. Our ODSE schemes offer various levels of robustness in the distributed setting. In the semi-honest setting, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ are robust against corrupted servers that do not respond due to system/network failure. All ODSE schemes can be extended to be secure against malicious adversary. Specifically, the extended $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ scheme can detect if there exists any malicious server in the system (but without knowing which server it is). The extended $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ schemes can not only detect which server(s) is malicious, but also be robust against incorrect replies by malicious servers.
- **Full-fledged implementation and open-sourced framework:** We fully implemented all the proposed ODSE schemes, and evaluated their performance on real-cloud infrastructure. To the best of our knowledge, we are among the first to open-source an oblivious access framework for the encrypted index in DSSE (see §8). The code is available at https://github.com/thanghoang/ODSE.

Improvement over the IFIP DBSec'18 Conference Version [31]. This article is the extended version of [31], which includes the following improvements. First, we introduce a new ODSE scheme called $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$, which is a hybrid scheme between $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ originally presented in [31]. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ inherits the best of both worlds, in which it features low search/update delay and robustness in the distributed setting simultaneously. Second, we extended all the proposed ODSE schemes into the malicious-setting, which was only discussed briefly in [31]. Third, we conducted more experiments to evaluate the performance of the new $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ scheme as well as all the extended ODSE schemes in the malicious setting with different number of corrupted servers.

2. Preliminaries and Building Blocks

2.1. Notation

We denote a finite field as $\mathbb{F}_p$ where p is a prime. Operators $||$ and $\oplus$ denote the concatenation and XOR, respectively. $\langle\cdot\rangle_{\mathsf{bin}}$ denotes the binary representation. $[N]$ denotes $\{1,\ldots,N\}$. $\mathbf{u}\cdot\mathbf{v}$ denotes the dot product of two vectors $\mathbf{u}$ and $\mathbf{v}$. $x \xleftarrow{\$} \mathcal{S}$ denotes that x is randomly and uniformly selected from $\mathcal{S}$. Given I as a row/column of a matrix, $I[i]$ denotes accessing the i-th component of I. Given a matrix $\mathbf{I}$, $\mathbf{I}[*, j\ldots j']$ denotes accessing columns j to j' of $\mathbf{I}$. $\mathbf{I}[i,*]$ and $\mathbf{I}[*,j]$ denotes accessing the entire row i and column j of $\mathbf{I}$, respectively. Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA symmetric encryption [32]: $\kappa \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)$ generating key with security parameter λ; $C \leftarrow \mathcal{E}.\mathsf{Enc}_\kappa(M,c)$ encrypting plaintext M with key κ and counter c; $M \leftarrow \mathcal{E}.\mathsf{Dec}_\kappa(C,c)$ decrypting ciphertext C with key κ and counter c.

2.2. Shamir Secret Sharing

We present (t,ℓ)-threshold Shamir Secret Sharing (SSS) scheme [33] in Figure 2. Given a secret $b \in \mathbb{F}_p$ to be shared, the dealer generates a random t-degree polynomial f and evaluates $f(x_i)$ for each party $\mathcal{P}_i \in \{\mathcal{P}_1,\ldots,\mathcal{P}_\ell\}$, at point x_i which is a non-zero element of $\mathbb{F}_p$. x_i is used to identify party $\mathcal{P}_i$ (SSS.CreateShare algorithm). We denote the share for $\mathcal{P}_i$ as $[\![b]\!]_i$ for $1 \leqslant i \leqslant \ell$. The secret can be reconstructed by combining at least $t+1$ correct shares via Lagrange interpolation (SSS.Recover algorithm).

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

2.3. Private Information Retrieval

Private Information Retrieval (PIR) technique enables private retrieval of a data item from a (*unencrypted*) public database server. PIR in the distributed setting is defined as follows.

Definition 1 (multi-server PIR [34, 35]). *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_j in $\mathbf{b}$ to be retrieved, the client creates queries $(\rho_1,\ldots,\rho_\ell) \leftarrow \mathsf{PIR.CreateQuery}(j)$ and distributes ρ_i to server $\mathcal{S}_i$ for each $i \in \{1\ldots\ell\}$. Each server $\mathcal{S}_i$ responds with an answer $r_i \leftarrow \mathsf{PIR.Retrieve}(\rho_i, \mathbf{b})$.*

$(\llbracket b \rrbracket_1, \ldots, \llbracket b \rrbracket_\ell) \leftarrow \mathsf{SSS.CreateShare}(b, t)$: Create ℓ shares of value b

1: $\{a_1, \ldots, a_t\} \xleftarrow{\$} \mathbb{F}_p$ `# Creates the coefficients of the polynomial`
2: $\{x_1, \ldots, x_\ell\} \leftarrow \mathbb{F}_p - \{0\}$ `# Pick the evaluation points of the polynomial`
3: **for** $i = 1, \ldots, \ell$ **do**
4: $\quad \llbracket b \rrbracket_i \leftarrow b + \sum_{u=1}^{t} a_u \cdot x_i^u$ `# Evaluate the polynomial at each point`
5: **return** $\mathcal{B} = (\llbracket b \rrbracket_1, \ldots, \llbracket b \rrbracket_\ell)$

$b \leftarrow \mathsf{SSS.Recover}(\mathcal{B}, t)$: Recover the value b from its shares

1: Randomly select $t+1$ shares $\{\llbracket b \rrbracket_{i_j}\}_{j=1}^{t+1}$ in $\mathcal{B}$, where $i_j \in \{1, \ldots \ell\}$
2: $g(x) \leftarrow \mathsf{LagrangeInterpolation}\big(\{(x_{i_j}, \llbracket b \rrbracket_{i_j})\}_{j=1}^{t+1}\big)$
3: **return** b, where $b \leftarrow g(0)$

Fig. 2. Shamir Secret Sharing (SSS) scheme [33].

Upon receiving $\mathcal{R} = \{r_1, \ldots, r_\ell\}$ *answers, the client computes the value of item* b *by invoking the reconstruction algorithm* $b \leftarrow \mathsf{PIR.Reconstruct}(\mathcal{R})$.

A multi-server PIR is correct *if the client can obtain the correct value of* b *from* ℓ *answers via* PIR.Reconstruct *algorithm with the probability 1. A multi-server PIR is* t-private *if* $\forall j, j' \in \{1, \ldots, n\}$, $\forall \mathcal{L} \subseteq \{1, \ldots, \ell\}$ *s.t.* $|\mathcal{L}| \leqslant t$, *the probability distributions of* $\big\{\rho_{j \in \mathcal{L}} : (\rho_1, \ldots, \rho_\ell) \leftarrow \mathsf{PIR.CreateQuery}(j)\big\}$ *and* $\big\{\rho'_{j' \in \mathcal{L}} : (\rho'_1, \ldots, \rho'_\ell) \leftarrow \mathsf{PIR.CreateQuery}(j')\big\}$ *are identical.*

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

- **XOR-based PIR [36].** It relies on XOR trick to perform the private retrieval, in which the database **b** contains n items b_i, each being interpreted as a m-bit string (Figure 3).
- **SSS-based PIR [34, 35].** It relies on SSS to improve the robustness of multi-server PIR, in which the database **b** contains n items b_i, each being interpreted as an element of $\mathbb{F}_p$ (Figure 4).

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

1: Initialize binary string $e \leftarrow 0^n$ and set $e[j] \leftarrow 1$
2: **for** $i = 1, \ldots, \ell - 1$ **do**
3: $\quad \rho_i \xleftarrow{\$} \{0,1\}^n$
4: $\rho_\ell \leftarrow \rho_1 \oplus \cdots \oplus \rho_{\ell-1} \oplus e$
5: **return** $(\rho_1, \ldots, \rho_\ell)$

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

1: Parse **b** as $(b_1, \ldots b_n)$
2: Initialize $r_i \leftarrow \{0\}^m$
3: **for** $j = 1, \ldots, n$ **do**
4: $\quad$ **if** $\rho_i[j] = 1$ **then**
5: $\quad\quad r_i \leftarrow r_i \oplus b_j$
6: **return** r_i

$b \leftarrow \mathsf{PIR}^{\mathsf{xor}}\mathsf{.Reconstruct}(\mathcal{R})$: Reconstruct the item

1: Parse $\mathcal{R}$ as $\{r_1, \ldots, r_\ell\}$
2: $b \leftarrow r_1 \oplus \cdots \oplus r_\ell$
3: **return** b

Fig. 3. XOR-based PIR [36].

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

$b \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\mathcal{B}, t)$: Recover the retrieved item from the set of answers $\mathcal{B}$
1: Parse $\mathcal{B}$ as $\{\llbracket b \rrbracket_1, \ldots, \llbracket b \rrbracket_\ell\}$
2: $b \leftarrow \mathsf{SSS.Recover}(\mathcal{B}, t)$
3: **return** b

Fig. 4. SSS-based PIR [34, 35].

Write-Only ORAM. ORAM allows the user to hide access patterns when accessing their encrypted data on the cloud. In contrast to generic ORAM where both read and write operations are hidden, Blass et al. [37] proposed a Write-Only ORAM scheme, which only hides the write pattern in the context of hidden volume encryption. Intuitively, $2n$ memory slots are used to store n blocks, each assigned to a distinct slot and a position map is maintained to keep track of block's location. Given a block to be rewritten, the client reads $\mathcal{O}(\lambda)$ slots chosen uniformly at random and writes the block to a dummy slot among $\mathcal{O}(\lambda)$ slots. Data in all slots are encrypted to hide which slot is updated. By selecting λ sufficiently large, one can achieve a negligible failure probability, which might occur when all λ slots are non-dummy. It is also possible to select a small λ. In this case, the client maintains a stash component S of size $\mathcal{O}(\log n)$ to temporarily store blocks that cannot be rewritten when all read slots are full.

3. System and Security Models

In this section, we present the system and security models of our framework.

3.1. 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. The encrypted files are stored on a separate server different from $\boldsymbol{\mathcal{S}}$ (as in [29]), which can be obliviously accessed via a generic ORAM scheme [23, 38]. In this paper, we focus only on oblivious access on distributed encrypted index $\mathcal{I}$ on $\boldsymbol{\mathcal{S}}$. We present the definition of ODSE as follows.

Definition 2. *An Oblivious Distributed Dynamic Searchable Symmetric Encryption (ODSE) scheme is a tuple of one algorithm and two protocols* $\mathit{ODSE} = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$*, where the input and the output for the client and the servers are separated with semicolon such that:*

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

(2) $(\mathcal{R}; \perp) \leftarrow \mathsf{Search}(w, \sigma; \mathcal{I})$: *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 file identifiers, in which w appears.*

(3) $(\sigma'; \mathcal{I}') \leftarrow \mathsf{Update}(f_{id}, \sigma; \mathcal{I})$: *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.*

3.2. Security Model

In our system, the client is trusted and the set of servers $\boldsymbol{\mathcal{S}}$ are untrusted. We first consider the servers to be *semi-honest*, meaning that they follow the protocol faithfully, but can record the protocol transcripts to learn information regarding the client's access pattern. Later, we show that our framework can be extended to be secure against *malicious* servers that can tamper with the input data to compromise the correctness and the security of the system (§6). We allow up to $t < \ell$ (privacy parameter) servers among $\boldsymbol{\mathcal{S}}$ to be colluding, meaning that they can share their own recorded protocol transcripts with each other. Formally, the security of ODSE in the semi-honest setting can be defined as follows.

Definition 3 (ODSE security w. r. t. semi-honest adversary). *Let $\vec{o} = (\mathsf{op}_1, \ldots, \mathsf{op}_q)$ be an operation sequence, where $\mathsf{op}_i \in \{\mathsf{Search}(w, \sigma; \mathcal{I}), \mathsf{Update}(f_{id}, \sigma; \mathcal{I})\}$, 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 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 negligible probability.*
t-security*: An ODSE is* t-secure *if $\forall \mathcal{L} \subseteq \{1, \ldots, \ell\}$ such that $|\mathcal{L}| \leqslant t$, for any two operation sequences $\vec{o}_1$ and $\vec{o}_2$ where $|\vec{o}_1| = |\vec{o}_2|$, the views $\{\boldsymbol{ODSE}_{l \in \mathcal{L}}(\vec{o}_1)\}$ and $\{\boldsymbol{ODSE}_{l \in \mathcal{L}}(\vec{o}_2)\}$ observed by a coalition of up to t servers are computationally indistinguishable.*

ODSE operation obliviousness. By Definition 2, keyword search and file update are the two main operations in searchable encryption. Given that these operations might incur different procedures, we can trigger both search and update protocols for any actual action to achieve the operation obliviousness according to Definition 3. In this case, the server can guess (at best) with a probability of $\frac{1}{2}$ what operation the client is performing "in real" i.e. either search or update.

4. The Proposed (Semi-Honest) ODSE Schemes

Intuition. In DSSE, keyword search and file update on $\mathbf{I}$ are read-only and write-only operations, respectively. This property permits us to leverage specific bandwidth-efficient oblivious access techniques for each operation such as multi-server PIR (for search) and Write-Only ORAM (for update) rather than using a generic ORAM. The second requirement is to identify a suitable data structure for $\mathbf{I}$ so that these bandwidth-efficient techniques can be adapted. In DSSE, forward index and inverted index are the ideal choices for the file update and keyword search operations, respectively as proposed in [19]. However, performing search and update on two isolated indexes will lead to inconsistency. The server might perform a synchronization to make two indices consistent; however, this will leak significant information

Table 1
ODSE Symbols and Notations.

Symbol	Description
N, M	Maximum number of files and keywords in DB.
$\mathbf{I}$	Incidence Matrix Index
N'	Number of $(\lceil \log_2 p \rceil - 1)$-bit blocks (i.e., $N' = \lceil \frac{N}{\lceil \log_2 p \rceil - 1} \rceil$).
T_f, T_w	Static hash tables for files and keywords.
$\mathcal{D}$	Set of dummy (empty) columns
S	Stash to (temporarily) store column data
$\mathbf{c}$	Column counter vector

regarding the client query and file content. Therefore, to avoid this problem, it is mandatory to seek a data structure, where both search index and update index can be integrated together. Fortunately, this can be achieved by harnessing a two-dimensional index (i.e., matrix), which allows keyword search and file update to be performed in two separate dimensions without creating any inconsistency at their intersections. 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.

In the following, we first describe the data structures used in ODSE framework, and then present semi-honest ODSE schemes in details. We analyze the security of ODSE schemes and present their extension into malicious setting in §5 and §6, respectively.

4.1. ODSE Data Structures

Our index to be stored at the server(s) is an incidence matrix ($\mathbf{I}$), where each cell ($\mathbf{I}[i, j] \in \{0, 1\}$) represents the relationship between the keyword indexed at row i and the file indexed at column j. So, each row of $\mathbf{I}$ represents the search result of a keyword and each column represents the content (i.e., keywords) of a file. Since we use Write-Only ORAM for file update, the number of columns in $\mathbf{I}$ are doubled to the maximum number of files that can be stored in the outsourced database. In other words, given N distinct files and M unique keywords in the database, our index is of size $M \times 2N$. At the client side, we leverage two position maps T_w, T_f to keep track of 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 $\mathsf{value} \leftarrow T[\mathsf{key}]$ is the (row/column) index of key in $\mathbf{I}$. Due to Write-Only ORAM, the client maintains a stash component S to temporarily store columns that might not be written back during the update due to the overflow.

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 the setup algorithm in ODSE as well as its oblivious search and update protocols as follows.

Setup. Figure 5 presents setup algorithm to construct the encrypted index in ODSE. Specifically, it first initializes an unencrypted incidence matrix ($\mathbf{I}'$) of size $M \times 2N$ (line 1), and generates a master key to be used for generating row keys to encrypt each row of $\mathbf{I}'$ (line 3). It extracts unique keywords from input files (line 4), assigns each keyword and file into a row and column of $\mathbf{I}'$ selected randomly (lines 6, 9), and then sets the value for each cell of $\mathbf{I}'$ corresponding to the relationship between keywords and files (line 10). Finally, the algorithm generates a distinct key for each row of $\mathbf{I}'$ by the master key (line 14),

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

1: $\mathbf{I}'[*,*] \leftarrow 0$, initialize counter $\mathbf{c} \leftarrow (c_1, \ldots, c_{2N})$ where $c_i \leftarrow 1$ for each $i \in \{1, \ldots, 2N\}$
2: Let Π and Π' be a random permutation on $\{1, \ldots, 2N\}$ and $\{1, \ldots, M\}$ respectively
3: $\kappa \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)$
4: Extract keywords $(w_1, \ldots, w_m)$ from files $\mathcal{F} = \{f_{id_1}, \ldots, f_{id_n}\}$
5: **for** $i = 1, \ldots, m$ **do**
6: $\quad T_w[w_i] \leftarrow \Pi'(i)$
7: $\quad$ **for** $j = 1, \ldots, n$ **do**
8: $\quad\quad T_f[id_j] \leftarrow \Pi(j)$
9: $\quad\quad$ **if** w_i appears in f_{id_j} **then**
10: $\quad\quad\quad \mathbf{I}'[x_i, y_j] \leftarrow 1$, where $x_i \leftarrow T_w[w_i]$, $y_j \leftarrow T_f[id_j]$
11: **for** $i = 1, \ldots, M$ **do**
12: $\quad \tau_i \leftarrow \mathsf{KDF}_\kappa(i)$ `# Compute key for each row`
13: $\quad$ **for** $j = 1, \ldots, 2N$ **do**
14: $\quad\quad \mathbf{I}[i, j] \leftarrow \mathcal{E}.\mathsf{Enc}_{\tau_i}(\mathbf{I}'[i, j], j||c_j)$ `# Ciphertext I[i, j] is one-bit long`
15: Let $\mathcal{I}$ contain ℓ copies of $\mathbf{I}$ and $\sigma \leftarrow (\kappa, T_w, T_f, \mathbf{c})$
16: **return** $(\sigma, \mathcal{I})$

Fig. 5. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ setup algorithm.

and encrypts each cell of $\mathbf{I}'$ by a distinct pair of row key and column counter resulting in an encrypted index $\mathbf{I}$ (line 14). We encrypt the index bit-by-bit and the resulting ciphertext of each input bit is also one bit long. This can be implemented by, for example, AES with CTR mode, where we generate a 128-bit pseudorandom stream key by the master row key (τ_i) and the column counter ($j||c_j$), but only XOR the plaintext bit with the most significant bit of the stream key. To this end, the client sends a replica of $\mathbf{I}$ to ℓ servers and keeps some information (i.e., $\kappa, T_w, T_f, \mathbf{c}$) private.

Search. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ harnesses XOR-based PIR on the row dimension of $\mathbf{I}$ to conduct the oblivious keyword search as shown in Figure 6. The client first looks up the keyword position map to get the row index of the searched keyword (line 1). The client then creates XOR-PIR queries (line 2) and sends them to corresponding servers, each answering the client with the output of the PIR retrieval algorithm (line 4). Notice that the data is IND-CPA encrypted rather than being public as in the standard PIR model. Therefore, after recovering the row from the PIR retrieval (line 6), the client generates the row key (line 7) and then decrypts the row to obtain the search result (line 9).

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

Client:
1: $i \leftarrow T_w[w]$
2: $(\rho_1, \ldots, \rho_\ell) \leftarrow \mathsf{PIR}^{\mathsf{xor}}.\mathsf{CreateQuery}(i)$
3: Send ρ_l to $\mathcal{S}_l$ for $l \in \{1, \ldots, \ell\}$
Server: each $\mathcal{S}_l \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving ρ_l **do**
4: $\hat{I}_l \leftarrow \mathsf{PIR}^{\mathsf{xor}}.\mathsf{Retrieve}(\rho_l, \mathbf{I}_l)$
5: Send $\hat{I}_l$ to the client
Client: On receive $(\hat{I}_1, \ldots, \hat{I}_\ell)$ from ℓ servers
6: $\mathbf{I}[i, *] \leftarrow \mathsf{PIR}^{\mathsf{xor}}.\mathsf{Reconstruct}(\hat{I}_1, \ldots, \hat{I}_\ell)$
7: $\tau_i \leftarrow \mathsf{KDF}_\kappa(i)$
8: **for** $j = 1, \ldots, 2N$ **do**
9: $\quad \mathbf{I}'[i, j] \leftarrow \mathcal{E}.\mathsf{Dec}_{\tau_i}(\mathbf{I}[i, j], j||c_j)$
10: Let $\mathcal{J} := \{j : (\mathbf{I}'[i, j] = 1)$ and $((j$ is not dummy) or $(\mathbf{I}'[i, j] \in S))\}$
11: **return** $(\mathcal{R}; \perp)$, where $\mathcal{R}$ contains file IDs at column indexes in $\mathcal{J}$

Fig. 6. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ search protocol.

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

Client:

1: Initialize a column $\hat{I}[i] \leftarrow 0$ for $i \in \{1, \ldots, 2N\}$
2: **for** each keyword $w_i \in f_{id}$ **do**
3: $\quad \hat{I}[x_i] \leftarrow 1$, where $x_i \leftarrow T_w[w_i]$
4: $S \leftarrow S \cup \{(id, \hat{I})\}$ and $T_f[id] \leftarrow 0$ `# Put updated column to stash`
5: Let $\mathcal{J}$ contain λ random-selected column indexes, send $\mathcal{J}$ to an arbitrary server $\mathcal{S}_l$

Server $\mathcal{S}_l$: On receive $\mathcal{J}$ **do**

6: Send $\{\mathbf{I}_l[*, j]\}_{j \in \mathcal{J}}$ to the client

Client: On receive $\{\mathbf{I}_l[*, j]\}_{j \in \mathcal{J}}$ **do**

7: **for** $i = 1, \ldots, M$ **do**
8: $\quad \tau_i \leftarrow \mathsf{KDF}_\kappa(i)$
9: $\quad$ **for** each index $j \in \mathcal{J}$ **do** `# Decrypt columns`
10: $\quad\quad \mathbf{I}'[i, j] \leftarrow \mathcal{E}.\mathsf{Dec}_{\tau_i}(\mathbf{I}_l[i, j], j || c_j)$
11: **for** each dummy index $\hat{j} \in \mathcal{J}$ **do** `# Put columns from stash`
12: $\quad \mathbf{I}'[*, \hat{j}] \leftarrow \hat{I}$ and $T_f[id] \leftarrow \hat{j}$, where $(id, \hat{I})$ is picked from S
13: **for** each index $j \in \mathcal{J}$ **do** `# Re-encrypt columns`
14: $\quad c_j \leftarrow c_j + 1$
15: $\quad$ **for** $i = 1, \ldots, M$ **do**
16: $\quad\quad \hat{\mathbf{I}}[i, j] \leftarrow \mathcal{E}.\mathsf{Enc}_{\tau_i}(\mathbf{I}'[i, j], j || c_j)$
17: Send $\{\hat{\mathbf{I}}[*, j]\}_{j \in \mathcal{J}}$ to ℓ servers

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

18: **for** each $j \in \mathcal{J}$ **do**
19: $\quad \mathbf{I}_l[*, j] \leftarrow \hat{\mathbf{I}}[*, j]$
20: **return** $(\sigma'; \mathcal{I}')$ where $\mathcal{I}'$ is $\mathbf{I}_l$ updated at ℓ servers, and σ' is updated client state

Fig. 7. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ update protocol.

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, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ applies Write-Only ORAM mechanism on the column dimension of $\mathbf{I}$ to update keyword-file pairs in f_{id} as shown in Figure 7. The client creates a new column representing the relationship between the updated file and keywords in the database (lines 2-3), and stores it in the stash (line 4). The client then randomly selects λ column indexes and requests an arbitrary server to transmit the corresponding columns of $\mathbf{I}$ (lines 5-6). The client generates row keys and decrypts λ columns (lines 7-10). The client overwrites dummy columns among λ columns with columns stored in the stash (lines 11-12). Finally, the client re-encrypts λ columns and sends them to ℓ servers (lines 18-20).

4.3. ODSE^{wo}_{ro}: Robust ODSE

The described $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ scheme requires all ℓ servers in the system to answer the client. If one server does not reply due to system/network failure, the correctness of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ will not hold anymore. We propose $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$, an ODSE scheme that can achieve the robustness against unresponsive servers. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ harnesses the t-out-of-ℓ property of SSS, which allows to maintain the correctness given that some servers (i.e., up to $\ell - t - 1$) do not answer. We define the setup algorithm along with the oblivious search and update protocols in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ scheme as follows.

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

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

Fig. 8. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ setup algorithm.

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

Client:

1: $i \leftarrow T_w[w]$
2: $(\llbracket \mathbf{e} \rrbracket_1,\ldots,\llbracket \mathbf{e} \rrbracket_\ell) \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{CreateQuery}(i)$
3: Send $\llbracket \mathbf{e} \rrbracket_l$, to $\mathcal{S}_l$ for $l \in \{1,\ldots,\ell\}$

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

4: **for** $j = 1\ldots,2N'$ **do**
5: **for** $k = 1,\ldots,M$ **do**
6: $\langle c^{(l)}_{jk} \rangle_{\mathsf{bin}} \leftarrow \mathbf{I}_l[k,(j-1)\cdot\lfloor \log_2 p \rfloor + 1\ldots j\cdot\lfloor \log_2 p \rfloor]$ `# j^th batch of k^th row`
7: $\mathbf{c}^{(l)}_j \leftarrow (c^{(l)}_{j1},\ldots,c^{(l)}_{jM})$
8: $\llbracket b_j \rrbracket_l \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Retrieve}(\llbracket \mathbf{e} \rrbracket_l, \mathbf{c}^{(l)}_j)$
9: Send $(\llbracket b_1 \rrbracket_l,\ldots,\llbracket b_{2N'} \rrbracket_l)$ to the client

Client: On receive $\{\mathcal{B}_j = \{\llbracket b_j \rrbracket_1,\ldots,\llbracket b_j \rrbracket_\ell\}\}_{j=1}^{2N'}$ from ℓ servers

10: **for** $j = 1\ldots,2N'$ **do**
11: $b_j \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\mathcal{B}_j, t)$
12: $\mathbf{I}[i,*] \leftarrow \langle b_1 \rangle_{\mathsf{bin}} || \ldots || \langle b_{2N'} \rangle_{\mathsf{bin}}$
13: $\tau_i \leftarrow \mathsf{KDF}_\kappa(i)$
14: **for** $j = 1,\ldots,2N$ **do**
15: $\mathbf{I}'[i,j] \leftarrow \mathsf{Dec}_{\tau_i}(\mathbf{I}[i,j], j||c_j)$
16: Let $\mathcal{J} := \{j : (\mathbf{I}'[i,j] = 1)$ and $((j$ is not dummy) or $(\mathbf{I}'[i,j] \in S))\}$
17: **return** $(\mathcal{R};\perp)$, where $\mathcal{R}$ contains file IDs at column indices in $\mathcal{J}$

Fig. 9. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ search protocol.

Setup. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ works over the index encrypted with IND-CPA encryption. Therefore, the setup algorithm of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ is identical to that of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ scheme as shown in Figure 8.

Search. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ harnesses SSS-based PIR protocol on the row dimension of $\mathbf{I}$ to conduct keyword search as shown in Figure 9. Specifically, the client first retrieves the row index of the searched keyword from the keyword position map (line 1). The client then creates SSS-based PIR queries (line 2) and sends to corresponding servers, each replying with the output of the SSS-based PIR retrieval algorithm. Notice that the SSS-based PIR retrieval algorithm performs the dot product between the client query and the database input via scalar multiplication and additive homomorphic properties of SSS. This requires the database input to be elements in $\mathbb{F}_p$. Since each row in $\mathbf{I}$ is a uniformly random binary string of length $2N$ due to IND-CPA encryption, the servers split each row of $\mathbf{I}$ into $2N'$ chunks (c_k) with the equal size such that $|c_k| < \log_2 p$ (line 6). The dot product is performed iteratively between the search query and divided chunks from all rows of $\mathbf{I}$ (lines 7-8). After receiving answers from ℓ servers, the client recovers all chunks of the searched row (lines 10-12) and finally, decrypts the row to obtain the search result (lines 13-17).

Update. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ harnesses Write-Only ORAM mechanism on the column dimension of $\mathbf{I}$ to perform file update. Since the index $\mathbf{I}$ in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ is identical to $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, the update protocol of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ is also identical to that of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ (Figure 10).

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

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

Fig. 10. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ update protocol.

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

$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ scheme relies on IND-CPA encryption for the encrypted index so that it only offers (at most) computational security. In this section, we introduce $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$, an ODSE scheme that can achieve the highest level of security (i.e., information-theoretic) for the index as well as any operations (search and update) on it. The main idea is to share the index with SSS, and harness SSS-based PIR to conduct private search. We describe the algorithms of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ as follows.

Setup. Figure 11 presents the setup algorithm to construct the distributed index in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$. Specifically, it first constructs an (unencrypted) index ($\mathbf{I}'$) representing keyword-file relationships as in other ODSE schemes. Instead of encrypting $\mathbf{I}'$ with an IND-CPA encryption scheme, it creates the shares of $\mathbf{I}'$ with SSS and distributes them to corresponding servers. As discussed above, SSS operates over elements in $\mathbb{F}_p$. Therefore, it is required to split each row of $\mathbf{I}'$ into $\lfloor \log_2 p \rfloor$-bit chunks (line 4), and compute SSS share for each chunk (line 5). Therefore, the "encrypted" index in ODSE contains ℓ SSS-shares of $\mathbf{I}'$ for ℓ servers, each being a matrix $\mathbf{I}_l$ of size $M \times 2N'$, where $\mathbf{I}_l[i,j] \in \mathbb{F}_p$ and $N' = N/\lfloor \log_2 p \rfloor$. To this end, the client sends $\mathbf{I}_l$ to server $\mathcal{S}_l$ and keep position maps (i.e., T_w, T_f) private.

Search. Similar to $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ harnesses the SSS-based PIR protocol on the row dimension of $\mathbf{I}$ to conduct the keyword search as presented in Figure 12. Generally speaking, the client gets the row index to be searched from the keyword position map, creates SSS-based PIR queries and send them to the corresponding servers, each replying with the outputs of the SSS-based PIR retrieval algorithm (lines 1-6). Notice that since the index stored on $\mathcal{S}_l$ is a share matrix, each dot product computation in the SSS-

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

1: $(\mathbf{I}', T_w, T_f) \leftarrow$ Execute lines 2–10 in Figure 5
2: **for** $i = 1, \ldots, M$ **do**
3: **for** $j = 1, \ldots, 2N'$ **do**
4: $\langle b_{ij} \rangle_{\mathsf{bin}} \leftarrow \mathbf{I}[i, (j-1) \cdot \lfloor \log_2 p \rfloor + 1 \ldots j \cdot \lfloor \log_2 p \rfloor]$ `# Pack each row into batches of size` $\lfloor \log_2 p \rfloor$
5: $(\mathbf{I}_1[i,j], \ldots, \mathbf{I}_\ell[i,j]) \leftarrow \mathsf{SSS.CreateShare}(\langle b_{ij} \rangle_{\mathsf{bin}}, t)$
6: **return** $(\sigma, \mathcal{I})$, where $\mathcal{I} \leftarrow \{\mathbf{I}_1, \ldots, \mathbf{I}_\ell\}$ and $\sigma \leftarrow (T_w, T_f)$

Fig. 11. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ setup algorithm.

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

Client:
$i \leftarrow T_w[w]$
$(\llbracket \mathbf{e} \rrbracket_1, \ldots, \llbracket \mathbf{e} \rrbracket_\ell) \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{CreateQuery}(i)$
Send $\llbracket \mathbf{e} \rrbracket_l$, to $\mathcal{S}_l$ for $l \in \{1, \ldots, \ell\}$
Server: each $\mathcal{S}_l \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $\llbracket \mathbf{e} \rrbracket_l$ **do**
for $j = 1 \ldots, 2N'$ **do**
$\llbracket b_j \rrbracket_l \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Retrieve}(\llbracket \mathbf{e} \rrbracket_l, \mathbf{I}_l[*, j])$
Send $(\llbracket b_1 \rrbracket_l, \ldots, \llbracket b_{2N'} \rrbracket_l)$ to the client
Client: On receive $\{\mathcal{B}_j = \{\llbracket b_j \rrbracket_1, \ldots, \llbracket b_j \rrbracket_\ell\}\}_{j=1}^{2N'}$ from ℓ servers
for $j = 1 \ldots, 2N'$ **do**
$b_j \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\mathcal{B}_j, 2t)$
$\mathbf{I}'[i, *] \leftarrow \langle b_1 \rangle_{\mathsf{bin}} || \ldots || \langle b_{2N'} \rangle_{\mathsf{bin}}$
Let $\mathcal{J} := \{ j : (\mathbf{I}'[i,j] = 1)$ and $((j$ is not dummy) or $(\mathbf{I}'[i,j] \in S))\}$
return $(\mathcal{R}; \perp)$, where $\mathcal{R}$ contains file IDs at column indices in $\mathcal{J}$

Fig. 12. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ search protocol.

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

Client:

1: Initialize a column $\hat{I}[i] \leftarrow 0$ for $i = 1, \ldots, 2N$
2: **for** each keyword $w_i \in f_{id}$ **do**
3: $\quad \hat{I}[x_i] \leftarrow 1$, where $x_i \leftarrow T_w[w_i]$
4: $S \leftarrow S \cup \{(id, \hat{I})\}$ and $T_f[id] \leftarrow 0$
5: Let $\mathcal{J}$ contain λ random-selected column indexes, send $\mathcal{J}$ to $(t+1)$ arbitrary servers $\mathcal{S}_{l_1}, \ldots, \mathcal{S}_{l_{t+1}}$

Server: each $\mathcal{S}_l \in \{\mathcal{S}_{l_1}, \ldots, \mathcal{S}_{l_{t+1}}\}$ receiving $\mathcal{J}$ **do**

6: Send $\{\mathbf{I}_l[*, j]\}_{j \in \mathcal{J}}$ to the client

Client: On receive $\{\mathcal{B}_{ij} = \{\mathbf{I}_{l_1}[i, j], \ldots, \mathbf{I}_{l_{t+1}}[i, j]\}\}_{j \in \mathcal{J}, i \in [M]}$ **do**

7: **for** $i = 1, \ldots, M$ **do**
8: $\quad$ **for** each index $j \in \mathcal{J}$ **do**
9: $\quad\quad b_{ij} \leftarrow \mathsf{SSS.Recover}(\mathcal{B}_{ij}, t)$
10: $\quad\quad \mathbf{I}'[i, j \cdot \lfloor \log_2 p \rfloor + 1, \ldots (j+1) \cdot \lfloor \log_2 p \rfloor] \leftarrow \langle b_{ij} \rangle_{\mathsf{bin}}$
11: **for** each dummy column $\mathbf{I}'[*, \hat{j}]$ **do**
12: $\quad \mathbf{I}'[*, \hat{j}] \leftarrow \hat{I}$ and $T_f[id] \leftarrow \hat{j}$, where $(id, \hat{I})$ is picked from S
13: **for** each index $j \in \mathcal{J}$ **do**
14: $\quad$ **for** $i = 1 \ldots, M$ **do**
15: $\quad\quad \langle b'_{ij} \rangle_{\mathsf{bin}} \leftarrow \mathbf{I}'[i, j \cdot \lfloor \log_2 p \rfloor + 1, \ldots, (j+1) \cdot \lfloor \log_2 p \rfloor]$
16: $\quad\quad (\hat{\mathbf{I}}_1[i, j], \ldots, \hat{\mathbf{I}}_\ell[i, j]) \leftarrow \mathsf{SSS.CreateShare}(b'_{ij}, t)$
17: Send $\{\hat{\mathbf{I}}_l[*, j]\}_{j \in \mathcal{J}}$ to $\mathcal{S}_l$ for $l \in \{1, \ldots, \ell\}$

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

18: **for** each $j \in \mathcal{J}$ **do**
19: $\quad \mathbf{I}_l[*, j] \leftarrow \hat{\mathbf{I}}_l[*, j]$
20: **return** $(\sigma'; \mathcal{I}')$ where $\mathcal{I}'$ is $\mathbf{I}_l$ updated at each server $\mathcal{S}_l$ and σ' is updated client state

Fig. 13. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ update protocol.

based PIR retrieval algorithm will result in a share represented by a $2t$-degree polynomial. Therefore, the client needs to call the SSS-based recover algorithm with the privacy parameter of $2t$ (*vs.* t as in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$) to obtain the correct search result (line 8).

Update. Similar to other ODSE schemes, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ harnesses Write-Only ORAM mechanism on the column dimension of the index for the oblivious file update as outlined in Figure 13. Specifically, the client creates a column representing the relationship between the updated file and keywords in the database, and temporarily stores it in the stash (lines 1-4). In $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$, each column of the share index $\mathbf{I}_l$ on $\mathcal{S}_l$ actually contains the share of $\lfloor \log_2 p \rfloor$ columns of the unencrypted index $\mathbf{I}'$. Therefore, it suffices to read $\lambda' = \lceil \frac{\lambda}{\lfloor \log_2 p \rfloor} \rceil$ random columns of $\mathbf{I}_l$ from $t+1$ arbitrary servers to reconstruct λ columns of $\mathbf{I}'$ (lines 5-10). The update is similar to other ODSE schemes, in which the client aggressively over-writes dummy columns of $\mathbf{I}'$ with columns stored in the stash (lines 11- 12). Finally, the client creates new SSS shares for the retrieved columns (lines 13-16) and writes them back to ℓ servers (lines 18-20).

5. Security Analysis

Remark 1. One might observe that search and update operations in ODSE schemes are performed on the row dimension and the column dimension of the encrypted index, respectively. This access structure might enable the adversary to learn whether the operation is search or update, even though each operation is secure. Therefore, to achieve security as in Definition 3, where the query type should also be

hidden, we can trigger both search and update protocols (one of them is the dummy operation) regardless of whether the intended action is search or update.

We argue the security of our proposed schemes as follows.

Theorem 1. ODSE^{wo}_{xor} *scheme is computationally* $(\ell-1)$*-secure by Definition 3.*

Proof. (Sketch) *(i) Oblivious Search*: ODSE^{wo}_{xor} leverages XOR-based PIR and therefore, achieves $(\ell-1)$-privacy for keyword search as proven in [36]. *(ii) Oblivious Update*: ODSE^{wo}_{xor} employs Write-Only ORAM which achieves negligible write failure probability and therefore, it offers the statistical security without counting the encryption. The index in ODSE^{wo}_{xor} is IND-CPA encrypted, which offers computational security. Therefore in general, the update access pattern of ODSE^{wo}_{xor} scheme is computationally indistinguishable. ODSE^{wo}_{xor} performs Write-Only ORAM with an identical procedure on ℓ servers (e.g., the indexes of accessed columns are the same in ℓ servers), and therefore, the server coalition does not affect the update privacy of ODSE^{wo}_{xor}. *(iii) ODSE Security:* By Remark 1, ODSE^{wo}_{xor} performs both search and update regardless of the actual operation. As analyzed, search is $(\ell-1)$-private and update pattern is computationally secure. Therefore, ODSE^{wo}_{xor} achieves computational $(\ell-1)$-security by Definition 3. □

Theorem 2. ODSE^{wo}_{ro} *scheme is computationally t-secure by Definition 3.*

Proof. (Sketch) *(i) Oblivious Search*: ODSE^{wo}_{ro} leverages a SSS-based PIR protocol and therefore, achieves t-privacy for keyword search due to the t-privacy property of SSS as proven in [34, 35]. *(ii) Oblivious Update*: Similar to ODSE^{wo}_{xor}, ODSE^{wo}_{ro} leverages Write-Only ORAM over IND-CPA encrypted database, which offers computational security as shown in [37]. *(iii) ODSE Security:* By Remark 1, for each actual operation, the client triggers both search and update protocols. Given that search is t-private and update pattern is computationally oblivious, the access pattern in ODSE^{wo}_{ro} is a computationally indistinguishable in the presence of t colluding servers. □

Theorem 3. ODSE^{wo}_{it} *scheme is information-theoretically (statistically) t-secure by Definition 3.*

Proof. (Sketch) *(i) Oblivious Search*: ODSE^{wo}_{it} leverages an SSS-based PIR protocol and therefore, achieves t-privacy for keyword search due to the t-privacy property of SSS [34]. *(ii) Oblivious Update*: The index in ODSE^{wo}_{it} is SSS-shared, which is information-theoretically secure in the presence of t colluding servers. ODSE^{wo}_{it} also employs Write-Only ORAM, which offers statistical security due to negligible write failure probability. Therefore in general, the update access pattern of ODSE^{wo}_{it} scheme is information-theoretically (statistically) indistinguishable in the coalition of up to t servers. *(iii) ODSE Security*: By Remark 1, ODSE^{wo}_{it} performs both search and update protocols regardless of the actual operation. As analyzed above, search is t-private and update pattern is statistically t-indistinguishable. Therefore, ODSE^{wo}_{it} is information-theoretically (statistically) t-secure by Definition 3. □

6. ODSE in the Malicious Setting

In previous sections, we have shown that ODSE schemes offer a certain level of collusion-resiliency and robustness in the semi-honest setting where the servers follow the protocol faithfully. In some privacy-critical applications, it is necessary to achieve data integrity and robustness in the malicious

environment, where the adversary can tamper with the query and data to compromise the correctness and privacy of the protocol. In this section, we show that our proposed semi-honest ODSE schemes can be extended to be secure and robust against malicious adversaries.

To achieve integrity of the index and the server-computation, our main idea is to harness computational and information-theoretic message authentication code (MAC) techniques. We first provide the definition of computational and information-theoretic MAC as follows.

• *Computational MAC [32]:* Let $\Sigma = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ be a secure keyed MAC scheme [32]: $\theta \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$ generating a MAC key with security parameter λ; $\mu \leftarrow \Sigma.\mathsf{Mac}_\theta(m)$ generating a tag for message $m \in \{0,1\}^*$ with key θ; $\{0,1\} \leftarrow \Sigma.\mathsf{Vrfy}_\theta(m,\mu)$ verifying if the tag (μ) associated with the message (m) is either valid (1) or invalid (0).

• *Information-theoretic MAC [41]:* Let $\theta \xleftarrow{\$} \mathbb{F}_p$ be a global MAC key, which is known only by the client. The MAC tag (μ) for each data block (b) is computed as $\mu = \theta \cdot b$ (over $\mathbb{F}_p$). Given that the client maintains a consistent relationship between μ, b and θ while keeping them hidden from the adversary, the adversary cannot change b without changing μ and/or α. Therefore, μ is secret-shared among servers along with the shares of b. The verification can be done by reconstructing the block (b) as well as its tag (μ) from the shares, and comparing if $\mu = \theta \cdot b$ holds at the end.

In $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ schemes, we leverage the computational MAC scheme to achieve the integrity of the index encrypted by IND-CPA encryption . On the other hand, the $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ offers information-theoretic security since its index is secret-shared, instead of IND-CPA encrypted. Therefore, we apply information-theoretic MAC to this scheme to preserve its security level. We now present the extensions of ODSE schemes into the malicious setting in details as follows.

6.1. MD-$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$: Maliciously-Detectable $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$

We present $\mathsf{MD\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, the extended version of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ from §4.2, which offers security against malicious adversary using the computational MAC. The verification allows the client to *abort* the protocol if he/she detects any malicious behaviors attempting to tamper with the encrypted index and/or the search/update query. Our $\mathsf{MD\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ protocols are defined as follows.

Setup. Figure 14 presents the setup of $\mathsf{MD\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ scheme with the MAC tag generation for the encrypted index. Generally speaking, it first generates the encrypted index $\mathbf{I}$ similar to semi-honest $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ (line 1), and then generates a MAC key (line 2), followed by computing a matrix $\mathbf{T}$ containing the MAC tag for each $|\mu|$-bit blocks of each row of $\mathbf{I}$ (lines 3-5). In this context, each server in the system stores two matrices including the encrypted index $\mathbf{I}$ and the MAC matrix $\mathbf{T}$.

Search. Figure 15 presents the search protocol of $\mathsf{MD\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, which is extended from the search protocol of semi-honest $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ to be secure against malicious adversary. Specifically, the client generates XOR-PIR queries for ℓ servers similar to the semi-honest $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ scheme (line 1). Each server

$(\sigma, \mathcal{I}) \leftarrow \mathsf{MD\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Setup}(\mathcal{F})$:
1: $(\mathbf{I}, T_f, T_w, \mathbf{c}, \kappa) \leftarrow$ Execute lines 1-14 in Figure 5
2: $\theta \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
3: **for** $i = 1, \ldots, M$ **do**
4: **for** $j = 1, \ldots, 2N/|\mu|$ **do** `# |μ| :(pre-defined) length of the MAC tag.`
5: $\mathbf{T}[i,j] \leftarrow \Sigma.\mathsf{Mac}_\theta(\mathbf{I}[i, (j-1)\cdot|\mu| + 1 \ldots j \cdot |\mu|])$
6: Let $\mathcal{I}$ contain ℓ copies of ($\mathbf{I}$, $\mathbf{T}$) and $\sigma \leftarrow (\theta, \kappa, T_w, T_f, \mathbf{c})$
7: **return** $(\sigma, \mathcal{I})$

Fig. 14. $\mathsf{MD\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ setup algorithm. Extensions from its semi-honest version are highlighted.

$(\mathcal{R}; \perp) \leftarrow$ MD-ODSE$^{\text{wo}}_{\text{xor}}$.Search$(w, \sigma; \mathcal{I})$:

Client:

1: $(\rho_1, \ldots, \rho_\ell) \leftarrow$ Execute lines 1-2 in Figure 6 (to learn that the keyword w corresponds to i^{th} row and request retrieval of i^{th} row privately)
2: Send ρ_l to $\mathcal{S}_l$ for $l \in \{1, \ldots, \ell\}$

Server: each $\mathcal{S}_l \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving ρ_l **do**

3: $\hat{I}_l \leftarrow$ Execute line 4 in Figure 6
4: $\hat{T}_l \leftarrow$ PIR$^{\text{xor}}$.Retrieve$(\rho_l, \mathbf{T}_l)$
5: Send $(\hat{I}_l, \hat{T}_l)$ to the client

Client: On receive $(\langle \hat{I}_1, \ldots, \hat{I}_\ell \rangle, \langle \hat{T}_1, \ldots, \hat{T}_\ell \rangle)$ from ℓ servers

6: $\mathbf{I}[i, *] \leftarrow$ Execute line 6 in Figure 6
7: $\mathbf{T}[i, *] \leftarrow$ PIR$^{\text{xor}}$.Reconstruct$(\hat{T}_1, \ldots, \hat{T}_\ell)$
8: **for** $j = 1 \ldots 2N/|\mu|$ **do**
9: **if** Σ.Vrfy$_\theta(\mathbf{I}[i, (j-1) \cdot |\mu| + 1 \ldots j \cdot |\mu|], \mathbf{T}[i, j]) = 0$ **then**
10: **return** abort
11: $\mathcal{J} \leftarrow$ Execute lines 7-10 in Figure 6
12: **return** $(\mathcal{R}; \perp)$, where $\mathcal{R}$ contains file IDs at column indexes in $\mathcal{J}$

Fig. 15. MD-ODSE$^{\text{wo}}_{\text{xor}}$ search protocol. Extensions from its semi-honest version are highlighted.

$(\sigma'; \mathcal{I}') \leftarrow$ MD-ODSE$^{\text{wo}}_{\text{xor}}$.Update$(f_{id}, \sigma; \mathcal{I})$:

Client:

1: $(S, T_f) \leftarrow$ Execute lines 2-4 in Figure 7
2: $\mathcal{J}' \leftarrow$ Select λ random indexes of $|\mu|$-bit columns in $\mathbf{I}$
3: Send $\mathcal{J}'$ to an arbitrary server $\mathcal{S}_l$

Server $\mathcal{S}_l$**:** On receive $\mathcal{J}'$ **do**

4: Send $\{\mathbf{I}_l[*, (j'-1) \cdot |\mu| + 1 \ldots j' \cdot |\mu|], \mathbf{T}_l[*, j']\}_{j' \in \mathcal{J}'}$ to the client

Client: On receive $\{\mathbf{I}_l[*, (j'-1) \cdot |\mu| + 1 \ldots j' \cdot |\mu|], \mathbf{T}_l[*, j']\}_{j' \in \mathcal{J}'}$ **do**

5: **for** each $j' \in \mathcal{J}'$ **do**
6: **for** $i = 1, \ldots, M$ **do**
7: **if** Σ.Vrfy$_\theta(\mathbf{I}[i, (j'-1) \cdot |\mu| + 1 \ldots j' \cdot |\mu|], \mathbf{T}_l[i, j]) = 0$ **then**
8: **return** abort
9: $(\hat{\mathbf{I}}, S, T_f, \mathbf{c}) \leftarrow$ Execute lines 7-16 in Figure 7 with $\mathcal{J} = \{j', \ldots, j' \cdot |\mu| - 1\}_{j' \in \mathcal{J}'}$
10: $\hat{\mathbf{T}}[i, j'] \leftarrow \Sigma$.Mac$_\theta(\hat{\mathbf{I}}[i, (j'-1) \cdot |\mu| + 1 \ldots j' \cdot |\mu|])$ for each $j' \in \mathcal{J}'$ and $i = 1, \ldots, M$
11: Send $\{\hat{\mathbf{I}}[*, (j'-1) \cdot |\mu| + 1 \ldots j' \cdot |\mu|], \hat{\mathbf{T}}[*, j']\}_{j' \in \mathcal{J}'}$ to ℓ servers

Server: each $\mathcal{S}_l \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $\{\hat{\mathbf{I}}[*, (j'-1) \cdot |\mu| + 1 \ldots j' \cdot |\mu|], \hat{\mathbf{T}}[*, j']\}_{j' \in \mathcal{J}'}$ **do**

12: $\mathbf{I}_l \leftarrow$ Execute lines 18-19 in Figure 7
13: $\mathbf{T}_l[*, j'] \leftarrow \hat{\mathbf{T}}[*, j']$ for each $j' \in \mathcal{J}'$
14: **return** $(\sigma'; \mathcal{I}')$ where $\mathcal{I}'$ are $(\mathbf{I}, \mathbf{T})$ updated at ℓ servers, and σ' is the updated client state

Fig. 16. MD-ODSE$^{\text{wo}}_{\text{xor}}$ update protocol. Extensions from its semi-honest version are highlighted.

performs the XOR-PIR retrieval on both the encrypted index (line 3) and the MAC components (line 4) using the same query received, and sends the result to the client. The client recovers the row of the encrypted index (line 6) as well as its corresponding tag (line 7). The client verifies each $|\mu|$-bit block with its corresponding tag (lines 8-10). If all the tags are valid, the client continues to decrypt the row to obtain the search result as in the semi-honest ODSE$^{\text{wo}}_{\text{xor}}$ scheme (line 11). Otherwise, the client aborts and notifies that at least one of the servers is malicious (line 10).

Update. Figure 16 presents the update protocol of MD-ODSE$^{\text{wo}}_{\text{xor}}$ extended from the semi-honest ODSE$^{\text{wo}}_{\text{xor}}$

for malicious security. Instead of downloading λ random 1-bit columns as in the semi-honest $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, the client downloads λ random columns of $|t|$-bits as well as their corresponding MAC tag. Before decryption, the client verifies the integrity of the retrieved data by the MAC (lines 5-8). If there exists one invalid tag, the client aborts and notifies that at least one server is malicious (line 8). Otherwise, the client performs the update following the same line with the semi-honest $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ (line 9). Finally, the client creates new MAC tags for re-encrypted columns and send all of them to ℓ servers to be updated (lines 10-14).

6.2. MR-ODSE$^{wo}_{ro}$: Maliciously-Robust ODSE$^{wo}_{ro}$

Since $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ relies on SSS for oblivious search, we can extend it in various ways to not only detect but also be robust against malicious adversary. One straightforward extension is to consider SSS as a particular instance of Reed Solomon Code, and then implement Reed Solomon Decoding techniques [39, 40] to handle incorrect server replies. However, this approach can only handle a small number of the malicious servers in the system (e.g., $t < \ell/3$ if using [40]), which might increase the deployment cost. Another approach is to harness the t-out-of-ℓ threshold property of SSS along with the MAC technique presented in the previous section. The main idea is to select $(t+1)$ answers among ℓ answers from the servers to recover the encrypted search result and its MAC tags. If there exists one invalid MAC, we repeat the recover process by selecting a different set of $(t+1)$ answers until we find that all the tags are valid. This strategy offers the detection capability and robustness against malicious behaviors given

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

Fig. 17. MR-ODSE$^{\mathsf{wo}}_{\mathsf{ro}}$ setup algorithm.

$(\mathcal{R}; \perp) \leftarrow \mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}.\mathsf{Search}(w, \sigma; \mathcal{I})$:
Client:
1: $(i, \langle [\![\mathbf{e}]\!]_1, \ldots, [\![\mathbf{e}]\!]_\ell \rangle) \leftarrow$ Execute lines 1-2 in Figure 9
2: Send $[\![\mathbf{e}]\!]_l$ to $\mathcal{S}_l$ for each $l \in \{1, \ldots, \ell\}$
Server: each $\mathcal{S}_l \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $[\![\mathbf{e}]\!]_l$ **do**:
3: $([\![b_1]\!]_l, \ldots, [\![b_{2N'}]\!]_l) \leftarrow$ Execute lines 4-8 in Figure 9
4: $[\![\mu_j]\!]_l \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Retrieve}([\![\mathbf{e}]\!]_l, \mathbf{T}_l[*, j])$ for $j = 1 \ldots, 2N/|\mu|$
5: Send $(\{[\![\mu_j]\!]_l\}_{j=1}^{2N/|\mu|}, \{[\![b_j]\!]_l\}_{j=1}^{2N'})$ to the client
Client: On receive $\{ ([\![\mu_1]\!]_l, \ldots, [\![\mu_{2N/|\mu|}]\!]_l), ([\![b_1]\!]_l, \ldots, [\![b_{2N'}]\!]_l)\}_{l=1}^{\ell}$ from ℓ servers
6: $\mathcal{X} \leftarrow$ Select $t+1$ servers among ℓ servers
7: $\mathcal{B}_j \leftarrow \{[\![b_j]\!]_x\}_{x \in \mathcal{X}, j \in [2N']}$, $\mathcal{T}_j \leftarrow \{[\![\mu_j]\!]_x\}_{x \in \mathcal{X}, j \in \left[\frac{2N}{|\mu|}\right]}$
8: $\mathbf{I}[i, *] \leftarrow$ Execute lines 10-12 in Figure 9
9: **for** $j = 1 \ldots, 2N/|\mu|$ **do**
10: $\quad \mathbf{T}[i, j] \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\mathcal{T}_j, t)$
11: $\quad$ **if** $\Sigma.\mathsf{Vrfy}_\theta(\mathbf{I}[i, (j-1) \cdot |\mu| + 1 \ldots j \cdot |\mu|], \mathbf{T}[i, j]) = 0$ **then**
12: $\quad\quad$ **if** all distinct subset $\mathcal{X}$ have been processed **then**
13: $\quad\quad\quad$ **return** abort
14: $\quad\quad \mathcal{X} \leftarrow$ Select another set of $t+1$ servers and **goto** line 7
15: $\mathcal{J} \leftarrow$ Execute lines 13 -17 in Figure 9
16: **return** $(\mathcal{R}; \perp)$, where $\mathcal{R}$ contains file IDs at column indexes in $\mathcal{J}$

Fig. 18. MR-ODSE$^{\mathsf{wo}}_{\mathsf{ro}}$ search protocol. Extensions from its semi-honest version are highlighted.

$(\sigma';\mathcal{I}') \leftarrow$ MR-ODSE$^{\text{wo}}_{\text{ro}}$.Update$(f_{id},\sigma;\mathcal{I})$:
1: $(\sigma';\mathcal{I}') \leftarrow$ MD-ODSE$^{\text{wo}}_{\text{xor}}$.Update$(f_{id},\sigma;\mathcal{I})$
2: **return** $(\sigma';\mathcal{I}')$

Fig. 19. MR-ODSE$^{\text{wo}}_{\text{ro}}$ update protocol.

that the majority of the servers is honest (i.e., $t < \ell/2$). Therefore, we opt-to this approach to design MR-ODSE$^{\text{wo}}_{\text{ro}}$, the maliciously-robust version of ODSE$^{\text{wo}}_{\text{ro}}$ as follows.

Setup. The index structure of MR-ODSE$^{\text{wo}}_{\text{ro}}$ is identical to that of MD-ODSE$^{\text{wo}}_{\text{xor}}$. Thus, its setup algorithm is identical to that of MD-ODSE$^{\text{wo}}_{\text{xor}}$, where the MAC tag is created for each $|t|$-bit blocks in each row of the encrypted index (Figure 17).

Search. Figure 18 outlines the search protocol of MR-ODSE$^{\text{wo}}_{\text{ro}}$ extended from that of ODSE$^{\text{wo}}_{\text{ro}}$ for malicious security. For each time of oblivious keyword search, the client creates SSS-based PIR query as in the semi-honest ODSE$^{\text{wo}}_{\text{ro}}$ (line 1), and the servers perform the SSS-based PIR retrieval on both the encrypted index (line 3) and MAC components (line 4). Once receiving answers from ℓ servers, the client picks $t+1$ out of ℓ replies (lines 6-7), and performs the SSS recover via the Lagrange interpolation to obtain the encrypted search row (line 8) as well its MAC tag (lines 9-14) . The client verifies the integrity of the encrypted row and decrypts it if all MAC tags are valid. If there exists one invalid tag, the client selects another set of $t+1$ replies, and repeats the verification process. If the client tries all possible sets, which incurs (in total) $\binom{\ell}{t+1}$ verification tests, but none produces all valid tags, the client aborts the protocol and notifies that a majority of servers ($t > \ell/2$) is corrupted (line 13).

Update. The update protocol in MR-ODSE$^{\text{wo}}_{\text{ro}}$ is similar to that of MD-ODSE$^{\text{wo}}_{\text{xor}}$ (Figure 19). To improve the robustness against malicious adversary, the client can request ℓ servers to transfer λ $|t|$-bit columns, and selects one of ℓ replies to verify the integrity and performs the update.

6.3. MR-ODSE$^{wo}_{it}$: Maliciously-Robust and IT-Secure ODSE$^{wo}_{it}$

In this section, we present MR-ODSE$^{\text{wo}}_{\text{it}}$, the extended version of ODSE$^{\text{wo}}_{\text{it}}$ that inherits all properties of ODSE$^{\text{wo}}_{\text{it}}$ (e.g., information-theoretic security) along with the robustness against malicious adversary. To preserve the information-theoretic security, we use the information-theoretic MAC as defined above for each block. The details are as follows.

Setup. MR-ODSE$^{\text{wo}}_{\text{it}}$ follows the principles in the semi-honest ODSE$^{\text{wo}}_{\text{it}}$ scheme to create the share index (Figure 20, line 1). It then creates a global MAC key by selecting a random element in $\mathbb{F}_p$ (line 2). It multiplies the representative element in $\mathbb{F}_p$ of each index block with the global MAC key over $\mathbb{F}_p$ yielding the MAC tag, and then creates the SSS shares for each tag (line 3). The SSS shares of MAC tags are distributed along with the share index across ℓ servers.

Search. Figure 21 presents the search protocol of MR-ODSE$^{\text{wo}}_{\text{it}}$ extended from that of ODSE$^{\text{wo}}_{\text{it}}$ for malicious security. The extension follows the line of the MR-ODSE$^{\text{wo}}_{\text{ro}}$ scheme. Specifically, the servers

$(\sigma,\mathcal{I}) \leftarrow$ MR-ODSE$^{\text{wo}}_{\text{it}}$.Setup$(\mathcal{F})$:
1: $(\langle \mathbf{I}_1,\ldots,\mathbf{I}_\ell\rangle, T_w, T_f, \langle b_{11},\ldots,b_{M2N'}\rangle) \leftarrow$ Execute lines 1-5 in Figure 12
2: $\alpha \xleftarrow{\$} \mathbb{F}_p$
3: $(\mathbf{T}_1[i,j],\ldots,\mathbf{T}_\ell[i,j]) \leftarrow$ SSS.CreateShare$(\alpha \cdot b_{ij}, t)$ for $i = 1,\ldots,M$ and for $j = 1,\ldots,2N'$
4: **return** $(\sigma,\mathcal{I})$, where $\mathcal{I} \leftarrow \{\langle \mathbf{I}_1,\ldots,\mathbf{I}_\ell\rangle, \langle \mathbf{T}_1,\ldots,\mathbf{T}_\ell\rangle\}$ and $\sigma \leftarrow (\alpha, T_w, T_f)$

Fig. 20. MR-ODSE$^{\text{wo}}_{\text{it}}$ setup algorithm. Extensions from its semi-honest version are highlighted.

$(\mathcal{R}; \perp) \leftarrow$ MR-ODSE$^{\text{wo}}_{\text{it}}$.Search$(w, \sigma; \mathcal{I})$:

Client:

1: $(i, \langle [\![\mathbf{e}]\!]_1, \ldots, [\![\mathbf{e}]\!]_\ell \rangle) \leftarrow$ Execute lines 1-2 in Figure 12
2: Send $[\![\mathbf{e}]\!]_l$, to $\mathcal{S}_l$ for each $l \in \{1, \ldots, \ell\}$

Server: each $\mathcal{S}_l \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $[\![\mathbf{e}]\!]_l$ **do**

3: $([\![b_1]\!]_i, \ldots, [\![b_{2N'}]\!]_i) \leftarrow$ Execute lines 4 -5 in Figure 12
4: $[\![\mu_j]\!]_l \leftarrow$ PIR$^{\text{sss}}$.Retrieve$([\![\mathbf{e}]\!]_l, \mathbf{T}_l[*, j])$ for each $j \in \{1 \ldots, 2N'\}$
5: Send $(\langle [\![\mu_1]\!]_l, \ldots, [\![\mu_{2N'}]\!]_l \rangle, \langle [\![b_1]\!]_l, \ldots, [\![b_{2N'}]\!]_l \rangle)$ to the client

Client: On receive $\{\langle [\![\mu_k]\!]_1, \ldots, [\![\mu_k]\!]_\ell \rangle, \langle [\![b_k]\!]_1, \ldots, [\![b_k]\!]_\ell \rangle\}_{k=1}^{2N'}$ from ℓ servers

6: $\mathcal{X} \leftarrow$ Select $2t+1$ servers among ℓ servers
7: $\mathcal{B}_j \leftarrow \{[\![b_j]\!]_x\}_{x \in \mathcal{X}, j \in [2N']}$, $\mathcal{T}_j \leftarrow \{[\![\mu_j]\!]_x\}_{x \in \mathcal{X}, j \in [2N']}$
8: $(b_1, \ldots, b_{2N'}) \leftarrow$ Execute lines 7-8 in Figure 12
9: **for** $j = 1 \ldots, 2N'$ **do**
10: $\quad \mu_j \leftarrow$ PIR$^{\text{sss}}$.Reconstruct$(\mathcal{T}_j, 2t)$
11: $\quad$ **if** $(\alpha \cdot \beta_j \neq \mu_j)$ **then**
12: $\quad\quad$ **if** (all distinct subset $\mathcal{X}$ have been processed) **then**
13: $\quad\quad\quad$ **return** abort
14: $\quad\quad \mathcal{X} \leftarrow$ Select another set of $2t+1$ servers and **goto** line 7
15: $\mathcal{J} \leftarrow$ Execute lines 9-10 in Figure 12
16: **return** $(\mathcal{R}; \perp)$, where $\mathcal{R}$ contains file IDs at column indexes in $\mathcal{J}$

Fig. 21. MR-ODSE$^{\text{wo}}_{\text{it}}$ search protocol. Extensions from its semi-honest version are highlighted.

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

Client:

1: $(\mathcal{J}, S, T_f) \leftarrow$ Execute lines 1-5 in Figure 13
2: Send $\mathcal{J}$ to ℓ servers $(\mathcal{S}_1, \ldots, \mathcal{S}_\ell)$

Server: each $\mathcal{S}_l \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $\mathcal{J}$ **do**

3: Send $\{\mathbf{I}_l[*, j], \mathbf{T}_l[*, j]\}_{j \in \mathcal{J}}$ to the client

Client: On receive $\{\langle \mathbf{I}_1[*, j], \ldots, \mathbf{I}_\ell[*, j] \rangle, \langle \mathbf{T}_1[*, j], \ldots, \mathbf{T}_\ell[*, j] \rangle\}_{j \in \mathcal{J}}$ **do**

4: $\mathcal{X} \leftarrow$ Select $t+1$ servers among ℓ servers
5: $\mathcal{B}_{ij} \leftarrow \{\mathbf{I}[i, j]_x\}_{x \in \mathcal{X}, j \in \mathcal{J}, i \in [M]}$
6: $\{\mathbf{I}'[*, j], \langle b_{1j}, \ldots, b_{Mj} \rangle\}_{j \in \mathcal{J}} \leftarrow$ Execute lines 7-10 in Figure 13
7: **for** $i = 1 \ldots M$ **do**
8: $\quad$ **for** each $j \in \mathcal{J}$ **do**
9: $\quad\quad$ **if** $(\alpha \cdot \mathbf{I}'[i, j] \neq \mathbf{T}[i, j])$ **then**
10: $\quad\quad\quad$ **if** (all distinct subset $\mathcal{X}$ have been processed) **then**
11: $\quad\quad\quad\quad$ **return** abort
12: $\quad\quad\quad \mathcal{X} \leftarrow$ Select another set of $t+1$ servers and **goto** line 5
13: $\{\langle \hat{\mathbf{I}}_1[*, j], \ldots \hat{\mathbf{I}}_\ell[*, j] \rangle, \langle b'_{1j}, \ldots, b'_{Mj} \rangle\}_{j \in \mathcal{J}} \leftarrow$ Execute lines 11-16 in Figure 13
14: $(\hat{\mathbf{T}}_1[i, j], \ldots, \hat{\mathbf{T}}_\ell[i, j]) \leftarrow$ SSS.CreateShare$(\alpha \cdot b'_{ij}, t)$ for each $j \in \mathcal{J}$ and for $i = 1 \ldots M$
15: Send $\{\hat{\mathbf{I}}_l[*, j], \hat{\mathbf{T}}_l[*, j]\}_{j \in \mathcal{J}}$ to $\mathcal{S}_l$ for $l = 1, \ldots, \ell$

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

16: $\{\mathbf{I}_l[*, j]\}_{j \in \mathcal{J}} \leftarrow$ Execute lines 18-19 in Figure 13
17: $\mathbf{T}_l[*, j] \leftarrow \hat{\mathbf{T}}_l[*, j]$ for each $j \in \mathcal{J}$
18: **return** $(\sigma'; \mathcal{I}')$ where $\mathcal{I}'$ are $(\mathbf{I}_l, \mathbf{T}_l)$ updated at ℓ servers and σ' is updated client state

Fig. 22. MR-ODSE$^{\text{wo}}_{\text{it}}$ update protocol. Extensions from its semi-honest version are highlighted.

perform SSS-based PIR retrieval on both index and the MAC components (lines 3-4). The client picks $2t+1$ out of ℓ replies to recover and verify the integrity of the search result (lines 6-7). If after $\binom{\ell}{2t+1}$ trials with different subsets but none producing the valid tags, the client aborts the protocol and notifies

that more than $\ell/3$ servers are malicious (line 7). Otherwise, the client continues to process the recovered data as in the semi-honest MR-ODSE$^{\text{wo}}_{\text{it}}$ scheme to obtain the final search result (line 15).

Update. Figure 22 presents the update protocol of MR-ODSE$^{\text{wo}}_{\text{it}}$. Basically, the client downloads λ columns of the share index and their corresponding MAC from ℓ servers. The client selects $t+1$ replies to recover and verify the integrity of downloaded data before performing update. If all tags are valid, the client performs the write-only ORAM procedure as in ODSE$^{\text{wo}}_{\text{it}}$ scheme, re-calculates the MAC tag for each block, and then creates new SSS shares for each tag. Otherwise, the client aborts the protocol and notifies that a majority of servers is malicious.

7. Implementation

We fully implemented all ODSE schemes in C++. We used `Google Sparsehash` library [42] to implement position maps T_f and T_w. We utilized `Intel AES-NI` library [43] to implement AES-CTR encryption/decryption in ODSE$^{\text{wo}}_{\text{xor}}$ and ODSE$^{\text{wo}}_{\text{ro}}$ schemes. We leveraged `Shoup NTL` library [44] for pseudo-random number generator and arithmetic operations over finite field. We used `ZeroMQ` library [45] for client-server communication. We used multi-threading technique to accelerate PIR computation at the server. The code is available at https://github.com/thanghoang/ODSE.

8. Performance Evaluation

8.1. Configurations

Hardware and network settings. We used Amazon EC2 with `r4.4xlarge` instance for server(s), each equipped with 16 vCPUs Intel Xeon @ 2.3 GHz and 122 GB RAM. We used a laptop with Intel Core i5 @ 2.90 GHz and 16 GB RAM as the client. All machines ran Ubuntu 16.04. The client established a network connection with the server via WiFi connection. We used a real network setting, in which the download/upload throughput is 27/5 Mbps, respectively.

Dataset. We used the subsets of the `Enron` dataset to build $\mathbf{I}$ containing from millions to billions of keyword-file pairs. The largest dataset contain around 300,000 files with 320,000 unique keywords. Our tokenization is identical to [25] so that our keyword distribution and query pattern are similar to [25].

Instantiation of compared techniques. We compared ODSE with a standard DSSE scheme [7], and the use of generic ORAM atop the DSSE encrypted index. The performance of all schemes was measured under the same setting and configuration We configured ODSE schemes and their counterparts as follows.

• *ODSE*: For the semi-honest setting, we deployed two servers for ODSE$^{\text{wo}}_{\text{xor}}$ and ODSE$^{\text{wo}}_{\text{ro}}$ schemes, and three servers for ODSE$^{\text{wo}}_{\text{it}}$ scheme. We selected $\lambda = 4$ for ODSE$^{\text{wo}}_{\text{xor}}$ and ODSE$^{\text{wo}}_{\text{ro}}$, and $\lambda' = 4$ with $\mathbb{F}_p$ where p is a 16-bit prime for ODSE$^{\text{wo}}_{\text{ro}}$ schemes ODSE$^{\text{wo}}_{\text{it}}$. We note that selecting larger p (e.g., $|p| = 64$ bits) can reduce the PIR computation time with the cost of the bandwidth overhead due to the increase of query size. We chose a 16-bit prime field to achieve a balanced computation and communication overhead. For the malicious setting, we first fixed the number of servers for ODSE$^{\text{wo}}_{\text{xor}}$ and ODSE$^{\text{wo}}_{\text{ro}}$ schemes to be two, three and four, respectively to handle one adversary. We then increased the number of servers to allow more malicious servers (see §8.6 for details).

• *Standard DSSE*: We selected one of the most efficient DSSE schemes by Cash et al. in [7] (i.e., Π^{dyn}_{2lev} variant) to demonstrate the performance gap between ODSE and the standard DSSE. We estimated the performance of Π^{dyn}_{2lev} using the same software/hardware environments and optimizations as ODSE (e.g., parallelization, AES-NI acceleration). Note that we did not use the Java implementation of this scheme available in `Clusion` library [46] for comparison due to its lack of hardware acceleration support (i.e., no AES-NI) and the difference between running environments (Java VM *vs.* C). Our estimation is *conservative* in which, we used numbers that would be better than the Clusion library.

• *Using generic ORAM atop DSSE encrypted index*: We selected *non-recursive* Path-ORAM [23] and Ring-ORAM [38], as ODSE counterparts since they are the most efficient generic ORAM schemes for data outsourcing to date. Since we focus on encrypted index rather than encrypted files in DSSE, we did not explicitly compare our schemes with TWORAM [24] but instead used one of their techniques to optimize the performance of using generic ORAM on DSSE encrypted index. Specifically, we applied the selected ORAMs on the dictionary index as in [25] along with the round-trip optimization as in [24]. Note that these estimates are also conservative, where memory access delays were excluded, and cryptographic operations were optimized and parallelized for an objective comparison.

8.2. Overall End-to-end Delay in the Semi-honest Setting

Figure 23 presents the end-to-end delays of ODSE schemes and their counterparts, where we performed both search and update protocols in ODSE schemes to hide the actual type of operation (see Remark 1). ODSE offers a higher security than standard DSSE at the cost of a longer delay. Nevertheless, ODSE schemes are $3\times$-$57\times$ faster than the use of generic ORAMs atop DSSE encrypted index to hide the access patterns. Specifically, with an encrypted index consisting of ten billions of keyword-file pairs, Π^{dyn}_{2lev} cost 36 milliseconds and 600 milliseconds to finish a search and update operation, respectively. ODSE^{wo}_{xor} and ODSE^{wo}_{it}, respectively, took 2.8 seconds and 8.6 seconds to accomplish both keyword search and file update operations, compared with 160 seconds by using Path-ORAM with the round-trip optimization [24].

We present the separate delay for the search and update operations in ODSE schemes in Table 2. ODSE^{wo}_{xor} is the most efficient in terms of search, whose delay was less than 1 second. This is due to the fact that ODSE^{wo}_{xor} only triggers XOR operations and the size of the search query is minimal (i.e., a binary

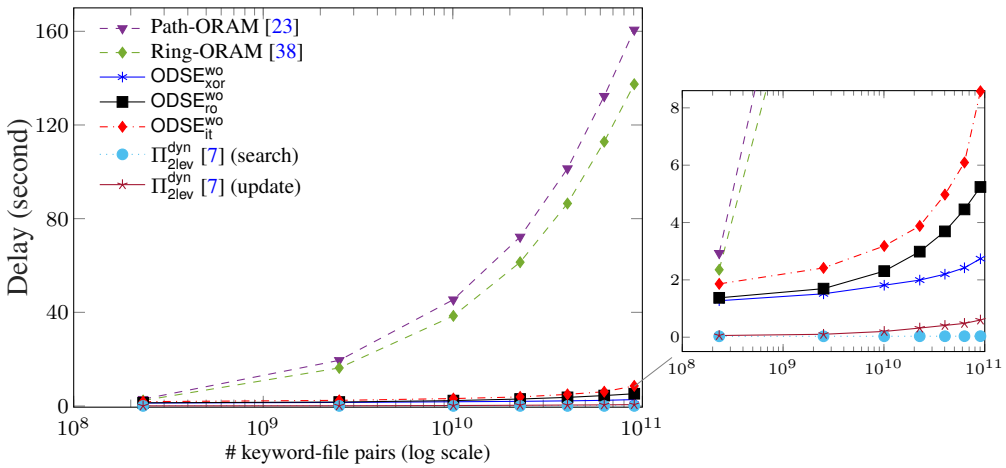


Fig. 23. Latency of semi-honest ODSE schemes and their counterparts.

Table 2
Comparison of ODSE and its counterparts for oblivious access on **I**.

Scheme	Security				Delay (second)		Distributed Setting†	
	Forward privacy	Backward privacy	Hidden access pattern‡	Encrypted index*	Search	Update	Privacy level	Improved Robustness
Standard DSSE [7]	✗	✗	✗	Computational	0.036	0.62	-	-
Path-ORAM[23]	✓	✓	Computational	Computational	160.6		-	-
Ring-ORAM [38]	✓	✓	Computational	Computational	137.4		-	-
$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$	✓	✓	Computational	Computational	**0.48**	**2.32**	$\ell - 1$	✗
$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$	✓	✓	Computational	Computational	**3.45**	**1.85**	$< \ell$	✓
$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$	✓	✓	**Information theoretic**	**Information theoretic**	**4.54**	**4.08**	$< \ell/2$	✓

This delay is for *semi-honest* setting with encrypted index containing 300,000 files and 320,000 keywords under the network and configuration presented in §8.1.
* The encrypted index in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is information-theoretically secure because it is SSS. Other schemes employ IND-CPA encryption so that their index is computationally secure (see §5).
‡ All ODSE schemes perform search and update protocols to hide the actual query type. In $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, search is IT-secure due to SSS-based PIR and update is computationally secure due to IND-CPA encryption. Hence, its overall security is computational.
† ℓ is # servers in the system. We define the *robustness* in distributed setting as the ability to tolerate unresponsive server(s) in the semi-honest setting or incorrect replies in the malicious setting. In $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$, encrypted index and search query are SSS with the same privacy level. Generic ORAM solutions have a stronger adversarial model than ours because they are not vulnerable to collusion that arises in the distributed setting.

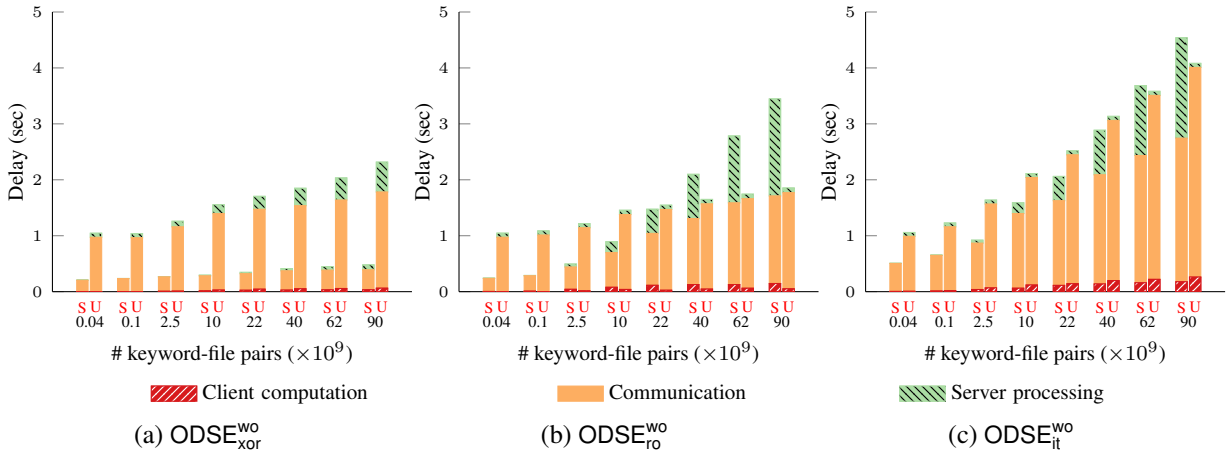
string). $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ are more robust (e.g., malicious tolerant) and one of which is more secure (e.g., information-theoretic security) than $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ at the cost of higher search delay (i.e., 4 seconds) due to its larger search query and SSS arithmetic computations. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is the slowest among the three ODSE schemes since it requires three servers and, therefore, the client needs to transmit more data.

For the oblivious file update, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ 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 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ because 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 oblivious operations, especially the keyword search, is much more efficient than generic ORAMs. We further provide a comparison of ODSE schemes with their counterparts in Table 2. In the following section, we dissect the end-to-end delay of ODSE schemes to understand which factors contributing the most to their performance.

8.3. *Detailed Cost Analysis*

Figure 24 presents the detailed delays of separate keyword search and file update operations in ODSE schemes. There are three main factors impacting the end-to-end delay of ODSE schemes as follows.

• *Client processing:* As shown in Figure 24, the client computation contributes the least amount to the overall search delay (less than 10%) in all ODSE schemes. It comprises the following operations: (*i*) Generate search queries with PRF in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ or SSS in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ schemes; (*ii*) SSS recovery (in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$) and/or IND-CPA decryption (in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$); (*iii*) Filter dummy columns and collect columns in the stash. Note that the client delay of ODSE schemes can be further reduced (by at least 50%-60%) via pre-computation of some values such as row keys and PIR queries (only contain shares of 0 or 1). For the file update, the client performs either decryption followed by 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



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

Fig. 24. Detailed Search (S) and Update (U) costs of semi-honest ODSE schemes.

used crypto acceleration (i.e., Intel AES-NI) and highly optimized number theory libraries (i.e., NTL), all these computations only contributed to a small fraction of the total delay.

• *Client-server communication:* Data transmission is the most dominating factor in the delay of ODSE schemes. The communication cost of ODSE$^{wo}_{xor}$ is the smallest among all ODSE schemes since the size of search query and the data transmitted from servers are only binary strings. In ODSE$^{wo}_{ro}$ and ODSE$^{wo}_{it}$ schemes, the size of components in the search query vector is 16 bits. Their communication overhead can be reduced by using a smaller finite field at the cost of increased PIR computation on the server side.

• *Server processing:* The cost of PIR operations in ODSE$^{wo}_{xor}$ is negligible as it uses XOR tricks. The PIR computation overhead in ODSE$^{wo}_{ro}$ and ODSE$^{wo}_{it}$ is reasonable because it operates on a considerably large amount of 16-bit values. For the file update operations, the server-side cost is mainly due to memory accesses to overwrite some columns of the encrypted index. ODSE$^{wo}_{ro}$ and ODSE$^{wo}_{it}$ schemes are highly memory access-efficient since we store their matrix-based index column-wise in the memory. This memory layout organization allows the inner product in PIR to access *contiguous* memory blocks thereby, minimizing the memory access delay not only in the update but also in the search. In ODSE$^{wo}_{xor}$, we stored the matrix row-wise for row-friendly access to permit efficient XOR operations during search. However, this requires file update to access non-contiguous memory blocks. Hence, the file update in ODSE$^{wo}_{xor}$ incurred a higher memory access delay than that of ODSE$^{wo}_{ro}$ and ODSE$^{wo}_{it}$ as shown in Figure 24.

8.4. Storage overhead

The main limitation of ODSE schemes is the size of encrypted index, whose asymptotic cost is $\mathcal{O}(N \cdot M)$, where N and M are the number of files and unique keywords, respectively. Given the largest database being experimented, the size of our encrypted index is 23 GB. The client storage includes two position maps of size $\mathcal{O}(M \log M)$ and $\mathcal{O}(N \log N)$, the stash of size $\mathcal{O}(M \cdot \log N)$, a counter vector of size $\Omega(N)$ and a master key (in ODSE$^{wo}_{xor}$ scheme). Empirically, with the same database size discussed above, the client requires approximately 22 MB in all ODSE schemes.

8.5. Experiment with various query sizes

We studied the performance of our schemes and their counterparts in the context of various keyword and file numbers involved in search and update operations that we refer to as “query size". As shown

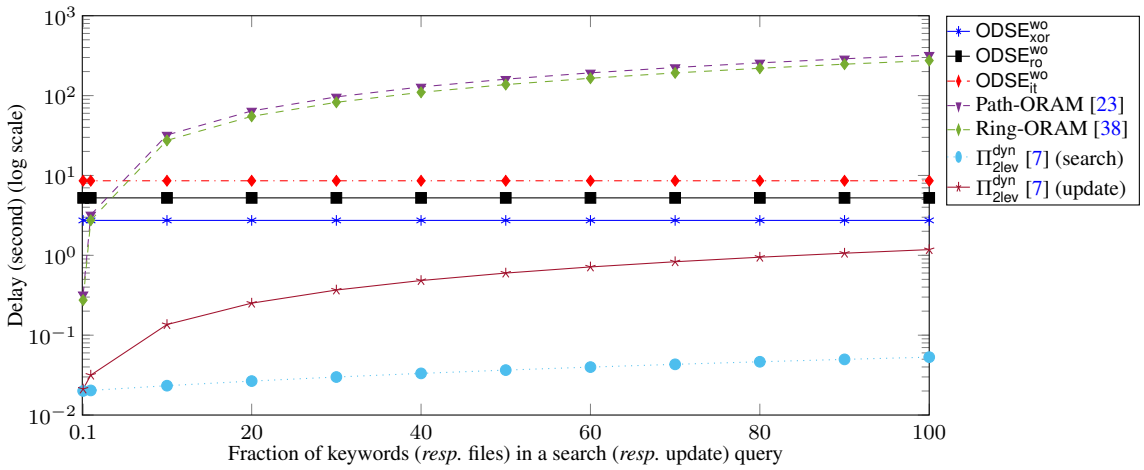


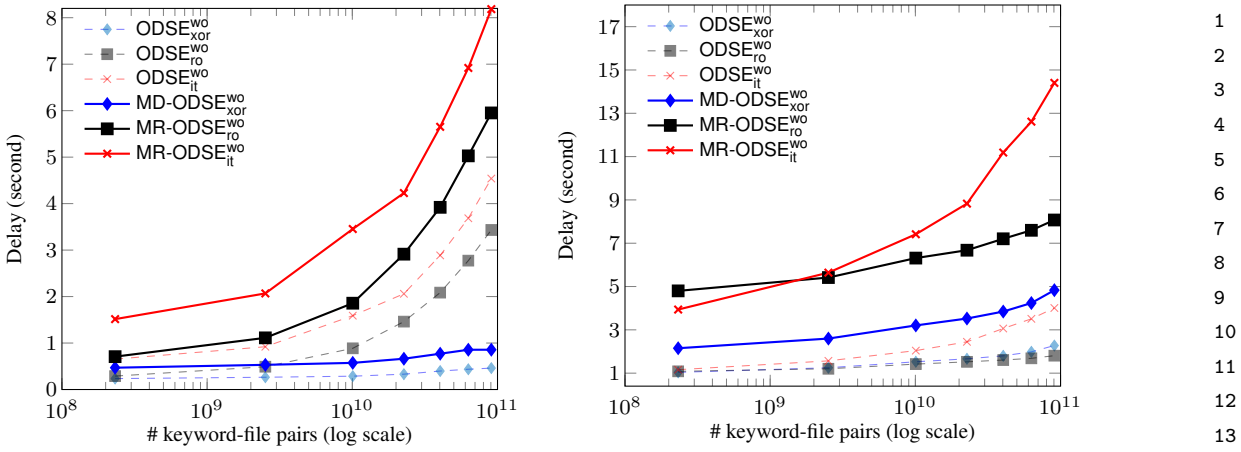
Fig. 25. Delay of semi-honest ODSE schemes and their counterparts with different fraction of keywords/files involved in a search/update.

in Figure 25, ODSE schemes are more efficient than using generic ORAMs when more than 5% of keywords/files in the database are involved in the search/update operations. Since the complexity of ODSE schemes is linear to the number of keywords and files (i.e., $\mathcal{O}(M+N)$), their delay is constant and independent from the query size. The complexity of ORAM approaches is $\mathcal{O}(r\log^2(N\cdot M))$, where r is the query size. Although the bandwidth cost of ODSE schemes is asymptotically linear, their actual delay is much lower than using generic ORAM, whose cost is poly-logarithmic to the total number of keywords/files but linear to the query size. This confirms the results of Naveed et al. in [25] on the performance limitations of generic ORAM and DSSE composition, wherein we used the same dataset for our experiments.

8.6. *ODSE Performance in the Presence of Malicious Adversary*

In this section, we present the performance of maliciously-secure ODSE schemes described in §6. Figure 26 presents the search and update delay of $\text{MD-ODSE}^{\text{wo}}_{\text{xor}}$, $\text{MR-ODSE}^{\text{wo}}_{\text{ro}}$ and $\text{MR-ODSE}^{\text{wo}}_{\text{it}}$ schemes in the presence of one malicious adversary, compared with their corresponding semi-honest version. Recall that in this setting, we set the number of servers in the system for $\text{MD-ODSE}^{\text{wo}}_{\text{xor}}$, $\text{MR-ODSE}^{\text{wo}}_{\text{ro}}$ and $\text{MR-ODSE}^{\text{wo}}_{\text{it}}$ schemes to be two, three and four, respectively. We can see that the search delays of maliciously-secure ODSE schemes are around two times slower than their semi-honest version. It is mainly due to the additional processing and network transmission overhead for the MAC components stored at the server-side, which has the same size with the encrypted index. The update of $\text{MR-ODSE}^{\text{wo}}_{\text{ro}}$ and $\text{MR-ODSE}^{\text{wo}}_{\text{it}}$ schemes are around three times slower than that of their semi-honest version. The main reason is that $\text{MR-ODSE}^{\text{wo}}_{\text{ro}}$ and $\text{MR-ODSE}^{\text{wo}}_{\text{it}}$ requires an extra server in the system to detect one malicious adversary, which leads to the increase of the client bandwidth overhead.

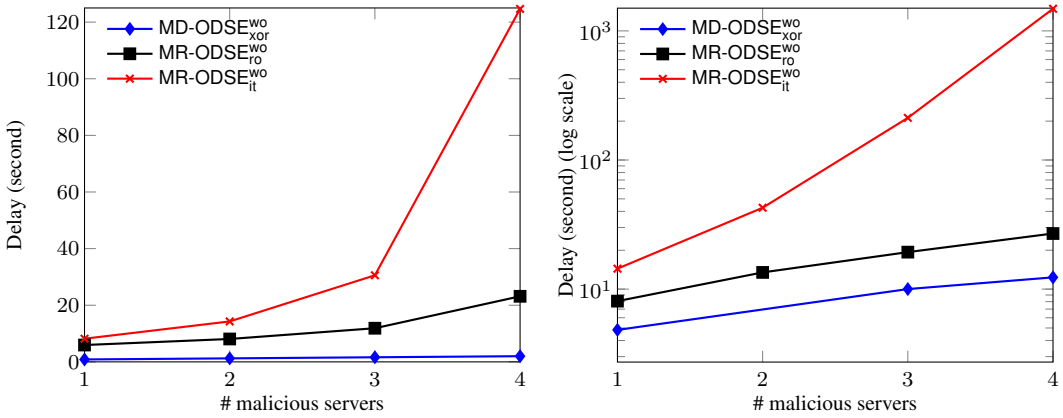
We also explored the performance of maliciously-secure ODSE schemes when the number of malicious servers increases. Allowing more servers to be malicious requires to deploy more servers in the system. Specifically, $\text{MR-ODSE}^{\text{wo}}_{\text{ro}}$ and $\text{MR-ODSE}^{\text{wo}}_{\text{it}}$ schemes need $2t+1$ and $3t+1$ servers in total to be robust against t number of malicious servers, respectively. Figure 27 presents the performance of maliciously-secure ODSE schemes with the varied number of corrupted servers. We can see that it is expensive to offer the robustness for a number of malicious servers in the system. This is because it



(a) search (b) update

Fig. 26. End-to-end delay of maliciously-secure ODSE schemes in the presence of one malicious adversary.

incurs not only the client bandwidth overhead to communicate with more servers, but also the client computation overhead. In the worst case, MR-ODSE$^{wo}_{ro}$ and MR-ODSE$^{wo}_{it}$ requires the client to perform $\binom{\ell}{t+1}$ and $\binom{\ell}{2t+1}$ times of MAC verification, respectively, to find an authentic $|t|$-bit data block in the presence of (less than) t malicious servers. Since MD-ODSE$^{wo}_{xor}$ can only detect the malicious behavior (without knowing which server it is), its overhead only increases slightly when allowing more servers to be malicious. This is because it only requires to deploy more servers in the system, and the client aborts the protocol immediately when he/she finds an invalid MAC tag (without trying aggressively to find an alternative authentic block as in MR-ODSE$^{wo}_{ro}$ and MR-ODSE$^{wo}_{it}$ schemes).



(a) search (b) update

Fig. 27. Delay of maliciously-secure ODSE schemes with varied number of malicious servers.

9. Conclusion

In this article, we present a novel Oblivious Distributed DSSE framework called ODSE, which offers access pattern obliviousness, hidden size pattern, and low end-to-end for index access. These properties are achieved by exploiting unique characteristics of the index data structure and searchable encryption, which allows to deploy computation- and bandwidth-efficient techniques (i.e., multi-server PIR and Write-Only ORAM) to conduct oblivious search and update separately. Our framework contains a series of ODSE schemes each featuring different levels of performance and security in terms of data confidentiality and access pattern obliviousness. Specifically, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ offers the lowest end-to-end delay, smallest bandwidth overhead and the highest resiliency against colluding servers. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ offers the robustness and information-theoretic security for access patterns and the encrypted index. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ inherits the best of both $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ schemes: low end-to-end delay and robustness in the distributed setting. All these schemes can also be extended to be secure/robust against malicious adversary.

References

[1] 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*, IEEE Computer Society, 2000, pp. 44–55.

[2] R. Curtmola, J. Garay, S. Kamara and R. Ostrovsky, Searchable symmetric encryption: improved definitions and efficient constructions, in: *Proceedings of the 13th ACM CCS*, ACM, 2006, pp. 79–88.

[3] C. Wang, N. Cao, J. Li, K. Ren and W. Lou, Secure ranked keyword search over encrypted cloud data, in: *IEEE 30th International Conference on Distributed Computing Systems*, IEEE, 2010, pp. 253–262.

[4] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y.T. Hou and H. Li, Privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking, in: *ACM SIGSAC AsiaCCS*, ACM, 2013, pp. 71–82.

[5] N. Cao, C. Wang, M. Li, K. Ren and W. Lou, Privacy-preserving multi-keyword ranked search over encrypted cloud data, *IEEE Transactions on parallel and distributed systems* **25**(1) (2014), 222–233.

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

[7] 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** (2014), 853.

[8] R. Bost, B. Minaud and O. Ohrimenko, Forward and backward private searchable encryption from constrained cryptographic primitives, Technical Report, IACR Cryptology ePrint Archive 2017, 2017.

[9] R. Zhang, R. Xue, T. Yu and L. Liu, Dynamic and Efficient Private Keyword Search over Inverted Index–Based Encrypted Data, *ACM Transactions on Internet Technology (TOIT)* **16**(3) (2016), 21.

[10] F. Zhou, Y. Li, A.X. Liu, M. Lin and Z. Xu, Integrity Preserving Multi-keyword Searchable Encryption for Cloud Computing, in: *International Conference on Provable Security*, Springer, 2016, pp. 153–172.

[11] T. Moataz, I. Ray, I. Ray, A. Shikfa, F. Cuppens and N. Cuppens, Substring search over encrypted data, *Journal of Computer Security* (2018), 1–30.

[12] C. Bösch, P. Hartel, W. Jonker and A. Peter, A survey of provably secure searchable encryption, *ACM Computing Surveys (CSUR)* **47**(2) (2015), 18.

[13] D. Cash, P. Grubbs, J. Perry and T. Ristenpart, Leakage-abuse attacks against searchable encryption, in: *Proceedings of the 22nd ACM CCS*, 2015, pp. 668–679.

[14] D. Pouliot and C.V. Wright, The Shadow Nemesis: Inference Attacks on Efficiently Deployable, Efficiently Searchable Encryption, in: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 1341–1352.

[15] M.S. Islam, M. Kuzu and M. Kantarcioglu, Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation., in: *NDSS*, 2012.

[16] C. Liu, L. Zhu, M. Wang and Y.-a. Tan, Search pattern leakage in searchable encryption: Attacks and new construction, *Information Sciences* (2014).

[17] 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)*, 2016, pp. 707–720.

[18] E. Stefanov, C. Papamanthou and E. Shi, Practical Dynamic Searchable Encryption with Small Leakage, *NDSS, San Diego, California, USA* (2014).

[19] 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*, ACM, 2014, pp. 310–320.
[20] S. Kamara and C. Papamanthou, Parallel and dynamic searchable symmetric encryption, in: *Financial Cryptography and Data Security*, Springer, 2013, pp. 258–274.
[21] R. Bost, Sophos – Forward Secure Searchable Encryption, in: *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, ACM, 2016.
[22] M.D. Green and I. Miers, Forward secure asynchronous messaging from puncturable encryption, in: *Security and Privacy (SP), 2015 IEEE Symposium on*, IEEE, 2015, pp. 305–320.
[23] 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 CCS*, ACM, 2013, pp. 299–310.
[24] S. Garg, P. Mohassel and C. Papamanthou, TWORAM: Round-Optimal Oblivious RAM with Applications to Searchable Encryption., *IACR Cryptology ePrint Archive* **2015** (2015), 1010.
[25] M. Naveed, The Fallacy of Composition of Oblivious RAM and Searchable Encryption, in: *Cryptology ePrint Archive, Report 2015/668*, 2015.
[26] 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*, Springer, 2016, pp. 145–174.
[27] P. Paillier, Public-key cryptosystems based on composite degree residuosity classes, in: *International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 1999, pp. 223–238.
[28] I. Abraham, C.W. Fletcher, K. Nayak, B. Pinkas and L. Ren, Asymptotically Tight Bounds for Composing ORAM with PIR, in: *IACR Public Key Cryptography*, Springer, 2017, pp. 91–120.
[29] 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.
[30] C. Bosch, A. Peter, B. Leenders, H.W. Lim, Q. Tang, H. Wang, P. Hartel and W. Jonker, Distributed searchable symmetric encryption, in: *Privacy, Security and Trust (PST), 12th International Conference on*, IEEE, 2014, pp. 330–337.
[31] T. Hoang, A.A. Yavuz, F.B. Durak and J. Guajardo, Oblivious Dynamic Searchable Encryption on Distributed Cloud Systems, in: *IFIP Annual Conference on Data and Applications Security and Privacy*, Springer, 2018, pp. 113–130.
[32] J. Katz and Y. Lindell, *Introduction to modern cryptography*, CRC Press, 2014.
[33] A. Shamir, How to share a secret, *Communications of the ACM* (1979).
[34] I. Goldberg, Improving the robustness of private information retrieval, in: *IEEE Symposium on Security and Privacy*, IEEE, 2007, pp. 131–148.
[35] A. Beimel and Y. Stahl, Robust information-theoretic private information retrieval, in: *International Conference on Security in Communication Networks*, Springer, 2002, pp. 326–341.
[36] B. Chor, E. Kushilevitz, O. Goldreich and M. Sudan, Private information retrieval, *Journal of the ACM (JACM)* (1998).
[37] 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 CCS*, ACM, 2014, pp. 203–214.
[38] 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).
[39] V. Guruswami and M. Sudan, Improved decoding of Reed-Solomon and algebraic-geometric codes, in: *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, IEEE, 1998, pp. 28–37.
[40] L.R. Welch and E.R. Berlekamp, Error correction for algebraic block codes, Google Patents, 1986, US Patent 4,633,470.
[41] I. Damgård, V. Pastro, N. Smart and S. Zakarias, Multiparty computation from somewhat homomorphic encryption, in: *Annual Cryptology Conference*, Springer, 2012, pp. 643–662.
[42] sparsehash: An extemely memory efficient hash_map implementation, February 2012.
[43] S. Gueron, White Paper: Intel Advanced Encryption Standard (AES) New Instructions Set, Document Revision 3.01, September 2012.
[44] V. Shoup, NTL: A Library for doing Number Theory, 2016.
[45] ZeroMQ library, 2016.
[46] The Clusion Library.