Privacy-Preserving and Functional Information Systems

by

Thang Hoang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Attila A. Yavuz, Ph.D.
Jean-François Biasse, Ph.D.
Morris Chang, Ph.D.
Jay Ligatti, Ph.D.
Xinming (Simon) Ou, Ph.D.
Mike Rosulek, Ph.D.

Date of Approval:
April 22, 2020

Keywords: privacy-enhancing technologies, secure computation, oblivious RAM

Dedication

To my beloved family and friends, especially my little Christopher Khang.

Acknowledgments

I would like to express my deepest appreciation to my PhD advisor, Dr. Attila Yavuz, who devoted significant time and efforts to guide me through my graduate education as well as provide best advice in life. A special thank to Dr. Jorge Guarjado for offering me internship opportunities in his research lab and letting me freely explore exciting research areas. Jorge is not only my mentor but also a friend, who kept giving me invaluable advice in my life and my academic endeavors.

I gratefully acknowledge Dr. Yeongjin Jang and Dr. Mike Rosulek for their training and education in system security and theoretical cryptography. I would like thank my committee members, Dr. Jean-François Biasse, Dr. Morris Chang, Dr. Jay Ligatti, Dr. Xinming (Simon) Ou and Dr. Mike Rosulek, for their invaluable advice and suggestions to improve this dissertation.

I would like to thank all my colleagues at Oregon State University (OSU) and University of South Florida (USF), Gungor Basa, Rouzbeh Behnia, Gabriel Hackebeil Daniel Lin, Ceyhun Ozkaptan, Ozgur Ozmen, Peter Rindal, Efe Seyitoglu, Sadman Sakib, Ni Trieu, for their supports. I am also grateful to all my friends at OSU and USF, who were always be there for me as a major source of support and encouragement when things got gloomy.

Finally, I would like to express my deepest gratefulness to my family, especially my wife, Tam Nguyen, for their endless love, support, encouragement and patience throughout my academic career, all of which cannot be described by words but have to be felt by heart.

Table of Contents

List of Tables

List of Figures

Abstract

Information systems generally involve storage and analytics of large-scale data, many of which may be highly sensitive (*e.g.*, personal information, medical records). It is vital to ensure that these systems not only provide essential functionalities at large scale efficiently but also achieve a high level of security against cyber threats. However, there are significant research challenges in offering security and privacy for such information systems while preserving their original functionalities (*e.g.*, search, analytics) effectively. Hence, there is a critical need for efficient cryptographic protocols that can address data privacy *vs.* utilization dilemma for real-life applications.

In this dissertation, we introduce a new series of privacy-enhancing technologies toward enabling breach-resilient and functional information systems. We focus on privacy-preserving data outsourcing applications featuring critical functionalities such as data query, accessibility and analytics. Specifically, we designed new dynamic searchable encryption schemes that permit the client to perform encrypted search and update queries on the encrypted data. We proposed new distributed oblivious access frameworks that allow the client to access and compute over the outsourced data efficiently without leaking the access pattern, thereby achieving a very high level of privacy in the presence of powerful adversaries. Finally, we built several privacy-preserving data storage and query platforms, which harness Trusted Execution Environment to enable critical functionalities (*e.g.*, search, update, concurrent access), security (*e.g.*, access control, integrity) and privacy properties (*e.g.*, access pattern obliviousness) in a highly efficient manner (*i.e.*, high throughput, low delay).

Chapter 1: Introduction

Data privacy *vs.* utilization is one of the most fundamental dilemmas in many applications ranging from personal data outsourcing to large-scale breach-resilient infrastructures. Recent cyberattacks targeting online applications (*e.g.*, Apple iCloud, Equifax, British Airways), in which the privacy of millions of customers was compromised, have demonstrated the importance of preserving data secrecy on the untrusted execution environment. While standard encryption techniques (*e.g.*, AES) can offer data confidentiality, they also prevent the user from performing even simple operations (*e.g.*, search/update) on the encrypted data, thereby diminishing the data usability. On the other hand, although there exist functional encryption techniques such as homomorphic encryption [69] and garbled circuit [188] that can enable data confidentiality and secure computation simultaneously, they are mainly designed for general purposes. These techniques may incur high communication and computation overhead, which may not be feasible for some real-world applications that have specific resource constraints and functionality requirements.

To enable some fundamental functionalities (*e.g.*, search/update) while preserving user privacy, recent studies focus on designing either novel cryptographic protocols with provable security (*e.g.*, searchable encryption) [35, 36, 49, 109], or encrypted queries that can be compliant with the legacy (database) infrastructures (*e.g.*, mySQL, mongoDB) [147]. Despite many efforts, all these techniques have been shown to be vulnerable against many practical attacks that exploited the way how the user access their own data on the untrusted memory (*e.g.*, access pattern) [34, 103, 124, 137, 148, 190]. To seal such a leakage, several secure access techniques such as Oblivious RAM [75] (ORAM) and

Private Information Retrieval [45] (PIR) were proposed. Despite their very high level of privacy and security guarantee, all these techniques incur high communication/computation overhead, which was shown costly to be used in many large-scale applications [20, 136].

Our main research objective is to address the security *vs.* functionality (*e.g.*, private search, update, access, analytics) dilemma of certain applications by creating new cryptographic schemes, which can meet security requirements, while respecting the original functionalities along with optimizations. This generally involves a three-step research process as follows. We first determine the critical security requirements of real-world applications and then examine their unique characteristics (*e.g.*, architecture, performance, system requirements). Second, we seek synergies among various cryptographic primitives to fill the needs of such applications regarding their special constraints while guaranteeing provable security. Finally, we fully implement the proposed techniques and strictly evaluate their performance on a myriad of real-world infrastructures.

We outline our contributions toward enabling privacy-preserving and functional information systems in the following section.

1.1 Contributions

In this dissertation, we propose a series of privacy-enhancing techniques, which features a high level of security and privacy while, at the same time, offering mandatory functionalities (*e.g.*, search, update, analytics) for critical cyberinfrastructures. To make our techniques efficient and more practical to be used in practice, we explore a broad range of system architecture spanning from the standard (client-server) model to the distributed setting and Trusted Execution Environments (TEE). We highlight our proposed schemes and their properties as follows.

1. *New Efficient Dynamic Searchable Encryption Schemes:* We develop several new searchable encryption techniques called IM-DSSE [97] (see §3.4), FS-DSSE [143] (see §3.5) with rigorous security analyses and full-fledged implementations. One of the most notable security features is that they are among the first to offer forward privacy, backward privacy and size obliviousness, all of which are mandatory to prevent statistical analysis attacks. We deployed all the proposed schemes on real cloud platforms, and the experimental results confirmed their significant advantages over the state-of-the-art in various performance and security metrics.

2. *Efficient Distributed Oblivious Access Techniques:* While Oblivious RAM (ORAM) [75] can hide the access pattern leakage in many applications (*e.g.*, searchable encryption, data outsourcing, secure CPU, multi-party computation, blockchains), state-of-the-art ORAM schemes either incur a logarithmic communication overhead [169, 179] or expensive cryptographic operations such as homomorphic encryption [54], both of which were shown costly in the context of data outsourcing [20, 89, 136]. Toward making ORAM more practical, we have investigated ORAM in various settings including distributed computation and TEE. We first construct S^{3}ORAM[89, 95], a multi-server ORAM framework based on secret sharing (see §4.4). S^{3}ORAM is among the first distributed ORAM that offers the most desirable properties for data outsourcing including low delay, low client-server bandwidth and small storage overhead without relying on costly cryptographic primitives such as homomorphic encryption. We further develop MACAO [87], a multi-server ORAM framework, which not only inherits all the desired performance properties from S^{3}ORAM but also offers security guarantee against malicious adversaries (see §4.5). We fully implemented both S^{3}ORAM and MACAO and the experiments on real-cloud platforms indicated that they were one order of magnitude faster than most efficient (single-server) ORAMs

featuring logarithmic bandwidth, and at least three orders of magnitude faster than HE-based ORAM alternatives featuring constant bandwidth.

3. *Oblivious Data Structures for Searchable Encryption and Database Services:* Access pattern is one of the most fundamental leakages in searchable encryption, on which many attacks have exploited to compromise the data and keyword privacy. We develop a series of oblivious data structures for searchable encryption, which can conceal/mitigate the search/update pattern leakages on the encrypted index while retaining the efficiency of search and update functionalities. We propose DOD-DSSE, which breaks the linkability between consecutive search/update queries by distributing the encrypted index across two non-colluding servers (see §4.6.1). This strategy prevents statistical attacks from exploiting the public information about the keyword frequency while achieving a high level of efficiency. We further develop ODSE framework [93, 94], which offers fully oblivious search and update operations on the encrypted index and robustness against the malicious adversaries with information-theoretic security (see §4.6.2). This is achieved by exploiting some unique properties of searchable encryption along with synergizing various efficient oblivious access primitives. We further propose several oblivious data structures called OMAT and OTREE [88], which are specially designed to be effectively integrated with state-of-the-art ORAM schemes to enable oblivious queries on legacy database management systems (*e.g.*, MongoDB, SQL) (see §4.6.3). The experimental results demonstrated the performance of our proposed techniques, compared with the state-of-the-art when tested on commodity cloud platforms (*i.e.*, Amazon EC2).

4. *Hardware-Supported Oblivious Storage and Query Platforms:* We also explore secure hardware techniques to build privacy-preserving and functional data storage systems. We designed POSUP platform [91], which enables oblivious search and update functionalities with very low

latency for personal data outsourcing, by synergizing Intel SGX, efficient ORAM paradigms and oblivious data structures (see §5.4). We fully implemented POSUP and intensively benchmarked its performance on a very large dataset (*i.e.*, full Wikipedia dataset) with commodity hardware. The experimental results confirmed the efficiency of POSUP, where it was up to three orders of magnitude faster than the state-of-the-art. Finally, we build MOSE [86], an oblivious storage platform using Intel SGX to enable private concurrent access for multiple users sharing a common database (see §5.5).

1.2 Dissertation Organization

We organize this dissertation as follows. We first present common notation being used throughout the rest of the dissertation in §2. In §3, we present a general overview of searchable encryption as well as our proposed IM-DSSE framework [97] (§3.4) and FS-DSSE scheme [143] (§3.5) in detail. In §4, we discuss ORAM and present our proposed S^{3}ORAM framework [89] (§4.4) and MACAO frameworks §4.5 in detail. We also present our proposed oblivious data structures techniques in §4.6 including DOD-DSSE (§4.6.1), ODSE (§4.6.2) and OMAT (§4.6.3). In §5, we present the overview of hardware-supported privacy-preserving information processing systems followed by our proposed POSUP platform [91] (§5.4) and MOSE platform [86] (§5.5) in detail. Finally, §6 concludes this dissertation and discusses our future work.

Chapter 2: Notation

We start by defining notation being frequently used in this dissertation. We denote $||$ and $\oplus$ as the concatenation and the Exclusive-OR (XOR) operation, respectively. $|x|$ denotes the size of x. Given a bit b, $\neg b$ means the complement of b. $\lfloor x \rfloor$ and $\lceil x \rceil$ denote the floor and the ceiling of x, respectively. $\langle \cdot \rangle_{\mathsf{bin}}$ denotes the binary representation. $[N]$ means $\{1, \ldots, N\}$. We denote $\{0,1\}^*$ as a set of binary strings of any finite length. $x \xleftarrow{\$} \mathcal{S}$ denotes that x is randomly and uniformly selected from set $\mathcal{S}$. $|\mathcal{S}|$ denotes the cardinality of set $\mathcal{S}$. $\mathcal{S} \setminus \{x\}$ (or $\mathcal{S} - \{x\}$) denotes $x \in \mathcal{S}$ is removed from $\mathcal{S}$. $(x_1, \ldots, x_l) \xleftarrow{\$} \mathcal{S}$ means $(x_1 \xleftarrow{\$} \mathcal{S}, \ldots, x_l \xleftarrow{\$} \mathcal{S})$.

Given $\mathbf{u}$ and $\mathbf{v}$ as vectors with the same length, $\mathbf{u} \cdot \mathbf{v}$ denotes the dot product of $\mathbf{u}$ and $\mathbf{v}$. Given an n-dimensional vector $\mathbf{u}$ and a matrix $\mathbf{I}$ of size $n \times m$, $\mathbf{v} = \mathbf{u} \cdot \mathbf{I}$ denotes the matrix product of $\mathbf{u}$ and $\mathbf{I}$ resulting in an m-dimensional vector $\mathbf{v}$. $\mathbf{I}[i][*]$ (or $\mathbf{I}[i,*]$) and $\mathbf{I}[*][j]$ (or $\mathbf{I}[*,j]$) denote accessing the row i and column j of matrix $\mathbf{I}$, respectively. $\mathbf{A}[i,j]$ denotes accessing the element at row i and column j of $\mathbf{A}$. If $\mathbf{I}$ is a matrix then $\mathbf{I}_u$ denotes the row indexed by u (*i.e.*, $\mathbf{I}_u = \mathbf{I}[u,*]$). We abuse this notation to also indicate a whole column indexed by u (*i.e.*, $\mathbf{I}_u = \mathbf{I}[*,u]$). This abuse of notation simplifies somewhat the presentation of our schemes. $\mathbf{I}[*, a \ldots b]$ (or $\mathbf{I}[*][a \ldots b]$) denotes accessing columns from a to b of matrix $\mathbf{u}[i]$ denotes accessing the i-th component of $\mathbf{u}$. $\mathbf{I}[*][i \ldots j]$ denotes accessing the columns from i to j of matrix $\mathbf{I}$.

We denote an encryption scheme with Indistinguishability against Chosen Plaintext Attack (IND-CPA) as a triplet $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, where $k \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)$ generates a symmetric key k given a security parameter 1^λ; $c \leftarrow \mathcal{E}.\mathsf{Enc}_k(M)$ returns the ciphertext c of M encrypted with key k;

$M \leftarrow \mathcal{E}.\mathsf{Dec}_k(c)$ returns the plaintext M of c, which is previously encrypted by k. Given a counter u, we denote $c \leftarrow \mathsf{Enc}_k(M, u)$ as IND-CPA encryption scheme, which takes as input a secret key k, the counter u and a message M and returns a ciphertext c. We denote $M \leftarrow \mathsf{Dec}_k(c, u)$ as IND-CPA decryption scheme, which takes as input a key k, the counter u and ciphertext c, and returns message M.

We denote a Pseudo Random Function (PRF) is a polynomial-time computable function, which is indistinguishable from a true random function by any PPT adversary We denote a keyed Pseudo Random Function (PRF) as $\tau \leftarrow G_k(x)$, which takes as input a secret key $k \overset{\$}{\leftarrow} \{0,1\}^{\kappa}$ and a string x, and returns a token/key r. We also denote $\tau \leftarrow \mathsf{KDF}(x)$ as a key derivation function, which takes as input an arbitrary string $x \in \{0,1\}^*$ and outputs a key τ. We denote $H : \{0,1\}^{|x|} \rightarrow \{0,1\}$ as a Random Oracle (RO), which takes an input x and returns a bit.

We denote a finite field as $\mathbb{F}_p$, where p is a prime. Unless otherwise stated, $a \cdot \mathbf{b}$ (or $a\mathbf{b}$) denotes the scalar multiplication, and all arithmetic operations are performed over $\mathbb{F}_p$.

Chapter 3: Searchable Encryption[1]

3.1 Introduction

The advent of cloud storage and computing platform provides vast benefits to the human society and IT industry. One of the most eminent cloud services is Storage-as-a-Service (SaaS), which offer a sophisticated infrastructure for millions of users, ranging from individuals to large-scale businesses, to store and access their own data remotely, thereby significantly reducing the data management and maintenance costs. Despite its effectiveness, SaaS also brings significant security and privacy concerns. Specifically, once a user outsources their personal data to the cloud, sensitive information (*e.g.*, email, bank transactions) might be exploited by a malicious party (*i.e.*, malware). Standard encryption such as Advanced Encryption Standard (AES) can enable data confidentiality, However, it also prevents the user from searching or updating information on the cloud and therefore, completely invalidates the benefits of using SaaS services. Such a data privacy *vs.* utilization dilemma poses a critical research challenge in enabling the security and privacy while, at the same time, retaining the functionality and efficiency of the underlying cloud services.

Searchable Symmetric Encryption (SSE) [49] permits a user to encrypt data in such a way that they can later perform keyword searches on. These encrypted queries are performed via "search tokens" [165] over an encrypted index, which represents the relationship between search token (keywords) and encrypted files. A prominent application of SSE is to enable privacy-preserving keyword search on the cloud (*e.g.*, Amazon S3), where the data owner can outsource a collection of

[1] This chapter was published in [97, 143]. Permission is included in Appendix A.

encrypted files and perform keyword searches on it without revealing the file and query contents [109]. Preliminary SSE schemes (*e.g.*, [49, 163]) only provide search-only functionality on static data (*i.e.*, no dynamism), which strictly limits their applicability due to the lack of update capacity. Later, several Dynamic SSE (DSSE) schemes [35, 109] were proposed that permit the user to add and delete files after the system is set up. As a trade-off between security efficiency, there is *no* single DSSE scheme that can outperform *all* the other alternatives in terms of *all* the metrics: privacy (*e.g.*, information leakage), performance (*e.g.*, search, update delay), storage efficiency and query functionality (*e.g.*, range, boolean).

Although a number of DSSE schemes have been introduced in the literature, most of them only provide a theoretical analysis and a prototype implementation. The lack of experimental performance evaluations on real platforms creates a challenge in assessing the application and performance of proposed DSSE schemes, as the impacts of hidden computation costs, multi-round communication delay and storage blowup might be overlooked. Moreover, several studies have shown that the most efficient (sublinear) DSSE schemes [35, 82] leak significant information and are vulnerable to statistical inference analysis [34, 148, 190]. For instance, Zhang *et al.* [190] has demonstrated a file-injection attack strategy, in which the semi-honest adversary can recover all keywords being searched or updated in the context of applying DSSE for personal email. It has been identified that the forward-privacy is an imperative security feature for modern DSSE constructions to mitigate the impact of the attack. Several forward-secure DSSE schemes with an optimal asymptotic complexity have been proposed, they incur either high delay due to public-key operations [26], or high storage blow-up at both client and server side [165, 189].

Therefore, it is vital to develop new DSSE schemes that can achieve a high level of security with a well-quantified information leakage, while maintaining a performance and functionality balance between the search and update operations. More importantly, it is critical that the performance of proposed DSSE schemes should be experimentally evaluated in a realistic cloud environment with various parameter settings, rather than relying on asymptotic results. The investigation of alternative data structures and their optimized implementations on commodity hardware seem to be the key factors towards achieving these objectives.

3.2 Related Work

SE has been proposed in both symmetric [97, 109, 163] (known as SSE) and asymmetric [15, 16, 24] (commonly known as PEKS) settings. In this dissertation, we mainly focus on SSE techniques. SSE was first introduced by Song *et al.* [163]. Curtmola *et al.* [49] proposed a sublinear SSE scheme and introduced the security notion for SSE called *adaptive security against chosen-keyword attacks (CKA2)* (see Definition 1). Refinements of [49] have been proposed, which offer extended functionalities [33, 171]. However, the static nature of those schemes limited their applicability to applications that require dynamic file collections. Kamara *et al.* were among the first to develop a DSSE scheme in [109] that could handle dynamic file collections via an encrypted index. However, it leaks significant information for updates and it is not parallelizable. Kamara *et al.* [108] proposed a DSSE scheme, which leaks less information than that of [109] and is parallelizable. Recently, a series of new DSSE schemes [26, 35, 82, 107, 138, 165] have been proposed, which offer various trade-offs between security, functionality and efficiency properties such as small leakage [165], scalable searches with various query and data types [36, 65, 107, 170, 175–177], or high efficiency [138]. Inspired by the work from [35], Kamara *et al.* in [107] proposed a new sublinear DSSE scheme, which supports more complex queries such as disjunctive and boolean queries.

Zhang *et al.* in [190] later showed that new DSSE constructions should offer the forward-privacy property to mitigate the impact of practical attacks. Several forward-private DSSE schemes achieving high efficiency in terms of asymptotic complexity and actual performance have been proposed. Specifically, Bost *et al.* in [26] proposed Sophos, which offers forward-privacy using an asymmetric primitive (*i.e.*, trapdoor permutation). Rizomiliotis *et al.* in [153] leveraged ORAM techniques [169] (see §4) to enable forward-privacy. Recently, several forward-private DSSE schemes only relying on symmetric primitives have been proposed [58, 115, 118, 164], some of which offer parallelism [58, 115, 118],and improved I/O access with computation efficiency using a caching strategy [115, 118, 164]. For example, Lai *et al.* [118] modeled the relationship between keywords and files in DSSE as bipartite graphs. The authors also proposed a novel data structure called *cascaded triangles*, which offers parallelism and efficient update (add/delete). Kim *et al.* [115] leveraged two hash tables to integrate forward index and inverted index together in the form of encrypted index, which offers efficient update with direct deletion. Several forward-private DSSE schemes, which offer extended query functionalities such as boolean query [107], similarity search [177] were also proposed. Bost *et al.* [27] proposed some (single-keyword) DSSE schemes that achieve both forward-privacy and backward-privacy with optimal asymptotic complexity using asymmetric primitives.

Due to the deterministic keyword-file relationship, most traditional DSSE schemes (*including our framework*) leak search and access patterns (to be defined in Definition 2), which are vulnerable to statistical inference attacks [34, 103, 124, 148, 190]). Several DSSE schemes have been proposed to deal with such leakages [25]) but they are neither efficient nor provably secure. ORAM [169] or Private Information Retrieval (PIR) [45, 73] can hide search and access patterns in DSSE. However, its cost is still high to be applied to DSSE in practice [136].

3.3 Preliminaries

The security notion for DSSE is *Dynamic adaptive security against Chosen-Keyword Attacks (CKA2)* security [108, 109, 165, 189]), which captures information leakage via leakage functions characterizing the information leakage due to search and update operations (see [108, 165, 189] for the details).

Definition 1 (IND-CKA2 Security [49, 109]). Let $\mathcal{A}$ be a stateful adversary and $\mathcal{S}$ be a stateful simulator. Consider the following probabilistic experiments:

- $\mathsf{Real}_{\mathcal{A}}(\kappa)$: The challenger executes $\mathcal{K} \leftarrow \mathsf{Gen}(1^{\kappa})$. $\mathcal{A}$ produces $(\delta, \mathbf{f})$ and receives $(\gamma, \mathcal{C}) \leftarrow \mathsf{Enc}_{\mathcal{K}}(\delta, \mathbf{f})$ from the challenger. $\mathcal{A}$ makes polynomially bounded number of adaptive queries $\mathsf{Query} \in (w, f_{id}, f_{id'})$ to the challenger. If $\mathsf{Query} = w$ is a keyword search query then $\mathcal{A}$ receives a search token $\tau_w \leftarrow \mathsf{SearchToken}(\mathcal{K}, w)$ from the challenger. If $\mathsf{Query} = f_{id}$ is a file addition query then $\mathcal{A}$ receives an addition token $(\tau_f, c) \leftarrow \mathsf{AddToken}(\mathcal{K}, f_{id})$ from the challenger. If $\mathsf{Query} = f_{id'}$ is a file deletion query then $\mathcal{A}$ receives a deletion token $\tau'_f \leftarrow \mathsf{DeleteToken}(\mathcal{K}, f_{id'})$ from the challenger. Eventually, $\mathcal{A}$ returns a bit b that is the output of the experiment.

- $\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa)$: $\mathcal{A}$ produces $(\delta, \mathbf{f})$. Given $\mathcal{L}_1(\delta, \mathbf{f})$, $\mathcal{S}$ generates and sends $(\gamma, \mathcal{C})$ to $\mathcal{A}$. $\mathcal{A}$ makes a polynomial number of adaptive queries $\mathsf{Query} \in (w, f_{id}, f_{id'})$ to $\mathcal{S}$. For each query, $\mathcal{S}$ is given $\mathcal{L}_2(\delta, \mathbf{f}, w, t)$. If $\mathsf{Query} = w$ then $\mathcal{S}$ returns a simulated search token τ_w. If $\mathsf{Query} = f_{id}$ or $\mathsf{Query} = f_{id'}$, $\mathcal{S}$ returns a simulated addition token τ_f or deletion token τ'_f,respectively. Eventually, $\mathcal{A}$ returns a bit b that is the output of the experiment.

A DSSE is said to be $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$-secure against adaptive chosen-keyword attacks (CKA2-security) if for all PPT adversaries $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that

$$|\Pr[\mathsf{Real}_{\mathcal{A}}(\kappa) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa) = 1]| \quad \leq \quad \mathsf{neg}(\kappa)$$

All existing DSSE schemes (e.g. [35, 36, 82, 138, 165, 189] with Dynamic CKA2 security leak *data structure-access pattern* [168] defined as follows:

Definition 2 (Access Pattern). Access pattern is a data request sequence $\overrightarrow{\sigma_b} = \{\mathsf{op}_i^{(b)}, u_i^{(b)}, \mathsf{data}_i^{(b)}\}_{i=1}^q$ of length q over an encrypted memory region on server S_b during search and update operations, where $\mathsf{op}_i^{(b)} \in \{\mathsf{read}(u_i^{(b)}, \mathsf{data}_i^{(b)}), \mathsf{write}(u_i^{(b)}, \mathsf{data}_i^{(b)})\}$, $u_i^{(b)}$ is the address identifier on S_b to be read or written and $\mathsf{data}_i^{(b)}$ is the data located at $u_i^{(b)}$ to be read or written on S_b.

Access pattern leaks search patterns and update patterns, which can be defined as follows:

Definition 3 (Search and Update Pattern). Search pattern indicates if the same keyword has been previously searched. Given a query on w at time t, the search pattern is a binary vector of length t with 1 at location i if the search $i \leq t$ was for w, 0 otherwise. *Update pattern* indicates information being leaked during an update operation with different levels, in that level 1 (as defined in [189]) leaks least information, which is similar to search pattern.

3.4 IM-DSSE: Searchable Encryption Framework for Critical Cloud Services

Towards filling the gaps between theory and practice in DSSE research community, we introduce IM-DSSE, a fully-implemented Incidence Matrix-based DSSE framework which favors desirable properties for realistic privacy-critical cloud systems including high security against practical attacks and low end-to-end delay. In this framework, we provide the full-fledged implementation of our DSSE schemes, which are specially designed to meet various application requirements and cloud data storage-as-a-service infrastructures in practice.

IM-DSSE offers ideal features for privacy-critical cloud systems as follows.

- *Highly secure against file-injection attacks:* IM-DSSE offers *forward privacy* (see [165] or §3.4.5 for definition) which is an imperative security feature to mitigate the impact of practical file-injection attacks [26, 190]. Only a limited number of DSSE schemes offer this property (*i.e.*, [26, 58, 115, 118, 153, 164, 165, 177]), some of which incur high client storage with costly update (*e.g.*, [165]) or high delay, due to oblivious access techniques (*e.g.*, [153]) and public-key operations (*e.g.*, [26]). Additionally, IM-DSSE offers *size-obliviousness* property, where it hides all size information involved with the encrypted index and update query including *(i)* update query size (*i.e.*, number of unique keywords in the updated file); *(ii)* and the number of keyword-file pairs in the database. One of the IM-DSSE variants achieves *backward privacy* defined in [165]. We notice that Bost *et al.* in [27] have recently proposed a new DSSE scheme that can achieve all these security properties with padding. This scheme leverages asymmetric primitives (*e.g.*, puncturable encryption [79]), which might incur high computation cost. Our scheme relies on symmetric primitives but with the cost of an extra communication overhead.

- *Updates with improved features*: (i) IM-DSSE allows to *directly* update keywords of an existing file without invoking the file delete-then-add operation sequence. The update in IM-DSSE also leaks minimal information, where it does not leak timing information (*i.e.*, all updates take the same amount of time) and how many keywords are being added/deleted in the updated file. (ii) The encrypted index of our schemes does not grow with update operations and, therefore, it does not require re-encryption due to frequent updates. This is more efficient than some alternatives (*e.g.*,[165]) in which the encrypted index can grow linearly with the number of deletions.

- *Fully parallelizable*: IM-DSSE also supports parallelization (as in [35, 58, 115, 118]) for both update and search operations and, therefore, it takes full advantage of modern computing architecture to minimize the delay of cryptographic operations. Experiments on Amazon cloud indicates that the search latency of our framework is highly practical and mostly dominated by the client-server communication (see §3.4.6).
- *Detailed experimental evaluation and open-source framework*: We deployed IM-DSSE in a realistic cloud environment (*i.e.*, Amazon EC2) to assess the practicality of our framework. We experimented with different database sizes and investigated the impacts of network condition and storage unit on the overall performance. We also evaluated the performance of IM-DSSE on a resource-limited mobile client. We gave a comprehensive cost breakdown analysis to highlight the main factors contributing to the overall delay in all these settings. We have released the implementation of our framework to public to provide opportunities for broad adaptation and testing [85] (see §3.4.6).

Figure 3.1 illustrates the overview of our IM-DSSE framework for file-storage applications.

3.4.1 System and Threat Models

3.4.1.1 System Model

Our system model comprises one server and one client, which can be mobile resource-constrained (*e.g.*, cell phone) as illustrated in Figure 3.1. Our model can be extended into multiple clients that share the same keys.

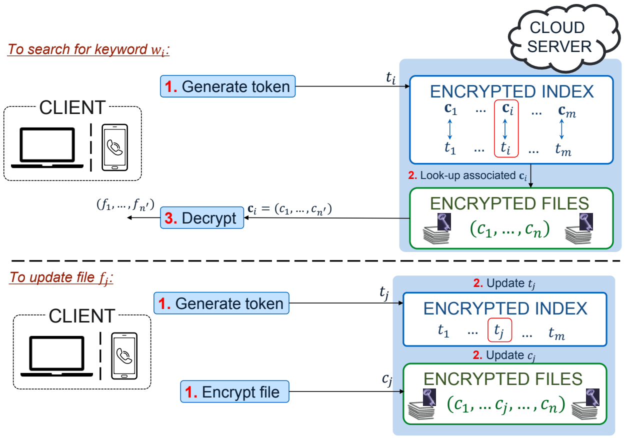


Figure 3.1: IM-DSSE framework for file-storage services.

3.4.1.2 Threat Model

In our threat model, the client is trusted and the server is honest-but-curious, meaning that it follows the protocol faithfully but attempts to extract sensitive information during the client's search/update operations. The server can know the encrypted files, the encrypted index and record the transcripts of the protocol. Our objective is to allow the client to perform search and update operations in a secure manner, in which files can be securely retrieved/updated while leaking least information to the server. Specifically, once the server is compromised, the client should only leak the query content and no file contents or specific keywords are ever compromised. Since search and update tokens are deterministic, our IM-DSSE framework leaks *search*, and *file-access patterns* as in all other DSSE schemes. We present the formal security model in §3.4.5.

3.4.2 Data Structures

Our encrypted index is an incidence matrix $\mathbf{I}$, in which $\mathbf{I}[i, j].v \in \{0, 1\}$ stores the (encrypted) relationship between keyword indexing at row i and file indexing at column j, and $\mathbf{I}[i, j].\mathsf{st} \in \{0, 1\}$ stores a bit indicating the state of $\mathbf{I}[i, j].v$. Particularly, $\mathbf{I}[i, j].\mathsf{st}$ is set to 1 or 0 if $\mathbf{I}[i, j].v$ is last

```
$\mathcal{K} \leftarrow \mathsf{IM\text{-}DSSE_{main}.Gen}(1^{\kappa})$: Given security parameter $\kappa$, generate secret key $\mathcal{K}$
  $k_1 \leftarrow \mathcal{E}.\mathsf{Gen}(1^{\kappa})$ and $(k_2, k_3) \overset{\$}{\leftarrow} \{0,1\}^{\kappa}$
  return $\mathcal{K}$, where $\mathcal{K} \leftarrow \{k_1, k_2, k_3\}$

$f \leftarrow \mathsf{IM\text{-}DSSE_{main}.Dec}_{\mathcal{K}}(c)$: Decrypt encrypted file $c$ with key $\kappa$
  $f \leftarrow \mathsf{Dec}k_1 c', y||c$ where $c \leftarrow T_f[y].\mathsf{ct}$, $(c', y) \leftarrow c$
  return $f$

$(\gamma, \mathcal{C}) \leftarrow \mathsf{IM\text{-}DSSE_{main}.Enc}_{\mathcal{K}}(\delta, \mathbf{f})$: Given index $\delta$ and plaintext files $\mathbf{f}$, generate corresponding encrypted index $\gamma$ and encrypted files $\mathcal{C}$
  $T_w[i].\mathsf{ct} \leftarrow 1$, $T_f[j].\mathsf{ct} \leftarrow 1$, for $0 \le i \le m, 0 \le j \le n$
  $\mathbf{I}[*,*].\mathsf{st} \leftarrow 0$ and $\delta[*,*] \leftarrow 0$
  Extract $(w_1, \ldots, w_{m'})$ from $\mathbf{f} = \{f_{id_1}, \ldots, f_{id_{n'}}\}$
  for $i = 1, \ldots, m'$ do
      $s_{w_i} \leftarrow G_{k_2}(w_i)$, $x_i \leftarrow T_w(s_{w_i})$
      for $j = 1, \ldots, n'$ do
          if $w_i$ appears in $f_{id_j}$ then
              $s_{id_j} \leftarrow G_{k_2}(id_j)$ and $y_j \leftarrow T_f(s_{id_j})$
              $\delta[x_i, y_j] \leftarrow 1$
  for $i = 1, \ldots, m$ do
      $r_i \leftarrow G_{k_3}(i||\overline{c}_i)$, where $\overline{c}_i \leftarrow T_w[i].\mathsf{ct}$
      for $j = 1, \ldots, n$ do
          $\mathbf{I}[i,j] \leftarrow \delta[i,j] \oplus H(r_i||j||c_j)$, where $c_j \leftarrow T_f[j].\mathsf{ct}$
  for $j = 1, \ldots, n'$ do
      $c_j \leftarrow (c'_j, y_j)$, where $c'_j \leftarrow \mathsf{Enc}k_1 f_{id_j}, y_j||u_{y_j}$
  return $(\gamma, \mathcal{C})$, where $\gamma \leftarrow (\mathbf{I}, T_f)$ and $\mathcal{C} \leftarrow \{c_1, \ldots, c_{n'}\}$
```

Figure 3.2: $\mathsf{IM\text{-}DSSE_{main}}$ detailed algorithms.

accessed by update or search, respectively. For simplicity, we write $\mathbf{I}[i,j]$ to denote $\mathbf{I}[i,j].v$, and be explicit about the state bit as $\mathbf{I}[i,j].\mathsf{st}$.

The encrypted index $\mathbf{I}$ is augmented by two static hash tables T_w and T_f that associate a keyword and file to a unique row and a column, respectively. Specifically, T_f is a file static hash table whose key-value pair is $(s_{id_j}, \langle y_j, c_j \rangle)$, where $s_{id_j} \leftarrow G_{k_2}(id_j)$ for file with identifier id_j, column index $y_j \in \{1, \ldots, n\}$ is equivalent to the index of s_{id_j} in T_f and c_j is a counter value. We denote access operations by $y_j \leftarrow T_f(s_{id_j})$ and $c_j \leftarrow T_f[y_j].\mathsf{ct}$. T_w is a keyword static hash table whose key-value pair is $\{s_{w_i}, \langle x_i, c_i \rangle\}$, where token $s_{w_i} \leftarrow G_{k_2}(w_i)$ for keyword w_i, row index $x_i \in \{1, \ldots, m\}$ is equivalent to the index of s_{w_i} in T_w and c_i is a counter value. We denote access operations by $x_i \leftarrow T_w(s_{w_i})$ and $c_i \leftarrow T_w[x_i].\mathsf{ct}$. All counter values are incremental and initially set to 1. So, the client state information is in the form of T_w and T_f, both of which offer (on average) $\mathcal{O}(1)$ access time.

3.4.3 IM-DSSE Main Scheme

We present the detailed algorithmic construction for the main scheme (denoted $\mathsf{IM\text{-}DSSE}_{\mathsf{main}}$) in IM-DSSE framework in Figure 3.2, Figure 3.3 and Figure 3.4, which consists of nine algorithms with high-level ideas as follows.

3.4.3.1 Setup Algorithm

The client first executes $\mathsf{IM\text{-}DSSE}_{\mathsf{main}}.\mathsf{Gen}$ Algorithm to generate secret keys ($\mathcal{K}$). Based on the generated keys $\mathcal{K}$, the client executes $\mathsf{IM\text{-}DSSE}_{\mathsf{main}}.\mathsf{Enc}$ Algorithm to create encrypted data structures to be outsourced to the cloud. In $\mathsf{IM\text{-}DSSE}_{\mathsf{main}}.\mathsf{Enc}$ Algorithm, it first extracts m' unique keywords $(w_1, \ldots, w_{m'})$ from n' files $\mathbf{f} = \{f_{id_1}, \ldots, f_{id_{n'}}\}$ with unique IDs $(id_1, \ldots, id_{n'})$ (step step 3). It then constructs an (unencrypted) incidence matrix δ (steps step 4–step 9) by setting each cell value $\delta[i, j]$ to $\{0, 1\}$, where i, j are the row and column indexes of keyword and file derived from their hash table indexes, respectively (steps step 5, step 13). Next, it encrypts each cell $\delta[i, j]$ with a unique $\langle$key-counter$\rangle$ pair, where the key (r_i) is uniquely derived for each row (i) from $\mathcal{K}$ (step step 11), and the counter (u_j) is distinct for each column (j) (step step 13). Finally, it encrypts each file in $\mathbf{f}$ resulting in encrypted files $\mathcal{C}$ (steps step 14–step 15). Once $\mathcal{C}$ and the encrypted matrix $(\mathbf{I})$ are constructed, the client sends them to the cloud server along with the file hash table (T_f).

```
$\tau_w \leftarrow$ IM-DSSE$_{\mathsf{main}}$.SearchToken$(\mathcal{K}, w)$: Generate search token $\tau_w$ from keyword $w$ and key $\mathcal{K}$
 1: $s_w \leftarrow G_{k_2}(w)$, $i \leftarrow T_w(s_w)$
 2: $\bar{c} \leftarrow T_w[i]$.ct, $r_i \leftarrow G_{k_3}(i||\bar{c})$
 3: if $\bar{c} = 1$ then
 4:     $\tau_w \leftarrow (i, r_i)$
 5: else
 6:     $\bar{r}_i \leftarrow G_{k_3}(i||\bar{c} - 1)$ and $\tau_w \leftarrow (i, r_i, \bar{r}_i)$
 7: $T_w[i]$.ct $\leftarrow \bar{c} + 1$
 8: return $\tau_w$
```

```
$(\mathcal{I}_w, \mathcal{C}_w) \leftarrow$ IM-DSSE$_{\mathsf{main}}$.Search$(\tau_w, \gamma)$: Given search token $\tau_w$ and encrypted index $\gamma$, return sets of file identifiers $\mathcal{I}_w$ and encrypted files $\mathcal{C}_w \subseteq \mathcal{C}$ matching with $\tau_w$
 1: for $j = 1, \ldots, n$ do
 2:     $c_j \leftarrow T_f[j]$.ct
 3:     if $(\tau_w = (i, r_i)$ or $\mathbf{I}[i, j]$.st $= 1)$ then
 4:         $\mathbf{I}'[i, j] \leftarrow \mathbf{I}[i, j] \oplus H(r_i||j||c_j)$
 5:         $\mathbf{I}[i, j]$.st $\leftarrow 0$
 6:     else
 7:         $\mathbf{I}'[i, j] \leftarrow \mathbf{I}[i, j] \oplus H(\bar{r}_i||j||c_j)$
 8:         $\mathbf{I}[i, j] \leftarrow \mathbf{I}'[i, j] \oplus H(r_i||j||c_j)$
 9: $l \leftarrow 0$
10: for each $j \in \{1, \ldots, n\}$ satisfying $\mathbf{I}'[i, j] = 1$ do
11:     $l \leftarrow l + 1$ and $y_l \leftarrow j$
12: $\mathcal{I}_w \leftarrow \{y_1, \ldots, y_l\}$
13: $\gamma \leftarrow (\mathbf{I}, T_f)$, $\mathcal{C}_w \leftarrow \{(c_{y_1}, y_1), \ldots, (c_{y_l}, y_l)\}$
14: return $(\mathcal{I}_w, \mathcal{C}_w)$
```

Figure 3.3: IM-DSSE$_{\mathsf{main}}$ detailed algorithms.

3.4.3.2 Search Protocol

To search for keyword w, the client executes IM-DSSE$_{\mathsf{main}}$.SearchToken Algorithm to generate a search token τ_w to be sent to the server. The token contains the row index (i) of w and row keys $(r_i, \bar{r}_i)$ derived from $\mathcal{K}$, i, and the row counter (u) (steps step 3–step 6). $\bar{r}_i$ and r_i are the old and new keys that are used to decrypt the row being searched for the first and latter times, respectively. Upon receiving τ_w, the server executes IM-DSSE$_{\mathsf{main}}$.Search Algorithm to decrypt the row and retrieve the search result. Specifically, if the cell $\mathbf{I}[i, j]$ is previously updated (indicated via the state bit $\mathbf{I}[i, j]$.st), or being searched for the first time (step step 3), it uses the new key (r_i) to decrypt the cell (step step 4) and set the state to be 0 (step step 5). Otherwise, it uses the old key $(\bar{r}_i)$ to decrypt the cell (step step 7) and re-encrypts it with the new key (r_i) (step step 8). Finally, the server determines column indexes j such that $\mathbf{I}[i, j] = 1$, and returns the corresponding j-labeled ciphertexts to the client (steps step 9–step 13). The client executes $\mathsf{IM\text{-}DSSE_{main}.Dec}$ Algorithm on each ciphertext to decrypt the files and obtain the search result.

3.4.3.3 Update Protocol

The client executes $\mathsf{IM\text{-}DSSE_{main}.AddToken}$ Algorithm to generate an addition token (τ_f). It gets the column index (j) of the file to be added from the file hash table (T_f) (step step 1), and then, derives fresh row keys from the keyword hash table (T_w) to be used for encrypting the column (steps step 2–step 3). It then extracts unique keywords ($w_1, \ldots w_t$) from the file, and constructs an (unencrypted) column $\bar{\mathbf{I}}[*, j]$ with values being set to $\{0, 1\}$ (steps step 4–step 6). Finally, it encrypts $\bar{\mathbf{I}}[*, j]$ with row keys (steps step 7–step 8) and the file with $\mathcal{K}$ (step step 9). The client sends τ_f containing the ciphertext and the encrypted column to the server. Upon receiving τ_f, the server executes $\mathsf{IM\text{-}DSSE_{main}.Add}$ Algorithm to update the column j and its state in $\mathbf{I}$ (step step 1). It increases the column counter (step step 2), and adds the ciphertext to the set of encrypted files (step step 3).

Similar to the file addition protocol, the client executes $\mathsf{IM\text{-}DSSE_{main}.DeleteToken}$ Algorithm to generate the deletion token, and the server executes $\mathsf{IM\text{-}DSSE_{main}.Delete}$ Algorithm to update the column and delete the file from the set of encrypted files.

Some existing schemes (*e.g.*, [138]) allow file addition/deletion, but do not permit updating keywords in an existing file *directly*. This can be easily achieved in our scheme as follows. Assume the client wants to update file f_{id} by adding (or removing) some keywords, they will prepare a new column $\mathbf{I}'[i, j] \leftarrow b_i$ for $1 \leq i \leq m$, where $b_i = 1$ if w_i is added and $b_i = 0$ otherwise and $j \leftarrow T_f(s_{id})$

$(\tau_f, c) \leftarrow \mathsf{IM\text{-}DSSE_{main}.AddToken}(\mathcal{K}, f_{id})$: Given key $\mathcal{K}$ and file f_{id}, generate addition token τ_f and ciphertext c of f_{id}

```
 1: $s_{id} \leftarrow G_{k_2}(id)$, $j \leftarrow T_f(s_{id})$, $T_f[j].\mathsf{ct} \leftarrow T_f[j].\mathsf{ct} + 1$ and $c_j \leftarrow T_f[j].\mathsf{ct}$
 2: for $i = 1, \ldots, m$ do
 3:     $r_i \leftarrow G_{k_3}(i||\bar{c}_i)$, where $\bar{c}_i \leftarrow T_w[i].\mathsf{ct}$
 4: Extract $(w_1, \ldots, w_t)$ from $f_{id}$ and set $\bar{\mathbf{I}}[*, j] \leftarrow 0$
 5: for $i = 1, \ldots, t$ do
 6:     $s_{w_i} \leftarrow G_{k_2}(w_i)$, $x_i \leftarrow T_w(s_{w_i})$, $\bar{\mathbf{I}}[x_i, j] \leftarrow 1$
 7: for $i = 1, \ldots, m$ do
 8:     $\mathbf{I}'[i, j] \leftarrow \bar{\mathbf{I}}[i, j] \oplus H(r_i||j||c_j)$
 9: $c \leftarrow (c', j)$, where $c' \leftarrow \mathsf{Enc}k_1 f_{id}, j||c_j$
10: return $(\tau_f, c)$ where $\tau_f \leftarrow (\mathbf{I}', j)$
```

$(\gamma', \mathcal{C}') \leftarrow \mathsf{IM\text{-}DSSE_{main}.Add}(\gamma, \mathcal{C}, c, \tau_f)$: Add addition token τ_f and ciphertext c to encrypted index γ and ciphertext set $\mathcal{C}$, resp.

```
1: Set $\mathbf{I}[i, j] \leftarrow \mathbf{I}'[i, j]$ and $\mathbf{I}[i, j].\mathsf{st} \leftarrow 1$, for $1 \leq i \leq m$
2: $T_f[j].\mathsf{ct} \leftarrow T_f[j].\mathsf{ct} + 1$
3: return $(\gamma', \mathcal{C}')$, where $\gamma' \leftarrow (\mathbf{I}, T_f)$ and $\mathcal{C}' \leftarrow \mathcal{C} \cup \{(c, j)\}$
```

$\tau'_f \leftarrow \mathsf{IM\text{-}DSSE_{main}.DeleteToken}(\mathcal{K}, f)$: Given key $\mathcal{K}$ and deleted file f_{id}, generate deletion token τ'_f

```
1: Execute steps step 1–step 3 of $\mathsf{IM\text{-}DSSE_{main}.AddToken}$ Algorithm to produce $(j, c_j, \langle r_1, \ldots, r_m \rangle)$ and increase
   $T_f[j].\mathsf{ct}$ to 1
2: for $i = 1, \ldots, m$ do
3:     $\mathbf{I}'[i, j] \leftarrow H(r_i||j||c_j)$
4: return $\tau'_f$, where $\tau'_f \leftarrow (\mathbf{I}', j)$
```

$(\gamma', \mathcal{C}') \leftarrow \mathsf{IM\text{-}DSSE_{main}.Delete}(\gamma, \mathcal{C}, \tau'_f)$: Update deletion token τ'_f to encrypted index γ' and delete a file from ciphertext set $\mathcal{C}'$

```
1: Set $\mathbf{I}[i, j] \leftarrow \mathbf{I}'[i, j]$ and $\mathbf{I}[i, j].\mathsf{st} \leftarrow 1$, for $1 \leq i \leq m$
2: $T_f[j].\mathsf{ct} \leftarrow T_f[j].\mathsf{ct} + 1$
3: return $(\gamma', \mathcal{C}')$, where $\gamma' \leftarrow (\mathbf{I}, T_f)$, $\mathcal{C}' \leftarrow \mathcal{C} \setminus \{(c, j)\}$
```

Figure 3.4: $\mathsf{IM\text{-}DSSE_{main}}$ detailed algorithms.

with $s_{id} \leftarrow G_{k_2}(id)$ as in $\mathsf{IM\text{-}DSSE_{main}.AddToken}$ algorithm (steps step 4-step 6). The rest of the algorithm remains the same.

3.4.3.4 Cost Analysis

For keyword search, $\mathsf{IM\text{-}DSSE_{main}}$ incurs n invocations of hash function H and n XOR operations. Although $\mathsf{IM\text{-}DSSE_{main}}$ has a linear search complexity, which is asymptotically less efficient than other DSSE schemes (*e.g.*, [26, 35]), we show in our experiments that, this impact is *insignificant* in practice for personal cloud usage with moderate database size where all optimizations are taken into account. Since IM-DSSE$_{\text{main}}$ is fully parallelizable, the search and update computation times can be reduced to n/p and m/p, respectively, where p is the number of processors. Therefore, cryptographic operations in IM-DSSE$_{\text{main}}$ only contribute a small portion to the overall end-to-end search delay, which is mostly dominated by the network communication latency between client and server. Notice that all sub-linear DSSE schemes [35, 165] are less secure and sometimes incur more costly updates than IM-DSSE$_{\text{main}}$. For file update, IM-DSSE$_{\text{main}}$ incurs m invocations of H and m XOR operations along with m bits of transmission.

IM-DSSE$_{\text{main}}$ costs $\Big(2m \cdot n + n \cdot (\kappa + |c|)\Big)$ bits of storage at the server for encrypted index $\mathbf{I}$ and file hash table T_f. At the client side, IM-DSSE$_{\text{main}}$ requires $(n+m)(\kappa + |c|) + 3\kappa$ bits for two hash tables T_w, T_f and secret key $\mathcal{K}$.

3.4.4 IM-DSSE Extended Schemes

We present extended schemes derived from IM-DSSE$_{\text{main}}$ presented above that IM-DSSE framework also supports.

3.4.4.1 IM-DSSE$_{\text{I}}$: Minimized Search Latency

In IM-DSSE$_{\text{main}}$, we encrypt each cell of $\mathbf{I}$ with a unique key-counter pair, which requires n invocations of H during keyword search. This might not be ideal for some applications that require extremely prompt response. Hence, we introduce an extended scheme called IM-DSSE$_{\text{I}}$, which aims at achieving a very low search latency with the cost of increasing update delay. Specifically, instead of encrypting the index bit-by-bit as in IM-DSSE$_{\text{main}}$ scheme, IM-DSSE$_{\text{I}}$ leverages b-bit block cipher encryption to encrypt b successive cells with the same key-counter pair. This is achieved by interpreting columns of $\mathbf{I}$ as $D = \lceil \frac{n}{b} \rceil$ blocks, each being IND-CPA encrypted using AES-CTR mode

```
$(\tau_f, c) \leftarrow \textsf{IM-DSSE}_{\textsf{I}}.\textsf{AddToken}(\mathcal{K}, f_{id})$: Generate addition token $\tau_f$ and ciphertext $c$ of $f_{id}$
$s_{id} \leftarrow G_{k_2}(id)$, $j \leftarrow T_f(s_{id})$, $l \leftarrow \lfloor \frac{j-1}{b} \rfloor$, $c_l \leftarrow \mathbf{u}[l]$
$a \leftarrow (l \cdot b) + 1$, $a' \leftarrow b \cdot (l + 1)$
Extract $(w_1, \ldots, w_t)$ from $f_{id}$
for $i = 1, \ldots, t$ do
    $s_{w_i} \leftarrow G_{k_2}(w_i)$, $x_i \leftarrow T_w(s_{w_i})$
Get from server $(\mathbf{I}[*, a \ldots a'])$ and $\mathbf{I}[*, l].\textsf{st}$
for $i = 1, \ldots, m$ do
    $\overline{c}_i \leftarrow T_w[i].\textsf{ct}$
    if $(\overline{c}_i > 1$ and $\mathbf{I}[i, l].\textsf{st} = 0)$ then
        $\overline{c}_i \leftarrow \overline{c}_i - 1$
    $r_i \leftarrow G_{k_3}(i||\overline{c}_i)^{\dagger}$
    $\mathbf{I}'[i, a \ldots, a'] \leftarrow \textsf{Dec}r_i\mathbf{I}[i, a \ldots a'], l||c_l$
$\mathbf{I}'[i, j] \leftarrow 0$ for $1 \leq i \leq m$ and $\mathbf{I}'[x_i, j] \leftarrow 1$ for $1 \leq i \leq t$
$\mathbf{u}[l] \leftarrow \mathbf{u}[l] + 1$, $c_l \leftarrow \mathbf{u}[l]$
for $i = 1, \ldots, m$ do
    if $(\overline{c}_i > 1$ and $\mathbf{I}[i, l].\textsf{st} = 0)$ then
        $r_i \leftarrow G_{k_3}(i||\overline{c}_i + 1)$
    $\overline{\mathbf{I}}[i, a \ldots a'] \leftarrow \textsf{Enc}r_i\mathbf{I}'[i, a \ldots a'], l||c_l$
$T_f[j].\textsf{ct} \leftarrow T_f[j].\textsf{ct} + 1$ and $u'_j \leftarrow T_f[j].\textsf{ct}$
$c \leftarrow (c', j)$ where $c' \leftarrow \textsf{Enc}k_1 f_{id}, j||u'_j$
return $(\tau_f, c)$ where $\tau_f \leftarrow (\overline{\mathbf{I}}, j)$

$(\gamma', \mathcal{C}') \leftarrow \textsf{IM-DSSE}_{\textsf{I}}.\textsf{Add}(\gamma, \mathcal{C}, c, \tau_f)$: Add addition token $\tau_f$ and ciphertext $\mathcal{C}$ to encrypted index $\gamma$ and ciphertext set $\mathcal{C}$, resp.
$l \leftarrow \lfloor \frac{j-1}{b} \rfloor$, $a \leftarrow (l \cdot b) + 1$, $a' \leftarrow b(l + 1)$
$\mathbf{I}[i, j'] \leftarrow \overline{\mathbf{I}}[i, j']$, for $1 \leq i \leq m$ and $a \leq j' \leq a'$
$\mathbf{u}[l] \leftarrow \mathbf{u}[l] + 1$ and $\mathbf{I}[*, l].\textsf{st} \leftarrow 1$
return $(\gamma', \mathcal{C}')$, where $\gamma' \leftarrow (\mathbf{I}, T_f)$ and $\mathcal{C}' \leftarrow \mathcal{C} \cup \{c\}$
```

† G should generate a suitable key for $\mathcal{E}$ (*e.g.*, 128-bit key for AES-CTR)

Figure 3.5: $\textsf{IM-DSSE}_{\textsf{I}}$ detailed algorithms.

with block cipher size b. The counter will be stored via a block counter array (denoted as $\mathbf{u}$) instead of $T_f[\cdot].c$ as in the main scheme. The update state is maintained for each block rather than each cell of $\mathbf{I}[i, j]$. Hence, $\mathbf{I}$ is decomposed into two matrices with different sizes: $\mathbf{I}.v \in \{0, 1\}^{m \times n}$ and $\mathbf{I}.\textsf{st} \in \{0, 1\}^{m \times D}$.

$\textsf{IM-DSSE}_{\textsf{I}}$ requires some algorithmic modifications from the main scheme. Figure 3.5 presents $\textsf{IM-DSSE}_{\textsf{I}}.\textsf{AddToken}$ Algorithm and $\textsf{IM-DSSE}_{\textsf{I}}.\textsf{Add}$ Algorithm for file addition procedure in $\textsf{IM-DSSE}_{\textsf{I}}$ (modifications for file deletion follow the same principle). Specifically, we substitute encryption and decryption using random oracle $H(r_i||j||u_j)$ with block cipher encryption $\mathsf{Enc}r_i\cdot, l||u'_l$ and $\mathsf{Dec}r_i\cdot, l||u'_l$, respectively, where u_l is a block counter (see steps step 12, step 20). Since $\mathbf{I}$ is encrypted by blocks, to update a column during the file update, the client needs to retrieve a whole block and its state from the server first (step step 6). The client then decrypts the block (steps step 7–step 12), updates a column within it (step step 13), re-encrypts the entire block (steps step 15–step 18), and finally sends the encrypted block to the server for replacement. So, the reduction of search cost increases the cost of communication overhead for the update as a trade-off.

The modifications for IM-DSSE$_\mathsf{I}$.Gen, IM-DSSE$_\mathsf{I}$.Enc, IM-DSSE$_\mathsf{I}$.SearchToken and IM-DSSE$_\mathsf{I}$.Search algorithms are straightforward. Only the underlying encryption is changed from random oracle to block cipher (*e.g.*, AES-CTR) as exemplified in IM-DSSE$_\mathsf{I}$.AddToken Algorithm. Hence, we will not present those algorithms in detail.

For keyword search, IM-DSSE$_\mathsf{I}$ requires n/b invocations of $\mathcal{E}$, which is theoretically b times faster than the main scheme. Given the CTR mode, the search time can be reduced to $n/(b\cdot p)$, where p is the number of processors. For file update, IM-DSSE$_\mathsf{I}$ requires transmission of $(2b+1)\cdot m$ bits along with decryption and encryption operations at the client side, compared with m non-interactive transmission and encryption-only in the main scheme. Thus, the keyword search speed in IM-DSSE$_\mathsf{I}$ is increased by a factor of b (*e.g.*, $b = 128$) with the cost of transmitting $(2b+1)\cdot m$ bits in the file update.

IM-DSSE$_\mathsf{I}$ reduces the server storage to $\left(\frac{n\cdot|c|+m\cdot n\cdot(b+1)}{b}\right)$ bits. The client storage remains the same as in IM-DSSE$_\mathsf{main}$.

$\underline{(\mathcal{I}_w, \mathcal{C}_w) \leftarrow \mathsf{Search}(\mathcal{K}, w)}$: Given keyword w and key $\mathcal{K}$, return sets of file identifiers $\mathcal{I}_w$ and encrypted files $\mathcal{C}_w \subseteq \mathcal{C}$ matching with w

```
1: $s_{w_i} \leftarrow G_{k_2}(w)$, $i \leftarrow T_w(s_{w_i})$, $r_i \leftarrow G_{k_3}(i)$, $l \leftarrow 0$
2: Fetch the $i$-th row data $\mathbf{I}[i,*]$ from server
3: for $j = 1, \ldots, n$ do
4:     $c_j \leftarrow T_f[j].\mathsf{ct}$
5:     $\mathbf{I}'[i,j] \leftarrow \mathbf{I}[i,j] \oplus H(r_i||j||c_j)$
6: for each $j \in \{1, \ldots, n\}$ satisfying $\mathbf{I}'[i,j] = 1$ do
7:     $l \leftarrow l + 1$ and $y_l \leftarrow j$
8: $\mathcal{I}_w \leftarrow \{y_1, \ldots, y_l\}$
9: Send $\mathcal{I}_w$ to server and receive $\mathcal{C}_w = \{(c_{y_1}, y_1), \ldots, (c_{y_l}, y_l)\}$
10: $f_i \leftarrow \mathsf{Dec}_{\mathcal{K}}(c_{y_i})$ for $1 \leq i \leq l$
11: return $(\mathcal{I}_w, \mathbf{f}_w)$, where $\mathbf{f}_w \leftarrow \{f_1, \ldots, f_l\}$
```

Figure 3.6: $\mathsf{IM\text{-}DSSE_{II}}$ detailed algorithms.

3.4.4.2 $\mathsf{IM\text{-}DSSE_{II}}$: Achieving Cloud SaaS with Backward Privacy

All DSSE schemes introduced so far require the server to perform some computation (*i.e.*, encryption/decryption) during keyword search, which might not be fully compatible with typical storage-only clouds (*e.g.*, Dropbox, Google Drive, Amazon S3). Hence, we propose an extended scheme derived from $\mathsf{IM\text{-}DSSE_{main}}$ called $\mathsf{IM\text{-}DSSE_{II}}$, where all computations are performed at the client side while the server does nothing rather than serving as a storage unit. This simple trick makes $\mathsf{IM\text{-}DSSE_{II}}$ not only compatible with with storage-only clouds, but also more importantly, achieve the backward-privacy property. This is because the server now cannot decrypt any part of encrypted index to keep track of historical update operations. Moreover, $\mathsf{IM\text{-}DSSE_{II}}$ also reduces the storage at both client and server sides by eliminating the state matrix and keyword counters that are needed in $\mathsf{IM\text{-}DSSE_{main}}$ and $\mathsf{IM\text{-}DSSE_{I}}$ to perform correct decryption and achieve forward-privacy during the search and update, respectively.

We present the keyword search procedure of $\mathsf{IM\text{-}DSSE_{II}}$ in Figure 3.6, which combines $\mathsf{SearchToken}$ and Search algorithms in DSSE. To search for keyword w, the client sends to the server the w's row index (i) and receives the corresponding row $\mathbf{I}[i,*]$ (step step 2). The client decrypts $\mathbf{I}[i,*]$, extracts column indexes j such that $\mathbf{I}[i,j]=1$. Since the client computes everything, it is not required to derive new row keys for forward-privacy and therefore, state matrix $\mathbf{I}[*,*].\mathsf{st}$ as well as file hash table T_f at the server and keyword counters $T_w.\mathsf{ct}$ at the client are not needed in IM-DSSE$_{\mathsf{II}}$ (see steps step 3–step 5 for example). The client then fetches and decrypts encrypted files indexed at j to obtain the search result (step step 9).

IM-DSSE$_{\mathsf{II}}$.Gen is identical to IM-DSSE$_{\mathsf{main}}$.Gen Algorithm. IM-DSSE$_{\mathsf{II}}$.Enc, IM-DSSE$_{\mathsf{II}}$.Add, IM-DSSE$_{\mathsf{II}}$.AddToken, IM-DSSE$_{\mathsf{II}}$.Delete, IM-DSSE$_{\mathsf{II}}$.DeleteToken can be easily derived from their version in the main scheme (IM-DSSE$_{\mathsf{main}}$) by *(1)* substituting row key generation $r_i \leftarrow G_{k_3}(i, \overline{c}_i)$ with $r_i \leftarrow G_{k_3}(i)$, *(2)* omitting all keyword counters $\overline{c}_i$, block states $\mathbf{I}[*,*].\mathsf{st}$, *(3)* and removing T_f from the server storage. Due to repetition, we will not present them in detail.

The computation cost of IM-DSSE$_{\mathsf{II}}$ is identical to IM-DSSE$_{\mathsf{main}}$ (*i.e.*, n and m invocations of H for search and update resp.). IM-DSSE$_{\mathsf{II}}$ requires two-round communication with n bits being transmitted during keyword search.

IM-DSSE$_{\mathsf{II}}$ reduces the client and server storage costs to $n(\kappa+|c|)+m\cdot\kappa+3\kappa$ and $m\cdot n$ bits, respectively.

3.4.4.3 IM-DSSE$_{\mathsf{I+II}}$: Efficient Search with Backward Privacy

Our IM-DSSE framework also supports IM-DSSE$_{\mathsf{I+II}}$, an extended DSSE scheme which is the combination of IM-DSSE$_{\mathsf{I}}$ and IM-DSSE$_{\mathsf{II}}$ schemes. In IM-DSSE$_{\mathsf{I+II}}$, the incidence matrix $\mathbf{I}$ is encrypted with b-bit block cipher encryption, and the decryption is performed by the client during search. Since IM-DSSE$_{\mathsf{I+II}}$ inherits all properties of IM-DSSE$_{\mathsf{I}}$ and IM-DSSE$_{\mathsf{II}}$ schemes, IM-DSSE$_{\mathsf{I+II}}$ is highly desirable for cloud SaaS infrastructure that requires a very low search latency and backward-privacy with the costs of more delayed update and an extra communication round during search.

3.4.5 Security Analysis

In this section, we analyze the security and update privacy of all the DSSE schemes provided in our IM-DSSE framework. Most known efficient SSE schemes (*e.g.*, [35, 138, 165]) reveal the *search* and *file-access patterns* defined as follows.

- Given search query w at time t, *the search pattern* $\mathcal{P}(\delta, \text{Query}, t)$ is a binary vector of length t with a 1 at location i if the search time $i \leq t$ was for w, and 0 otherwise. The *search pattern* indicates whether the same keyword has been searched in the past or not.

- Given search query w at time t, *the file-access pattern* $\Delta(\delta, \mathbf{f}, w, t)$ is identifiers $\mathcal{I}_w$ of files $\mathbf{f}$ containing w.

We consider leakage functions in the line of [108] that captures dynamic file addition/deletion in its security model, but we leak much less information compared to [108].

Definition 4 (Leakage Function). We define leakage functions $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$ as follows:

1. $(m, n, id, \langle |f_{id_1}|, \ldots, |f_{id_n}| \rangle) \leftarrow \mathcal{L}_1(\delta, \mathbf{f})$: Given the index δ and the set of files $\mathbf{f}$ (including their identifiers), $\mathcal{L}_1$ outputs the maximum number of keywords m, the maximum number of files n, the identifiers $id = \{id_1, \ldots, id_n\}$ of $\mathbf{f}$ and the size of file $|f_{id_j}|$ for $1 \leq j \leq n$ (which also implies the size of its corresponding ciphertext $|c_{id_j}|$).

2. $(\mathcal{P}(\delta, \text{Query}, t), \Delta(\delta, \mathbf{f}, w, t)) \leftarrow \mathcal{L}_2(\delta, \mathbf{f}, w, t)$: Given the index δ, the set of files $\mathbf{f}$ and a keyword w for the search operation at time t, it outputs the search pattern $\mathcal{P}$ and file-access pattern Δ.

3. $|f_{id}| \leftarrow \mathcal{L}_3(\delta, \mathbf{f}, id, t, \mathsf{op})$: Given the index δ, the set of files $\mathbf{f}$, a file identifier id, and the update type $\mathsf{op} \in \{\langle \mathsf{Add}, |f_{id}| \rangle, \mathsf{Delete}\}$ at time t, it outputs the size of updated file f_{id} (which also implies the size of its corresponding ciphertext $|c_{id}|$).

Remark 1. In Definition 1, we adopt the dynamic CKA2-security *notion in [108] that captures the file addition and deletion by simulating corresponding tokens* τ_f *and* τ'_f*, resp.*

The security of $\mathsf{IM\text{-}DSSE}$ can be stated as follows.

Theorem 1. If $\mathcal{E}.\mathsf{Enc}$ *is IND-CPA secure,* G *is PRF and* H *is a RO then* $\mathsf{IM\text{-}DSSE}$ *is* $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$*-secure in ROM by Definition 1 (CKA-2 security with update capacity).*

Proof. We prove the IND-CKA2 for $\mathsf{IM\text{-}DSSE}_{\mathsf{main}}$ proposed in §3.4.3. The proof for extended schemes in §3.4.4 can be easily derived from this proof (see Remark 2 below for argument) and therefore, we will not repeat it.

To begin with, we construct a simulator $\mathcal{S}$ that interacts with an adversary $\mathcal{A}$ in an execution of an $\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa)$ experiment as described in Definition 1. In this experiment, $\mathcal{S}$ maintains lists $\mathcal{LR}$, $\mathcal{LK}$ and $\mathcal{LH}$ to keep track of query results, states and history information, respectively. Initially, all lists are set to empty. $\mathcal{LR}$ is a list of key-value pairs and is used to keep track of $RO(\cdot)$ queries. We denote $\mathsf{value} \leftarrow \mathcal{LR}(\mathsf{key})$ and $\perp \leftarrow \mathcal{LR}(\mathsf{key})$ if key does not exist in $\mathcal{LR}$. $\mathcal{LK}$ is to keep track of random values generated during the simulation and it is similar to $\mathcal{LR}$. $\mathcal{LH}$ is to keep track of search and update queries, $\mathcal{S}$'s replies to those queries and their leakage output from $(\mathcal{L}_1, \mathcal{L}_2)$. $\mathcal{S}$ executes the simulation as follows.

1. *Handle* $RO(\cdot)$ *Queries*: $b \leftarrow RO(x)$ takes an input x and returns a bit b as output. Given x, if $\perp = \mathcal{LR}(x)$ set $b \xleftarrow{\$} \{0,1\}$, insert (x, b) into $\mathcal{LR}$ and return b as the output. Else, return $b \leftarrow \mathcal{LR}(x)$ as the output.

2. *Simulate* $(\gamma, \mathcal{C})$: Given $(m, n, \langle id_1, \ldots, id_{n'} \rangle, \langle |c_1|, \ldots, |c_{n'}| \rangle) \leftarrow \mathcal{L}_1(\delta, \mathbf{f})$, $\mathcal{S}$ simulates $(\gamma, \mathcal{C})$ as follows.

(a) $(s_{id_j}, k) \xleftarrow{\$} \{0,1\}^{\kappa}$, $y_j \leftarrow T_f(s_{id_j})$, insert (id_j, s_{id_j}, y_j) into $\mathcal{LH}$ and $c_{y_j} \leftarrow \mathsf{Enc}k\{0\}^{|c_{id_j}|}$ for $1 \leq j \leq n'$.

(b) For $j = 1, \ldots, n$ and $i = 1, \ldots, m$

i. $T_w[i].\mathsf{ct} \leftarrow 1$ and $T_f[j].\mathsf{ct} \leftarrow 1$.

ii. $z_{i,j} \xleftarrow{\$} \{0,1\}^{\kappa}$, $\mathbf{I}[i,j] \leftarrow RO(z_{i,j})$ and $\mathbf{I}[i,j].\mathsf{st} \leftarrow 0$.

(c) Output $(\gamma, \mathcal{C})$, where $\gamma \leftarrow (\mathbf{I}, T_f)$ and $\mathcal{C} \leftarrow \{\langle c_i, y_i \rangle\}_{i=1}^{n'}$

$\mathcal{C}$ has the correct size and distribution, since $\mathcal{L}_1$ leaks $\langle |c_{id_1}|, \ldots, |c_{id_{n'}}| \rangle$ and $\mathcal{E}.\mathsf{Enc}(\cdot)$ is a IND-CPA secure scheme, respectively. $\mathbf{I}$ and T_f have the correct size since $\mathcal{L}_1$ leaks (m, n). Each $\mathbf{I}[i,j]$ for $1 \leq j \leq n$ and $1 \leq i \leq m$ has random uniform distribution, since $RO(\cdot)$ is invoked with random value $z_{i,j}$. T_f has the correct distribution, since each s_{id_j} has random uniform distribution, for $1 \leq j \leq n'$. Hence, $\mathcal{A}$ does not abort due to $\mathcal{A}$'s simulation of $(\gamma, \mathcal{C})$. The probability that $\mathcal{A}$ queries $RO(\cdot)$ on any $z_{i,j}$ before $\mathcal{S}$ provides $\mathbf{I}$ to $\mathcal{A}$ is negligible (*i.e.*, $\frac{1}{2^{\kappa}}$). Hence, $\mathcal{S}$ also does not abort.

3. *Simulate* τ_w: Simulator $\mathcal{S}$ receives a search query for an arbitrary keyword w on time t. $\mathcal{S}$ is given $(\mathcal{P}(\delta, \text{Query}, t), \Delta(\delta, \mathbf{f}, w, t)) \leftarrow \mathcal{L}_2(\delta, \mathbf{f}, w, t)$. $\mathcal{S}$ adds these to $\mathcal{LH}$. $\mathcal{S}$ then simulates τ_w and updates lists $(\mathcal{LR}, \mathcal{LK})$ as follows.

(a) If w is in $\mathcal{LH}$, then fetch s_w. Else, $s_w \xleftarrow{\$} \{0,1\}^{\kappa}$, $i \leftarrow T_w(s_{w_i})$, $\overline{c}_i \leftarrow T_w[i].\mathsf{ct}$, insert $(w, \mathcal{L}_1(\delta, \mathbf{f}), s_w)$ into $\mathcal{LH}$.

(b) If $\perp = \mathcal{LK}(i||\overline{c}_i)$, then $r_i \xleftarrow{\$} \{0,1\}^{\kappa}$ and insert $(r_i, i, \overline{c}_i)$ into $\mathcal{LK}$. Else, $r_i \leftarrow \mathcal{LK}(i||\overline{c}_i)$.

(c) If $\overline{c}_i > 1$, then $\overline{r}_i \leftarrow \mathcal{LK}(i||\overline{c}_i - 1)$, $\tau_w \leftarrow (i, r_i, \overline{r}_i)$. Else, $\tau_w \leftarrow (i, r_i)$.

(d) $T_w[i].\mathsf{ct} \leftarrow \overline{c}_i + 1$.

(e) Given $\mathcal{L}_2(\delta, \mathbf{f}, w, t)$, $\mathcal{S}$ knows identifiers $\mathcal{I}_w = \{y_1, \ldots, y_l\}$. Set $\mathbf{I}'[i, y] \leftarrow 1$ for each $y \in \mathcal{I}_w$ and the rest of the elements as $\mathbf{I}'[i, j] \leftarrow 0$ for each $j \in \{\{1, \ldots, n\} \setminus \mathcal{I}_w\}$.

(f) If $((\tau_w = (i, r_i) \vee \mathbf{I}[i, j].\mathsf{st}) = 1)$, then $\mathbf{V}[i, j] \leftarrow \mathbf{I}[i, j]' \oplus \mathbf{I}[i, j]$ and insert tuple $(r_i||j||c_j, \mathbf{V}[i, j])$ into $\mathcal{LR}$, where $c_j \leftarrow T_f[j].\mathsf{ct}$ for $1 \leq j \leq n$.

(g) $\mathbf{I}[i, j].\mathsf{st} \leftarrow 0$ for $1 \leq j \leq n$.

(h) $\mathbf{I}[i, j] \leftarrow \mathbf{I}'[i, j] \oplus RO(r_i||j||c_j)$, where $c_j \leftarrow T_f[j].\mathsf{ct}$ for $1 \leq j \leq n$.

(i) Output τ_w and insert (w, τ_w) into $\mathcal{LH}$.

Given any $\Delta(\delta, \mathbf{f}, w, t)$, $\mathcal{S}$ simulates the output of $RO(\cdot)$ such that τ_w always produces the correct search result for $\mathcal{I}_w \leftarrow \mathsf{Search}\tau_w, \gamma$. $\mathcal{S}$ needs to simulate the output of $RO(\cdot)$ for two conditions: (i) The first search of w (*i.e.*, $\tau_w \stackrel{?}{=} (i, r_i)$), since $\mathcal{S}$ did not know δ during the simulation of $(\gamma, \mathcal{C})$. (ii) If any file f_{id} containing w has been updated after the last search on w (*i.e.*, $\mathbf{I}[i, j].\mathsf{st} \stackrel{?}{=} 1$), since $\mathcal{S}$ does not know the update content. $\mathcal{S}$ sets the output of $RO(\cdot)$ for those cases by inserting tuple $(r_i||j||c_j, \mathbf{V}[i, j])$ into $\mathcal{LR}$. In other cases, $\mathcal{S}$ just invokes $RO(\cdot)$ with $(r_i||j||c_j)$, which consistently returns the previously inserted bit from $\mathcal{LR}$.

During the first search on w, each $RO(\cdot)$ outputs $\mathbf{V}[i, j] = RO(r_i||j|c_j)$ that has the correct distribution, since $\mathbf{I}[i, *]$ of γ has random uniform distribution (see *II-Correctness and Indistinguishability* argument). Let $\mathcal{J} = \{j_1, \ldots, j_l\}$ be the set of indexes of files containing w, which are updated after the last search on w. If w is searched again after being updated, then each $RO(\cdot)$'s output $\mathbf{V}[i, j] = RO(r_i||j|c_j)$ has the correct distribution, since $\tau_f \leftarrow (\mathbf{I}', j)$ for indexes $j \in \mathcal{J}$ has random uniform distribution. Given that $\mathcal{S}$'s τ_w always produces correct $\mathcal{I}_w$ for given $\Delta(\delta, \mathbf{f}, w, t)$, and relevant values and $RO(\cdot)$ outputs have the correct distribution, $\mathcal{A}$ does not abort during the simulation due to $\mathcal{S}$'s search token. The probability that $\mathcal{A}$ queries $RO(\cdot)$ on any $(r_i||j|c_j)$ before querying $\mathcal{S}$ on τ_w is negligible ($i.e.$, $\frac{1}{2^\kappa}$) and, therefore, $\mathcal{S}$ does not abort due to $\mathcal{A}$'s search query.

4. *Simulate* (τ_f,τ'_f): $\mathcal{S}$ receives an update request $\mathsf{op} \in \{\langle \mathsf{Add}, |c| \rangle, \mathsf{Delete}\}$ for an arbitrary file having id at time t. Given $|c_{id}| \leftarrow \mathcal{L}_3(\delta, \mathbf{f}, id, t, \mathsf{op})$, $\mathcal{S}$ simulates update tokens (τ_f, τ'_f) as follows.
    (a) If id is in $\mathcal{LH}$, then fetch (id, s_{id}, j). Else set $s_{id} \overset{\$}{\leftarrow} \{0,1\}^\kappa$, $j \leftarrow T_f(s_{id})$ and insert (id, s_{id}, j) into $\mathcal{LH}$.

    (b) $T_f[j].\mathsf{ct} \leftarrow T_f[j].\mathsf{ct} + 1$, $c_j \leftarrow T_f[j].\mathsf{ct}$.

    (c) If $\perp = \mathcal{LK}(i||\overline{c}_i)$, then $r_i \overset{\$}{\leftarrow} \{0,1\}^\kappa$ and insert $(r_i, i, \overline{c}_i)$ into $\mathcal{LK}$, where $\overline{c}_i \leftarrow T_w[i].\mathsf{ct}$ for $1 \leq i \leq m$.

    (d) $\mathbf{I}'[i,j] \leftarrow RO(z_i)$, where $z_i \overset{\$}{\leftarrow} \{0,1\}^{2\kappa}$ for $1 \leq i \leq m$.

    (e) Set $\mathbf{I}[i,j] \leftarrow \mathbf{I}'[i,j]$ and $\mathbf{I}[i,j].\mathsf{st} \leftarrow 1$ for $1 \leq i \leq m$.

    (f) If $\mathsf{op} = \langle \mathsf{Add}, |c| \rangle$, then simulate $c_j \leftarrow \mathsf{Enc}k\{0\}^{|c|}$ add c_j into $\mathcal{C}$, set $\tau_f \leftarrow (\mathbf{I}', j)$ and output τ_f. Else, set $\tau'_f \leftarrow (\mathbf{I}', j)$, remove c_j from $\mathcal{C}$ and output τ'_f.

Given access pattern (τ_f, τ'_f) for a file f_{id}, $\mathcal{A}$ checks the correctness of update by searching all keywords $\mathbf{w} = \{w_{i_1}, \ldots, w_{i_l}\}$ in f_{id}. Since $\mathcal{S}$ is given access pattern $\Delta(\delta, \mathbf{f}, w, t)$ for a search query (which captures the last update before the search), the search operation always produces a correct result after an update as analyzed above. Hence, $\mathcal{S}$'s update tokens are correct and consistent.

It remains to show that (τ_f, τ'_f) have the correct probability distribution. In the real algorithm, the counter c_j is increased for each update. If f_{id} is updated after the keyword w at row i is searched, a new r_i is generated for w as simulated (r_i remains the same for consecutive updates but c_j increases). Hence, the real algorithm invokes $H(.)$ with a different $(r_i||j||c_j)$ for $1 \leq i \leq m$. $\mathcal{S}$ simulates this step by invoking $RO(\cdot)$ with z_i and $\mathbf{I}'[i,j] \leftarrow RO(z_i)$, for $1 \leq i \leq m$. (τ_f, τ'_f) have random uniform distribution since $\mathbf{I}'$ has random uniform distribution and update operations are correct and consistent as shown above. c_j also has the correct distribution since $\mathcal{E}.\mathsf{Enc}(\cdot)$ is an IND-CPA encryption. Hence, $\mathcal{A}$ does not abort during the simulation due to $\mathcal{S}$'s update tokens. The probability that $\mathcal{A}$ queries $RO(\cdot)$ on any z_i prior querying $\mathcal{S}$ on (τ_f, τ'_f) is negligible (*i.e.*, $\frac{1}{2^{2\cdot\kappa}}$) and, therefore, $\mathcal{S}$ does not abort due to $\mathcal{A}$'s update query.

5. *Final Indistinguishability Argument*: (s_{w_i}, s_{id_j}, r_i) for $1 \leq i \leq m$ and $1 \leq j \leq n$ are indistinguishable from real tokens and keys since they are generated by PRFs that are indistinguishable from random functions. $\mathcal{E}.\mathsf{Enc}(\cdot)$ is a IND-CPA scheme, the answers returned by $\mathcal{S}$ to $\mathcal{A}$ for $RO(\cdot)$ queries are consistent and appropriately distributed, and all query replies of $\mathcal{S}$ to $\mathcal{A}$ during the simulation are correct and indistinguishable as discussed above. Hence, for all PPT adversaries, the outputs of $\mathsf{Real}_{\mathcal{A}}(\kappa)$ and $\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa)$ experiment are:

$$|\Pr[\mathsf{Real}_{\mathcal{A}}(\kappa) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa) = 1]| \leq \mathsf{neg}(\kappa),$$

where $\mathsf{neg}(\cdot)$ is a negligible function.

□

We argue the security of extended schemes as follows.

Remark 2. IM-DSSE$_I$, IM-DSSE$_{II}$ and IM-DSSE$_{I+II}$ are secure by Definition 1, and their proofs can be easily derived from the security analysis of IM-DSSE$_{main}$ with the intuition as follows.

IM-DSSE$_I$ only differs from IM-DSSE$_{main}$ in terms of b-bit encryption, compared with 1-bit encryption. This modification does not impact the IND-CKA2 security of IM-DSSE$_I$ over IM-DSSE$_{main}$.

Given that we use ROs and an IND-CPA encryption scheme (e.g., AES with CTR mode), the security of IM-DSSE$_{I}$ is not affected in our model, and in particular, there is no additional leakage. The price that is paid for this performance improvement is that the scheme becomes interactive. Since the block data exchanged between client and server are encrypted with an IND-CPA encryption scheme, there is no additional leakage due to this operation.

IM-DSSE$_{II}$ only differs from IM-DSSE$_{main}$ in terms of where the decryption during keyword search takes place. Performing decryption at the client (instead of at the server) does not impact the IND-CKA2 security of IM-DSSE$_{II}$ over IM-DSSE$_{main}$. Given that we, respectively, use ROs and PRF for H and G as in IM-DSSE$_{main}$, the security of IM-DSSE$_{II}$ remains the same.

IM-DSSE$_{I+II}$ is merely the combination of IM-DSSE$_{I}$ and IM-DSSE$_{II}$, where block cipher encryption and client decryption during search are both implemented. As analyzed, each of these strategies does not impact the security and therefore, IM-DSSE$_{I+II}$ still preserve the IND-CKA2 security.

The leakage definition and formal security model imply various levels of privacy for different DSSE schemes. We summarize some important privacy notions based on the various leakage characteristics discussed in [165] as follows.

- *Size pattern*: The number of actual keyword-file pairs.
- *Forward privacy*: A search on a keyword w does not leak the IDs of files being updated in the future and having w.
- *Backward privacy*: A search on a keyword w does not leak all historical update operations (*e.g.*, addition /deletion) on the identifiers of files having this keyword.

Since keyword-file relationships are represented by an encrypted incidence matrix, IM-DSSE framework hides the *size pattern* (*i.e.*, number of '1' in **I**), so that it is size-oblivious.

Corollary 1. IM-DSSE framework offers forward-privacy.

Proof. In IM-DSSE framework, the update involves reconstructing a new column/block of encrypted index **I**. The column/block is always encrypted with row keys that have never been revealed to the server. This is achieved in IM-DSSE$_{\text{main}}$ and IM-DSSE$_{\text{I}}$ schemes by increasing the row counter after each keyword search operation so that fresh row keys will be used for subsequent update operations. In IM-DSSE$_{\text{II}}$ scheme, since all cryptographic operations are performed at the client side where no keys are revealed to the server, it is unable for the server to infer any information in the update, given that the encryption scheme is IND-CPA secure. These properties enable our IM-DSSE framework to achieve forward privacy. □

Corollary 2. IM-DSSE$_{II}$ and IM-DSSE$_{I+II}$ achieve backward-privacy.

Proof. In most DSSE schemes, the client sends a key that allows the server to decrypt a small part of the encrypted index during keyword search. The server can use this key to backtrack historical update operations on this part and therefore, compromise the backward-privacy. In IM-DSSE$_{\text{II}}$ and IM-DSSE$_{\text{I+II}}$ schemes, instead of sending the key to the server, the client requests this part and decrypts it locally. This prevents the server from learning information about historical update operations on the encrypted index and therefore, allows both schemes to achieve backward-privacy. □

3.4.6 Experimental Evaluation

In this section, we study the performance of our IM-DSSE framework in real-life networking and system settings. We present a detailed cost breakdown analysis to fully assess the criteria that constitute the performance overhead of our constructions. Given that such analysis is generally missing in the literature, this is the main focus of our performance evaluation. Finally, we give a brief asymptotic comparison of our framework with several DSSE schemes in the literature.

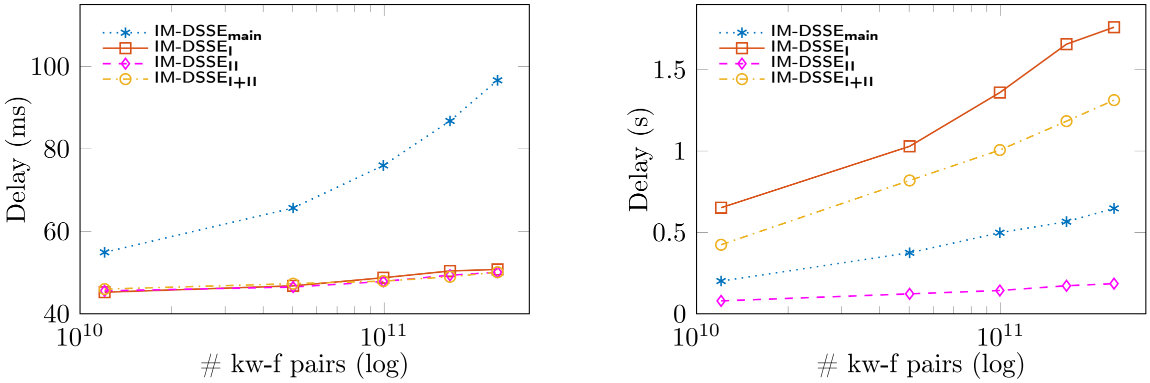
3.4.6.1 Implementation

We implemented our framework using C/C++. For cryptographic primitives, we used libtomcrypt library [53]. We modified low level routines to call AES hardware acceleration instructions (via Intel AES-NI library [80]) if they are supported by the underlying hardware platform. We used AES-128 Cipher-based Message Authentication Code (CMAC) for hash function. Our random oracles were all implemented via 128-bit AES CMAC. For hash tables, we employed Google's C++ sparse hash map library [4] with the hash function being implemented by the CMAC-based random oracles truncated to 80 bits. We implemented the IND-CPA encryption $\mathcal{E}$ using AES with CTR mode. For network communication, we used ZeroMQ library [3]. IM-DSSE framework contains the full implementation of all schemes including IM-DSSE$_{\text{main}}$, IM-DSSE$_{\text{I}}$, IM-DSSE$_{\text{II}}$ and IM-DSSE$_{\text{I+II}}$, which can be freely accessed via our following Github repository [85]: https://github.com/thanghoang/IM-DSSE/.

Our implementation supports the encrypted index stored on either memory or local disk. Therefore, our schemes can be directly deployed in either storage-as-a-service (*e.g.*, Amazon S3) or infrastructure-as-a-service clouds (*e.g.*, Amazon EC2). For this experimental evaluation, we selected block cipher size $b = 128$ for IM-DSSE$_{\text{I}}$ and IM-DSSE$_{\text{I+II}}$ schemes.

3.4.6.2 Configurations and Methodology

We used subsets of the Enron email dataset [2], ranging from 50,000 to 250,000 files with 240,000–940,000 unique keywords to evaluate the performance of our schemes with different encrypted



(a) Keyword search

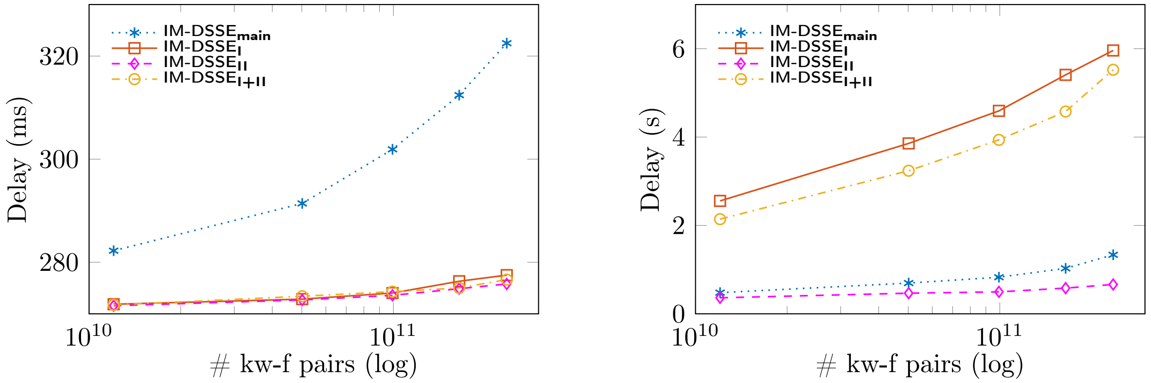
(b) File update

Figure 3.7: Latency of IM-DSSE with fast network.

index sizes. These selected sizes surpass the experiments in [109] by three orders of magnitude and are comparable to the experiments in [165].

We conducted the experiment with two settings: (i) We used HP Z230 Desktop as the client and built the server using Amazon EC2 with `m4.4xlarge` instance type. The desktop was equipped with Intel Xeon CPU E3-1231v3 @ 3.40GHz, 16 GB RAM, 256 GB SSD and CentOS 7.2 was installed. The server was installed with Ubuntu 14.04 and equipped with 16vCPUs @2.4 GHz Intel Xeon E5-2676v3, 64 GB RAM and 500 GB SSD hard drive. (ii) We selected LG G4 mobile phone to be the client machine running Android OS, v5.1.1 (Lollipop) and was equipped with Qualcomm Snapdragon 808 64-bit Hexa-core CPU @1.8 GHz, 3GB RAM and 32 GB internal storage. Notice that AES-NI library cannot be used to accelerate cryptographic operations on this mobile device due to its incompatible CPU, which affects the performance of our schemes in the mobile environment as will be shown in the following section.

We disabled the slow-start TCP algorithm and maximized initial congestion window parameters in Linux (*i.e.*, 65535 bytes) (see [57] for more insights) to reduce the network impact during the initial phase in case the scheme requires low amount of data to be transmitted.



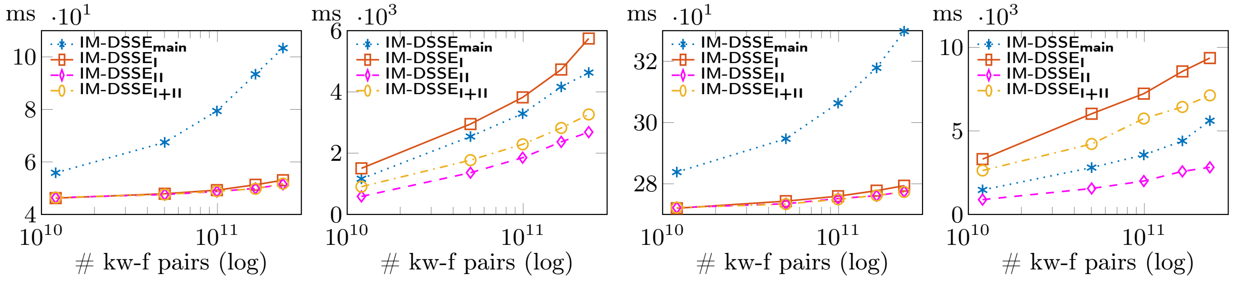
(a) Keyword search

(b) File update

Figure 3.8: IM-DSSE delay with moderate network.

3.4.6.3 Overall Results.

Figure 3.7 presents the overall performance in terms of end-to-end cryptographic delay of all the schemes in IM-DSSE framework. In this experiment, we located client and server in the same geographical region, resulting in a network latency of 11.2 ms and a throughput of 264 Mbps. We refer to this configuration as a *fast network* setting. Notice that we only measured the delay due to accessing the encrypted index **I**, and omitted the time to access encrypted files (*i.e.*, set $\mathcal{C}$) as it is identical for all searchable encryption and non-searchable encryption schemes. For instance, in keyword search, we measured the delay of IM-DSSE$_{main}$ scheme and IM-DSSE$_{I}$ scheme by the time the client sends the request and the server finishes decrypting an entire row of the encrypted index and gets cells whose value is 1. The IM-DSSE$_{main}$ scheme and its extended versions took less than 100 ms to perform a keyword search, while it took less than 2 seconds to update a file. The cost per keyword search depends linearly on the maximum number of files in the database (*i.e.*, $\mathcal{O}(n)$) and yet it is highly practical even for very large numbers of keyword-file pairs (*i.e.*, more than 10^{11} pairs). Indeed, we confirm that the search operation in IM-DSSE is very fast and most of the overhead is due to network communication delay as it will be later analyzed in this section. Note that the costs for



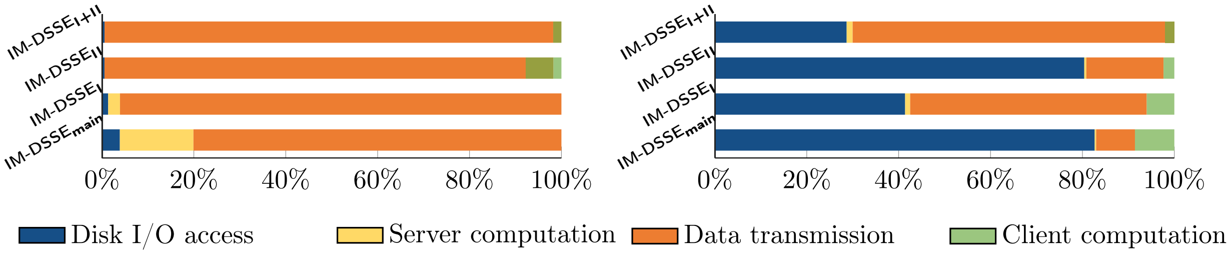
(a) Keyword search (b) File update (c) Keyword search (d) File update

Figure 3.9: IM-DSSE delay on SSD, and (a,b) fast and (c,d) moderate networks.

adding and deleting files (updates) over the encrypted index are highly similar since their procedure is almost identical.

The keyword search operation delay of IM-DSSE$_{\text{main}}$ is higher than that of extended schemes and the difference increases as the size of the encrypted index increases due to two reasons: First, the encrypted index **I** in IM-DSSE$_{\text{main}}$ scheme is bit-by-bit encrypted compared with 128-bit block encryption in IM-DSSE$_{\text{I}}$. Hence, the server needs to derive more AES keys than in IM-DSSE$_{\text{I}}$ to decrypt a whole row. Thus, the gap between IM-DSSE$_{\text{main}}$ and IM-DSSE$_{\text{I}}$ represents the server computation cost required for this key derivation and encryption. Second, the processes in IM-DSSE$_{\text{main}}$ scheme are performed subsequently, in which the server needs to receive some information sent from the client first before being able to derive keys to decrypt a row. Such processes in IM-DSSE$_{\text{I}}$ and IM-DSSE$_{\text{II}}$ can be parallelized, where the client generates the AES-CTR keys while receiving a row of data transmitted from the server. We can see that the delay is similar between IM-DSSE$_{\text{I}}$ and IM-DSSE$_{\text{II}}$ and IM-DSSE$_{\text{I+II}}$. This indicates that using 128-bit encryption significantly reduces the server computation cost to a point, where it becomes negligible as being later shown.

Considering the file update operation, our IM-DSSE$_{\text{main}}$ and IM-DSSE$_{\text{II}}$ schemes leverage 1-bit encryption and, therefore, it does not require to transfer a 128-bit block to the client first prior to updating the column as in IM-DSSE$_{\text{I}}$ and IM-DSSE$_{\text{I+II}}$ schemes. Hence, they are faster and less



(a) Keyword search

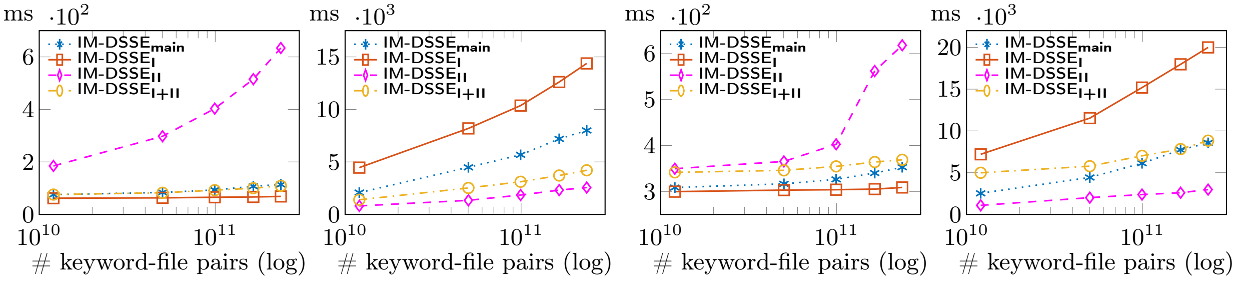
(b) File update

Figure 3.10: Detailed costs of $\mathsf{IM\text{-}DSSE}$ with moderate network and SSD storage.

affected by the network latency than $\mathsf{IM\text{-}DSSE}_{\mathsf{I}}$ and $\mathsf{IM\text{-}DSSE}_{\mathsf{I+II}}$. So, the gap between such schemes reflects the data download delays, which will be significantly higher on slower networks as shown in the next experiment. Update in $\mathsf{IM\text{-}DSSE}_{\mathsf{I+II}}$ is considerably faster than in $\mathsf{IM\text{-}DSSE}_{\mathsf{I}}$ because it allows for parallelization, in which the client can pre-compute AES-CTR keys while receiving data from the server. In $\mathsf{IM\text{-}DSSE}_{\mathsf{I}}$, such keys cannot be pre-computed as they need some information from the server beforehand (*i.e.*, state data $\mathbf{I}[*, j].\mathsf{st}$).

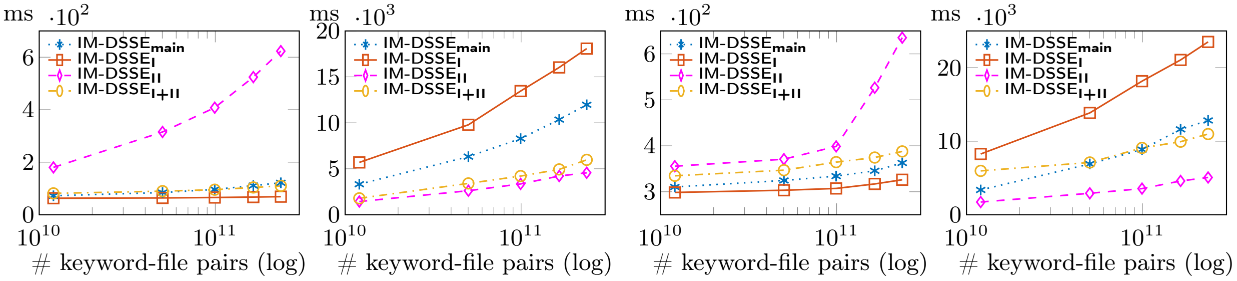
3.4.6.4 The Impact of Network Quality.

The previous experiments were conducted on a high-speed network, which might not be widely available in practice. Hence, we additionally investigated how our schemes perform when the network speed is degraded. We setup the server to be geographically located distant from the client machine, resulting in the network latency and throughput to be 67.5 ms and 46 Mbps, respectively. Figure 3.8 shows the end-to-end cryptographic delay of our schemes in this *moderate network* setting. Due to the high network latency, search operation of each scheme is slower than that of fast network by 230ms. The impact of the network latency is clearly shown in the update operation as reflected in Figure 3.8b. The delays of $\mathsf{IM\text{-}DSSE}_{\mathsf{I}}$, $\mathsf{IM\text{-}DSSE}_{\mathsf{I+II}}$ are significantly higher than those of $\mathsf{IM\text{-}DSSE}_{\mathsf{main}}$



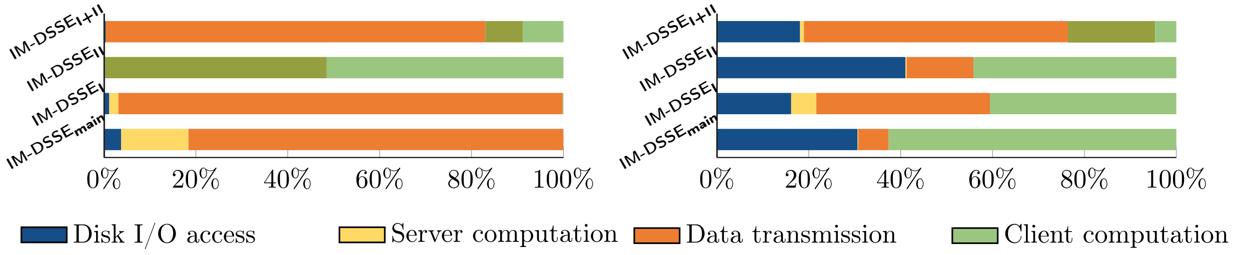
(a) Keyword search (b) File update (c) Keyword search (d) File update

Figure 3.11: IM-DSSE delay on mobile, RAM and (a,b) fast and (c,d) moderate networks.



(a) Keyword search (b) File update (c) Keyword search (d) File update

Figure 3.12: IM-DSSE delay on mobile, SSD and (a,b) fast and (c,d) moderate networks.



(a) Keyword search (b) File update

Figure 3.13: Detailed costs of IM-DSSE on mobile with moderate network and SSD storage.

and IM-DSSE$_{\text{II}}$. As explained previously, this gap actually reflects the download delay incurred by such schemes.

3.4.6.5 The Impact of I/O Access

Another important performance factor for DSSE is the encrypted index storage access delay. Hence, we investigated the impact of the encrypted index storage location on the performance of our schemes. Clearly, the ideal case is to store all server-side data on RAM to minimize the delay introduced by storage media access as shown in previous experiments. However, deploying a cloud server with a very large amount of RAM capacity can be very costly. Thus, in addition to the RAM-stored results shown previously, we stored the encrypted index on the secondary storage unit (*i.e.*, SSD drive), and then measured how overall delays of our scheme were impacted by this setting. Figure 3.9 presents the results with two aforementioned network quality environments (*i.e.*, fast and moderate speeds). In IM-DSSE$_{\mathsf{main}}$ and IM-DSSE$_{\mathsf{I}}$ schemes, the disk I/O access is incurred by loading a part of the encrypted index including value $\mathbf{I}.v$ and state $\mathbf{I}.\mathsf{st}$. It is clear that the disk I/O access time incurred an insignificant latency to the overall delay in terms of keyword search operation as shown in Figure 3.9a and Figure 3.9d, since our schemes achieve perfect locality as defined by Cash *et al.* [37]. However, in the file update operation, the delay in IM-DSSE framework was 1–4 seconds more, compared with RAM-based storage. That is because we stored all cells in each row of the encrypted matrix $\mathbf{I}$ in contiguous memory blocks. Therefore, keyword search invokes accessing *subsequent memory blocks* while update operation results in accessing *scattered blocks* which incurs much higher disk I/O access time. Due to the incidence matrix data structure and this storage strategy, our search operation was not affected as much by disk I/O access time as other non-local DSSE schemes (*e.g.*, [35, 36, 108]), which require accessing random memory blocks for security.

3.4.6.6 Cost Breakdown.

We dissected the overall cost of our schemes previously presented in Figure 3.7, Figure 3.8 and Figure 3.9 to investigate which factors contribute a significant amount to the total delay of each scheme. For analysis, we selected the cost of our schemes when performing on the largest encrypted index size being experimented (*i.e.*, 2.36×10^{11}) with moderate network speed, where the encrypted index is stored on an SSD drive. Figure 3.10 presents the major factors that contribute to the total delay of our schemes during keyword search and file update operations.

Considering the search operation, it is clear that data transmission occupied the largest amount of delay among all schemes. In our $\mathsf{IM\text{-}DSSE}_{\mathsf{main}}$ and $\mathsf{IM\text{-}DSSE}_{\mathsf{I}}$ schemes, most of the computations were performed by the server wherein cryptographic operations were accelerated by AES-NI so that they only took a small portion of the total, especially in $\mathsf{IM\text{-}DSSE}_{\mathsf{I}}$ scheme. Meanwhile, the client only performed simple computations such as search token generation so that its cost was negligible. In $\mathsf{IM\text{-}DSSE}_{\mathsf{II}}$ and $\mathsf{IM\text{-}DSSE}_{\mathsf{I+II}}$ schemes, encrypted data were decrypted at the client side, while the server did nothing but transmission. Therefore, the client computation cost took a small portion of the total delay and the server's cost was negligible. However, as indicated in §3.4.4, the client computation and data transmission in $\mathsf{IM\text{-}DSSE}_{\mathsf{II}}$ and $\mathsf{IM\text{-}DSSE}_{\mathsf{I+II}}$ are fully parallelizable where their partially parallel costs are indicated by their overlapping area in Figure 3.10a. Hence, we can infer that client computation was actually dominated by data transmission and, therefore, the computation cost did not affect the total delay of the schemes. As explained above, we stored the encrypted matrix on disk with row-friendly strategy so that the disk I/O access time due to keyword search was insignificant, which contributed less than 3% to the total delay.

In contrast, it is clear that disk I/O access time occupied a considerable proportion of the overall delay of the update operation, especially in the $\mathsf{IM\text{-}DSSE}_{\mathsf{main}}$ and $\mathsf{IM\text{-}DSSE}_{\mathsf{II}}$ schemes due to non-contiguous memory access. Data transmission was the second major factor contributing to the total delay. As the server did not perform any expensive computations, its cost was negligible in all schemes. The client performed cryptographic operations which were accelerated by AES-NI library so that it only contributed less than 7% to the overall cost. Additionally, the client computation was mostly parallelized with the data transmission and the server's operations in $\textsf{IM-DSSE}_{\textsf{II}}$ and $\textsf{IM-DSSE}_{\textsf{I+II}}$ schemes so that it can be considered not to significantly impact the total delay.

We evaluated our schemes' performance when deployed on a mobile device with limited computational resources. Similar to the desktop experiments, we tested on fast and moderate network by geographically locating the server close and distant from the mobile, respectively. The phone was connected to a local WiFi which, in turn, allowed the establishment of the connection to the server via a wireless network resulting in the latency and throughput of fast network case to be 18.8 ms, 136 Mbps while those of moderate case were 76.3 ms and 44 Mbps, resp. Figure 3.11 and Figure 3.12 present the benchmark results with aforementioned network settings when the data in the server were stored on RAM and SSD, respectively. In the mobile environment, the $\textsf{IM-DSSE}_{\textsf{II}}$ scheme performed considerably slower than others in terms of keyword search. That is because, in this scheme, a number of cryptographic operations (*i.e.*, $\mathcal{O}(n)$) were performed by the mobile device. Moreover, these operations were not accelerated by AES-NI library as in our Desktop machine because the mobile CPU did not have special crypto-accelerated instructions. Considering the keyword search performance of $\textsf{IM-DSSE}_{\textsf{II}}$ in the moderate network setting (*i.e.*, Figure 3.11c and Figure 3.12c), we can see that its delay significantly increased when the size of encrypted index exceeded 10^{11} keyword-file pairs. This is because starting from this size of the encrypted index, the client computation began to dominate the data transmission cost. The update delays of our schemes, especially the $\textsf{IM-DSSE}_{\textsf{main}}$ and $\textsf{IM-DSSE}_{\textsf{II}}$ schemes, were substantial in the mobile environment because the mobile platform had to perform intensive cryptographic operations.

Figure 3.13 shows the decomposition of the total end-to-end delay of our schemes in the out-of-state network setting when the server data were stored on an SSD drive. For the search operation, the detailed costs of $\textsf{IM-DSSE}_{\textsf{main}}$ and $\textsf{IM-DSSE}_{\textsf{I}}$ schemes are the same as in the desktop setting since computations were mostly performed by the server while the client only performed some lightweight computation to generate the token. In IM-DSSE$_{\text{II}}$, the client computation contributed almost 100% to the total delay due to $\mathcal{O}(n)$ number of AES-CTR decryptions, compared with $\mathcal{O}(n)/128$ in IM-DSSE$_{\text{I+II}}$ which was all dominated by the data transmission delay. The limitation of computational capability of the mobile device is reflected clearly in Figure 3.13b, wherein the client computation cost accounted for a considerable amount of the overall delay of most schemes except for the IM-DSSE$_{\text{I+II}}$ scheme.

We reported the delays of our framework without taking optimization into account. Meanwhile, the performance of our framework can be further optimized by applying several caching strategies to minimize the computation, I/O access and communication overhead at both client and server sides. Specifically, for each search operation, one can observe that our framework requires to decrypt and re-encrypt an entire row in the incidence matrix. This increases significantly the computation overhead whereas it is not necessary to protect the row confidentiality once its content is already revealed. Therefore, given a keyword to be searched at the first time, the server can decrypt the row, and cache all positions that indicate the files containing the keyword in a compact index such as encrypted dictionary. When the keyword is repeatedly searched, the server can simply look up this index to obtain the search result. This caching strategy reduces the server computation overhead with the cost of maintaining an extra data structure. Both client and server can also cache the content of keywords being searched/update frequently on local persistent memory to reduce the network communication and I/O access thereby, reducing the overall delay. We note that the efficiency of this caching is application-specific since it depends on the characteristics of the outsourced database and the user query. By open-sourcing the implementation of our framework, we leave its optimization to practitioners when deployed on specific use-cases in practice.

3.5 FS-DSSE: Forward-Private Searchable Encryption with Efficient Search

We propose a new DSSE scheme that offers important features for practical deployment including forward-privacy, sublinear search time with parallelization support, and low client storage. This is achieved by harnessing a secure update strategy on a special encrypted index structure along with a novel caching strategy using a dictionary data structure to partially store the result of previous search queries. We refer our scheme as *Forward-Private and Sublinear DSSE (FS-DSSE)* scheme, with the following desirable properties:

- *High-speed search with full parallelization:* The proposed scheme offers the lowest search delay among its counterparts. From the asymptotic point of view, our search complexity is *(i)* equivalent to the most efficient yet *forward-insecure* DSSE scheme [35], and *(ii)* lower than some state-of-the-art forward-private DSSE schemes. Our proposed scheme is also fully parallellizable, and therefore, can take advantage of multi-threading techniques offered by the cloud. The experimental evaluation showed that, our search delay was comparable to the most efficient yet forward-insecure DSSE scheme, while it was one to three orders of magnitude faster than its forward-private counterparts [26] (see §3.5.4).

- *Low client storage overhead:* The proposed scheme features $\mathcal{O}(1)$ client storage overhead, in which the client only needs to store a few symmetric keys. This property allows the proposed scheme to be deployed on mobile devices where the client has a limited memory capacity.

- *High security:* Our scheme not only achieves forward-privacy as the important security feature, but also can hide the size information of some operations on the encrypted index similar to [189]. Specifically, the proposed scheme does not leak the number of actual keyword-file pairs in update operation, and the encrypted index size as defined in [165]. Note that such information is leaked in most state-of-the-arts DSSE schemes (except [189]), which might be exploited in statistical attacks.

- *Full-fledged implementation and evaluation on real infrastructure:* We fully implemented the proposed scheme and extensively evaluated its performance on a real computing infrastructure. The experimental result demonstrated that the proposed scheme is highly efficient, which showed the potentials to be deployed on real-world breach-resilient infrastructure. We release the implementation of our scheme for public testing and wide adaptation (see §3.5.4).

Our main observation is that the search query in standard DSSE reveals a part of the encrypted index to the server while retrieving the corresponding encrypted files. Therefore, once a keyword is searched again, it is not necessary to repeat the computation on the encrypted index to extract corresponding files that were previously revealed. Instead, one can leverage a more compact and simple data structure (*e.g.*, dictionary) to store file IDs revealed in the first search so that if the same query is repeated, the server will simply get the results stored in this data structure. This strategy will amortize the computation cost incurred in the first search operation and therefore, will make DSSE schemes more efficient. Note that the price to pay for gaining this search efficiency is (in worst case) doubling the server storage overhead.

The second objective is to find a DSSE scheme that efficiently adopts the aforementioned strategy. We observe that, the DSSE scheme in [189] offers a high level of security including forward-privacy with the cost of linear search complexity. This computation cost can be significantly reduced by using our proposed caching strategy mentioned above. Therefore, we take the DSSE scheme in [189] as the base case to construct FS-DSSE with the caching strategy. We start by giving some brief overview about the data structures used in our scheme, some of which are borrowed from

[189]. We refer the reader to [189] for detailed description. Note that our caching strategy can be applied to *any forward-private DSSE scheme (e.g. Sophos [26])* to reduce the search cost.

3.5.1 FS-DSSE Data Structures

FS-DSSE leverages Incidence matrix **I** as the search index and (*ii*) hash table T_w to keep track of the position of keywords in **I** similar to IM-DSSE (see §3.4.2). For simplicity, we assume files are indexed from 1 to n and therefore, it is not required to create a hash table for them. Additionally, since FS-DSSE leverages a caching strategy at the server to reduce the computation cost of repeated search operations, we employ a dictionary data structure D to store the search result of the queries when the keyword is first searched. D can be considered as an array of size m, where $D[i]$ stores the list of file IDs which is revealed when searching the keyword indexing at row i in **I**. D is encrypted with IND-CPA encryption and is updated if there are file operations performed on **I** in between the search queries. We present the update policy in the following section to keep D consistent.

3.5.2 Detailed FS-DSSE Algorithms

In this section, we present the detailed procedures in FS-DSSE as follows:

3.5.2.1 Setup Algorithm

First, given a list of plaintext files **f**, the client creates the encrypted files $\mathcal{C}$ and encrypted index and sends them to the server by calling IM-DSSE$_{\mathsf{main}}$.Gen procedure. This procedure *(i)* generates three keys, $\{k_1, k_2, k_3\}$ that collectively constitutes the key K, used to encrypt the files, the incidence matrix, and the counters in the hash table, respectively, *(ii)* extracts keywords from the input files, each being assigned to a row index via the hash table T_w, and *(iii)* sets the corresponding value for

```
$(\mathbf{I}, \mathcal{C}, \sigma, \mathrm{K}) \leftarrow \mathsf{FS\text{-}DSSE.Setup}(1^\kappa, \mathbf{f})$: Create encrypted index
$k_1 \leftarrow \mathcal{E}.\mathsf{Gen}(1^\kappa)$ and $(k_2, k_3) \xleftarrow{\$} \{0,1\}^\kappa$
Set $\mathrm{K} \leftarrow \{k_1, k_2, k_3\}$
Extract keywords $(w_1, \ldots, w_{m'})$ from $\mathbf{f} = \{f_{id_1}, \ldots, f_{id_{n'}}\}$
Set counter $u_i = 1$, state $v_i = 1$ for each $w_i$ in hash table $T_w$
Set $\mathbf{I}[i, j] \leftarrow 1$ where $i$ is the keyword index in $T_w$ and $1 \leq j \leq n$ if $f_j$ contains keyword $i$
Generate row keys $r_i \leftarrow \mathsf{PRF}(k_2||i||u_i)$ for $1 \leq i \leq m$
Encrypt each row of $\mathbf{I}$ with key $r_i$ for $1 \leq i \leq m$
Encrypt counters in hash table $T_w$ with $k_3$
Encrypt all files $\mathbf{f}$ with $k_1$ as $\mathcal{C} = \{c_j : c_j \leftarrow \mathsf{Enc}k_1 f_j\}$
return $(\mathbf{I}, \mathcal{C}, \sigma, \mathrm{K})$, where $\sigma \leftarrow T_w$ (Send $(\mathbf{I}, \mathcal{C}, \sigma)$ to server)
```

Figure 3.14: FS-DSSE setup algorithm.

each $\mathbf{I}[i, j]$. Note that a counter for each keyword is also stored in T_w, which will be used to derive a key to achieve the forward-privacy during update. Finally, the client generates the encrypted incidence matrix $\mathbf{I}$, encrypts counters in hash table T_w and encrypts all files $\mathbf{f}$, and sends them to the server, while keeping the key K secret.

3.5.2.2 Search Protocol

To search a keyword w, the client first requests the encrypted counter of w stored in T_w. Then, they send a search token containing the row index i and the row key r_i derived from the counter, to the server. If the keyword is being searched for the first time, then the server decrypts the whole row $\mathbf{I}[i, *]$ with r_i, adds all column indexes j, where $\mathbf{I}[i, j] = 1$, to the dictionary $D[i]$, and encrypts $D[i]$[2]. The server returns the corresponding encrypted files matching with such indexes to the client.

If a previously-searched keyword is searched again, the server retrieves indexes of corresponding encrypted files by simply decrypting $D[i]$. It is important to note that $D[i]$ might need to be updated, given that there are some file update operations on $\mathbf{I}$ that happened after the latest search

[2] Server encrypts D with a self-generated key just to preserve the data privacy against outside attackers.

```
$(\mathcal{R}, \sigma') \leftarrow \mathsf{FS\text{-}DSSE.Search}_{\mathcal{K}}(w, \mathbf{I}, \sigma)$
Client:
Let $i$ be index of $w$ in $T_w$
Download the counter $u_i$ of keyword $w_i$ in $T_w$
Decrypt $u_i$ with $k_3$ and generate key $r_i \leftarrow \mathsf{PRF}(k_2||i||u_i)$
Send $(i, v'_i, r_i)$ to server
Server: On receive $(i, v'_i, r_i)$:
if $i$ is first-time searched then
    Let $\mathbf{I}'[i, *]$ be the decryption of $\mathbf{I}[i, *]$ using key $r_i$
    $D[i] \leftarrow \{j : \mathbf{I}'[i, j] = 1\}$
    Encrypt $D[i]$
else
    Decrypt $D[i]$
    Let $J = \{j : \mathbf{I}[i, j].st = 1\}$
    for each $j$ in $J$ do
        Let $\mathbf{I}'[i, j]$ be the decryption of $\mathbf{I}[i, j]$ using key $r_i$
        Add $j$ to $D[i]$ if $\mathbf{I}'[i, j] = 1$, or delete $j$ if otherwise
Send $\mathcal{R} = \{c_j : j \in D[i]\}$ to client
Re-encrypt $D[i]$, set $\mathbf{I}[i, *].st = 0$ and $T_w[i].v \leftarrow 0$
return $(\mathcal{R}, \sigma')$, where $\sigma' \leftarrow T_w$ with $i$-th entry being updated
Client: On receive $\mathcal{R}$:
$f_j \leftarrow \mathsf{Dec}k_1 c_j$ for each $c_j \in \mathcal{R}$
```

Figure 3.15: FS-DSSE search protocol.

on w_i. This is achieved by checking the state bit $\mathbf{I}[*, *].st$. Specifically, if $\mathbf{I}[i, j].st = 1$, then the server decrypts $\mathbf{I}[i, j]$ and adds the entry j to $D[i]$ if $\mathbf{I}[i, j] = 1$ (or deletes j if $\mathbf{I}[i, j] = 0$).

The use of data structure D enables FS-DSSE to have an amortized sublinear search complexity. Specifically, the computation cost of the first query is $\mathcal{O}(n)$ while that of repeated queries is $\mathcal{O}(r)$, where r is the result size of the first query. The amortized cost is $\mathcal{O}(r + d_w)$, where d_w is the number of updates, after n search repetitions.

3.5.2.3 Update Protocol

Given an updated file f_j, the client extracts the updated keywords and creates an unencrypted column $\mathbf{I}[*, j]$, which represents the relationship between f_j with all the keywords in DB. The client then generates m row keys r_i according to corresponding counters stored in T_w at the server. To achieve the forward-privacy, the client must encrypt $\mathbf{I}[*, j]$ with fresh keys, which are unknown to

```
$(\mathbf{I}', \mathcal{C}', \sigma') \leftarrow \textsf{FS-DSSE.Update}_{\mathcal{K}}(f_j, \mathbf{I}, \sigma, \mathcal{C})$
 Client:
  Set $\bar{\mathbf{I}}[*, j] \leftarrow 0$ and $c \leftarrow NULL$
  Download all counters $u_i$ and states $v_i$ $(1 \leq i \leq m)$ in $T_w$
  Download state column $\mathbf{I}[*, j].st$
  Let $V = \{i : v_i = 0\}$
  Extract keywords $(w_1, \ldots, w_t)$ from $f_j$
  Decrypt counters $u_i$ with $k_3$, and set $u_i \leftarrow u_i + 1$ for each $i \in V$
  Generate keys $r_i \leftarrow \textsf{PRF}(k_2 || i || u_i)$
  Set $\bar{\mathbf{I}}[x_i, j] \leftarrow 1$ for $1 \leq i \leq t$, where $x_i$ is index of $w_i$ in $T_w$
  Encrypt $f_j$ as $c_j \leftarrow \textsf{Enc}k_1 f_j$
  Encrypt each row $\bar{\mathbf{I}}[i, j]$ with row keys $r_i$, for $1 \leq i \leq m$
  Let $\overrightarrow{u} \leftarrow (u_1, \ldots, u_n)$, where each $u_i$ is encrypted with $k_3$
  Send $(\bar{\mathbf{I}}[*, j], c_j, j, \overrightarrow{u})$ to server
 Server:   On receive $(\bar{\mathbf{I}}[*, j], c_j, j, \overrightarrow{u})$:
  $T_w[i].u \leftarrow u_i$ and set $T_w[i].v \leftarrow 1$ for $1 \leq i \leq m$
  Set $\mathbf{I}[*, j] \leftarrow \bar{\mathbf{I}}[*, j]$ and $\mathbf{I}[*, j].st \leftarrow 1$
  return $(\mathbf{I}', \mathcal{C}', \sigma')$, where $\mathbf{I}' = \mathbf{I}$ with column $j$ updated, $\mathcal{C}' = \mathcal{C}$ updated with $c_j$, $\sigma' = T_w$ with counters and
  states updated
```

Figure 3.16: FS-DSSE update protocol.

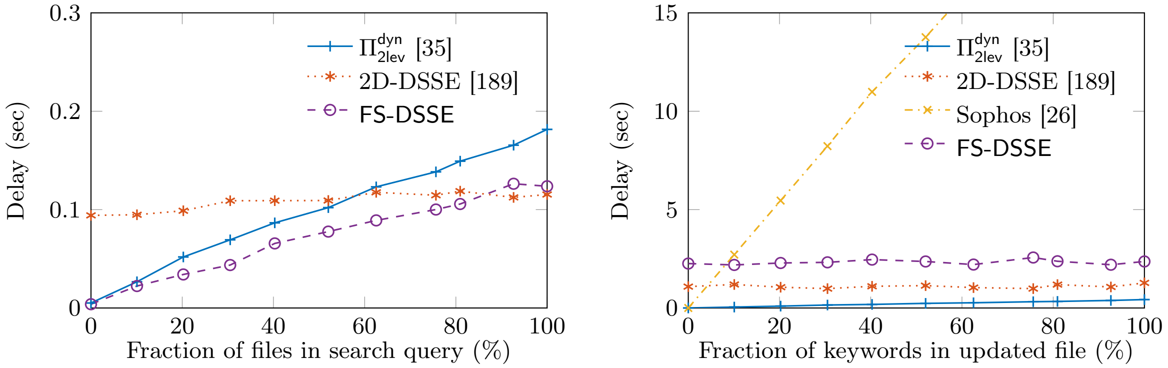
the server. This can be done by generating the row key with the incremental counter, given that the key generated with the current counter has been previously revealed to the server during the previous searches. Finally, the client sends the encrypted column and the encrypted files to the server, where the encrypted index and encrypted database are updated accordingly.

3.5.3 Security Analysis

Let $\mathcal{L}$ be a leakage function which captures information leakage in FS-DSSE including the maximum number of keywords and files, file IDs, the size of each file and access patterns. FS-DSSE achieves the following security.

Theorem 2. FS-DSSE is $(\mathcal{L})$-IND-CKA2 secure by Definition 1.

Proof (Sketch). The security of a DSSE is defined with the dynamic IND-CKA2 notion, which intuitively means that the search and update tokens sent from the client must not reveal any



(a) Keyword search

(b) File update

Figure 3.17: Latency of FS-DSSE and its counterparts.

information about the keywords being searched or updated. This notion pertains to leakage functions, which captures precisely what information is leaked from the ciphertext and the tokens. □

Lemma 1. *FS-DSSE is forward-private, in the sense that when the client conducts an update, this operation does not leak any information based on previous search operations [165].*

Proof (Sketch). The forward-privacy implies that the content of updated files should not be linked with any previous search operations. FS-DSSE harnesses the update strategy in [189], in which the update operation uses all fresh row keys which were never revealed to the server. It is achieved by increasing the keyword counter maintained in the hash table T_w. □

3.5.4 Experimental Evaluation

We first describe our implementation details, experimental setup and evaluation metrics with state-of-the-art schemes. We then present the performance of the proposed scheme along with in-depth comparison. Our code is publicly available at https://github.com/ozgurozmen/FS-DSSE

3.5.4.1 Configurations and Methodology

Our implementation uses the following libraries: tomcrypt for cryptographic primitives; Intel AES-NI for AES-CTR encryption acceleration; google-sparsehash for hash table; zeroMQ for network communication.

We used a Desktop with Intel Xeon E3-1231v3 @ 3.40 GHz and 16GB RAM at the client side. We leveraged our on-campus computing platform equipped with 32 CPUs @ 2.70GHz, and 512GB RAM as the server.

We used the full Enron email dataset, including 517401 files and 1728833 distinct keywords according to the standard tokenization method. The total number of keyword-file pairs is around 10^8.

We compare FS-DSSE with some state-of-the art DSSE schemes including the scheme in [189] (called as 2D-DSSE), Sophos [26] and $\Pi^{\mathsf{dyn}}_{\mathsf{2lev}}$ [35]. For Sophos and 2D-DSSE, we used their public open-source since their implementation setting is same with ours (C/C++), but we only simulated $\Pi^{\mathsf{dyn}}_{\mathsf{2lev}}$ due to its lack of open-source C/C++ implementation.

We evaluated all the schemes according to search and update delay, in terms of amortized costs. To compare the search time, we searched from least-common keywords (*e.g.*, only appears in a few, *e.g.*, 1-50, of the files) to most-common keywords (*e.g.*, appears in all, *e.g.*, 99-100% of the files) with 10% intervals. We applied the same strategy to compare the update times. We present the cost breakdown of search and update operations for our scheme.

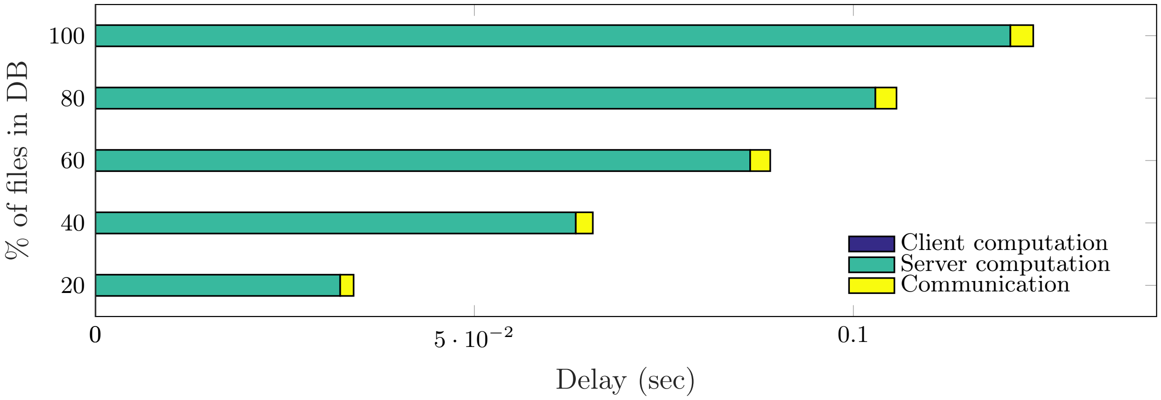


Figure 3.18: Cost breakdown of search query in FS-DSSE.

3.5.4.2 Overall Results

We present the end-to-end delays for search and update operations of FS-DSSE and its counterparts in Figure 3.17a and Figure 3.17b, respectively. Note that we excluded Sophos in the Figure 3.17a since it is beyond y-axis limit. FS-DSSE achieves the fastest search time among the counterparts in most cases, where it is even 1.4× faster than the most efficient yet forward-insecure scheme $\Pi^{\mathsf{dyn}}_{\mathsf{2lev}}$.

Since search complexity of 2D-DSSE is *linear with the maximum number of files in database,* its search time was constant in any size of search query results. Due to the sublinear property, FS-DSSE is faster than 2D-DSSE, for the most of the keywords, where it is up to 35× faster when searching least-common keywords. 2D-DSSE is only 8ms faster than FS-DSSE when searching the most-common keywords. The search time of Sophos could not fit into this graph due to its heavy public key operations. Specifically, we measured that end-to-end delay to be around 20 seconds even when searching for least-common keywords.

FS-DSSE has a constant update time similar to 2D-DSSE for all files with different number of keywords associated to, since they are both linear with the number of keywords in the database. The latency difference between them is that in FS-DSSE, we store T_w at the server instead of the client as in 2D-DSSE to achieve $\mathcal{O}(1)$ client storage, which incurs an extra round of communication overhead. On the other hand, update time of $\Pi^{\mathsf{dyn}}_{\mathsf{2lev}}$ is the fastest. It is due to the fact that $\Pi^{\mathsf{dyn}}_{\mathsf{2lev}}$ scheme has a smaller encrypted index size and therefore, random access is performed on a smaller memory region. Moreover, the random access cost dominates the total update cost in our scheme. The update cost of Sophos increases linearly with the number of keywords associated with the updated file. The cost is lower than FS-DSSE when file is associated with 8.18% of the total number of keywords and it is higher for the rest. Since the update cost of Sophos is dominated with the public key operations performed at the client-side, when the file is associated with 100% of keywords, it is 11.39× slower than FS-DSSE.

We also studied the detailed cost of search operation in FS-DSSE to observe the factors that had the most impact on the total delay. It is depicted in Figure 3.18 that total delay is mostly dominated by the server computation with the increasing number of files associated with the keyword. Even though server performs symmetric key encryption/decryption in parallel using 32 cores, it still dominates the total time since our network speed is extremely fast and the size of the dictionary D, that stores the indexes, highly increases. Since generating search token does not incur any expensive operations, the client computation cost is negligible.

Update cost of our scheme is constant as the number of keywords associated with the updated file increases (see Figure 3.17b). Our measurements showed that update cost of our scheme is dominated by the I/O access due to non-contiguous memory access. The second major dominating cost of update operation is the client computation. It requires the re-encryption of the encrypted index column with new keys to achieve forward-privacy. Since the network speed is fast in this experiment, the communication cost is lower than other factors.

Chapter 4: Oblivious RAM[3]

4.1 Introduction

ORAM, originally introduced by Goldreich [74], is a cryptographic protocol that allows a client to perform read/write operations over their personal data stored on an untrusted server without leaking any information regarding the requests' address (*i.e.*, access pattern). It has been shown in various application domains that sensitive information can be derived from the access pattern leakage, including data outsourcing [104, 111], secure enclaves [46, 178, 187] and searchable encryption [21, 34, 103, 117, 148, 156, 190]. Recent attempts have used ORAM to seal the access pattern leakages in many systems such as data outsourcing [20, 42, 48, 91], trusted execution environments [7, 8, 98, 149, 159], secure hardware [62, 63, 123, 127, 139, 151], multi-party computation [6, 56, 59, 76, 114, 180, 181] and block-chain [38, 84, 120].

Despite the strong security and privacy properties it offers, ORAM is known to incur high communication/computational cost. Specifically, there exists a logarithmic communication lower bound in any *passive* ORAM construction [29, 75, 119], where the server only acts as a storage entity (*i.e.*, no computation-capable). This overhead, however, has been shown costly for certain applications in the traditional client-server setting [20, 136]. To reduce the client bandwidth overhead, active ORAM schemes were introduced, in which the server can perform some sorts of secure computation [5, 52, 54, 77, 126, 130, 150, 166]. Several active ORAM constructions have successfully achieved a low (constant) client bandwidth overhead by using either Homomorphic Encryption (HE) [44] or

[3] This chapter was published in [87–89, 93–96]. Permission is included in Appendix A.

distributed computation model [166]. Although the HE performance has considerably been improved recently with efficient implementations, the use of HE in ORAM access still incur remarkable computation overhead at both the client and server sides. This results in a high access latency, which may negatively impact the user experience and degrade the quality of the cloud services. Moreover, most active ORAM schemes only consider em passive security in the sense that the adversary is assumed to follow the protocol faithfully [44, 130]. The cost to enable active security in the active ORAM schemes is high, especially in the single-server setting [54], which may not be suitable for real-life applications.

In many practical scenarios, it may not be possible to guarantee a reliable and high bandwidth network connection between the client and server. This is particularly true in the case of home networks and mobile devices with wireless network connectivity (*e.g.*, Wi-Fi, LTE). Given that ORAM with $\mathcal{O}(\log N)$ client bandwidth overhead (*e.g.*, Path-ORAM [169]) may not be suitable for such contexts, there is a significant need to design a new ORAM scheme that can achieve $\mathcal{O}(1)$ client bandwidth overhead. It is also important that the proposed ORAM is suitable for resource-limited clients and only incurs a low delay to provide a desirable quality of service. Moreover, in practice, it is likely that a malicious adversary can be present (*e.g.*, malware), who can deviate from the ORAM protocol to compromise the data integrity and the access pattern privacy. Therefore, *our objective is to create efficient ORAM frameworks that can simultaneously achieve (*i*) a low client-communication overhead (i.e.,* $\mathcal{O}(1)$ *bandwidth overhead), (*ii*) low computational overhead at both the client- and server-side, (*iii*) low storage, (*iv*) ability to securely compute on the accessed data and (*iv*) security against active adversaries.*

4.2 Related Work

The first ORAM proposed by Goldreich *et al.* [74] was in the context of software protection and followed by refinements [75]. Since then, several ORAM schemes have been proposed [75, 145]; however, many of these can achieve the logarithmic bandwidth overhead that was proven as the ORAM lower bound under $\mathcal{O}(1)$ blocks of client storage [75]. The recent ORAM schemes mainly have been considered in the client-server model to hide the data access pattern over a remote server [145]). In 2011, Shi *et al.* proposed a breakthrough in ORAM constructions by using a tree paradigm [161]. This tree paradigm led to efficient ORAM scheme proposals (*e.g.*, [39, 71, 150, 169, 179]) that can achieve the Goldreich-Ostrovsky logarithmic communication bound in [75]. The most simple and efficient ORAM based on tree-ORAM paradigm is Path-ORAM [169], where the client only needs to perform read and write operations over a data path, whereas the server only needs to provide storage functionality (*e.g.*, data sending and receiving only).

Tree-ORAM and Path-ORAM have been adapted to enable access pattern obliviousness in many applications such as secure processors [127], oblivious data structures [113, 154, 182], multi-party computation [179, 181] and oblivious storage [8, 43, 125, 135, 158, 186]. Several ORAM schemes were specifically designed for oblivious file systems [12, 22, 39, 128, 129]. Since all these systems rely on the Tree-ORAM paradigm, they incur the logarithmic communication overhead, which was shown costly for certain client-server applications [20, 136]. Recently, Larsen and Nielsen in [119] have re-confirmed the existence of the logarithmic bandwidth overhead in *passive* ORAM schemes (*i.e.*, the server is storage-only).

To reduce the communication overhead, the concept of *active* ORAM has been proposed, where the server can perform some computation. Ring-ORAM [150] reduced the communication cost of Path-ORAM by 2.5 times given that the server performs XOR computations. Path-PIR [130] used PIR [174] with Additively HE (AHE) [144]) on top of the Tree-ORAM [161]. [52] used PIR scheme [174] on top of ObliviStore [167], which is based on Partition-ORAM [168]. The TWORAM scheme [67] constructed a garbled circuit [188] over the tree ORAM structure, which allows the client and server to perform the secure computation to access the block. Although many active ORAM schemes have been presented [10, 52, 54, 61, 67, 130, 132, 150]), most of them either cannot surpass the logarithmic bound [130, 150] or were shown insecure (*i.e.*, [5, 132]. To the best of our knowledge, only active ORAM schemes that harness HE techniques [50, 70] can achieve the $\mathcal{O}(1)$ client bandwidth overhead under reasonably large block sizes (*e.g.*, $\mathcal{O}(\log^5 N)$ where N is the number of data blocks) [10, 54, 61]. Despite their communication efficiency, it has been shown in [89, 132] that performing HE computation during the ORAM access incurred significantly more latency than streaming $\mathcal{O}(\log N)$ data blocks as in passive ORAM schemes.

To improve the computation efficiency, ORAM has been explored in the distributed setting [5, 77, 89, 126, 166]. The first multi-server ORAM was proposed by Stefanov *et al.* [166], where the (single-server) Partition-ORAM [168] paradigm was transformed into the multi-server setting to achieve $\mathcal{O}(1)$ client bandwidth overhead and low computation at the servers. The main limitation of this scheme is that it incurs high client storage overhead (*i.e.*, $\mathcal{O}(\sqrt{N})$) due to the Partition-ORAM paradigm. Lu *et al.* [126] and Kushilevitz *et al.* [116] adapted the hierarchical ORAM construction in [75] to the multi-server setting to reduce the communication overhead. CHf-ORAM [131] attempted to use four non-colluding servers to achieve $\mathcal{O}(1)$ bandwidth blowup under $\mathcal{O}(1)$ blocks of client storage. However, CHf-ORAM [131] (as well as its predecessor [132]) was broken by Abraham *et al.* [5], which also showed an asymptotically tight sub-logarithmic communication bound for composing ORAM with PIR. Abraham *et al.* also proposed a two-server ORAM scheme [5], which exploits the XOR-PIR protocol [45] for the oblivious retrieval. Gordon *et al.* proposed a two-server ORAM scheme [77], which removes the need of updating the position map component in the tree-ORAM paradigm, thereby saving the factor of $\mathcal{O}(\log N)$ communication rounds incurred by accessing the position map recursively at the server. Very recently, Chan *et al.* proposed a perfectly secure 3-server ORAM scheme [40] based on the Hierarchical ORAM paradigm in [75]. Gordon *et al.* proposed a simple and efficient two-server tree-based ORAM [77], which achieves $\mathcal{O}(\log N)$ bandwidth overhead with $\mathcal{O}(1)$ communication round. In this scheme, the position map is static meaning that the path assigned for each data block is deterministic and unchanged, which can be computed by a pseudo-random function.

Another line of distributed ORAM research focuses on the context of multi-party computation in the RAM model [56, 59, 114, 181]. In these works, the access patterns are hidden from all parties so that such ORAM schemes are integrated with some secure computation protocol (*e.g.*, Yao's garbled circuit [188]) and, therefore, their cost is higher than classical client-server ORAM model. The aim is to perform secure computation in the RAM model where both instructions and functions are hidden from participants and thus ORAM is simply used as a building block. Due to the stronger privacy model, all these distributed ORAM schemes are less efficient than the distributed ORAM in the standard client-server setting.

Finally, in contrast to the generic ORAM (where both read and write patterns are hidden and indistinguishable), there exist some special ORAM schemes that conceal only either read or write patterns (but not both) [23, 155, 173]. Most of these constructions are more efficient than generic ORAMs since they target only on the specific operation type (read or write).

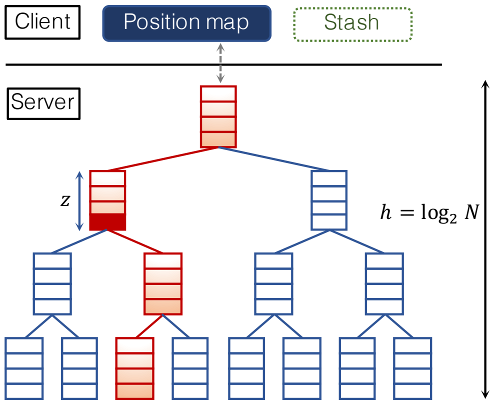


Figure 4.1: Tree-ORAM paradigm.

4.3 Preliminaries

Definition 5 (ORAM Security [161]). Let $\overrightarrow{o} = (\langle \mathsf{op}_1, \mathsf{idx}_1, \mathsf{data}_1 \rangle, \ldots, \langle \mathsf{op}_q, \mathsf{idx}_q, \mathsf{data}_q \rangle)$ be a sequence of data access requests on encrypted database, where $\mathsf{op}_i \in \{\mathsf{read}, \mathsf{write}\}$, idx_i is logical address to be read/written and data_i being the data at idx_i to be read/written (for $i \in \{1, \ldots, q\}$). Let $\mathbf{AP}(\overrightarrow{o})$ be an access pattern observed by the server $\mathcal{S}$ given a data request sequence $\overrightarrow{o}$. An ORAM scheme is secure if for any two data request sequences $\overrightarrow{o}_i$ and $\overrightarrow{o}_j$ of the same length, their access patterns $\mathbf{AP}(\overrightarrow{o}_i)$ and $\mathbf{AP}(\overrightarrow{o}_j)$ are computationally indistinguishable by anyone but the client.

4.3.1 Tree-ORAM

Most efficient ORAMs [52, 54, 150, 169] to-date follow the tree paradigm [161]. In this paradigm, there are two main components: A full binary tree data structure stored at the server side and a position map (denoted as pm) stored at the client side (Figure 4.1). The data blocks are organized into the tree, where each block is assigned to a path pid selected uniformly at random, and the position map is used to keep track of the location of each block in the tree. A tree with N leaves can store up to N data blocks. Each node in the tree is called a “bucket”, which has Z slots to contain data blocks. Each block b has a unique identifier id and all blocks are of the same size $|b|$

(*e.g.*, 4 KB). In the tree, since there are more slots than the number of blocks it can contain, all empty slots are filled with dummy data.

There are two main sub-protocols in the tree ORAM paradigm: *retrieval* and *eviction*. To access a block, the client first reads the block by executing the *retrieval* protocol on the path of the block stored in the position map. The client updates the block and assigns it to a new path selected uniformly at random. Finally, the client executes the *eviction* protocol on a random/deterministic path, which writes the block back to the top levels of the tree and obliviously pushes blocks down from top to bottom levels. ORAM schemes following the tree paradigm [54, 150, 169] all follow the aforementioned basic routine, and provides different trade-offs between communication and computation overhead.

The size of pm is $\mathcal{O}(N \log N)$. To reduce the client storage to $\mathcal{O}(1)$, it is possible to store pm remotely on the server using the recursive ORAM [161], which increases the number of communication rounds to $\mathcal{O}(\log N)$ for each access operation.

4.3.2 Path-ORAM

Stefanov *et al.* proposed Path-ORAM [169], the most efficient and simple ORAM scheme, which follows the tree paradigm by Shi *et al.* [161]. Apart from the position map and the binary tree, Path-ORAM requires an extra component called *stash* (**S**) tp temporarily store some data blocks at the client. To access a block with Path-ORAM, the client retrieves its path from pm and then performs a read operation (read) on its path, in which all real blocks in the path are fetched into **S**. The client updates the retrieved block with a new random path in pm and then performs an eviction operation (Evict) to push the blocks in S back to the read path such that each block resides somewhere in an intersection node between the read path and its assigned path toward the leaf. In Path-ORAM, the stash size $|\mathbf{S}|$ was proven to be upper-bounded by the security parameter λ as $|\mathbf{S}|= O(\lambda)$ blocks, which characterizes the overflow probability and statistical security.

4.3.3 Circuit-ORAM

Circuit-ORAM [179] further reduces the (circuit) complexity of Path-ORAM when implemented in the context of secure hardware (*e.g.*, Intel SGX) or multi-party computation by minimizing the number of blocks that are involved during the read and eviction operations. In this design, each bucket has a meta-data component to store the information about real blocks and their path ID that it contain.The retrieval and eviction of Circuit-ORAM work as follows.

To perform retrieval, similar to the original Tree-ORAM [161], the client reads all data in the path, but keeps only the block of interest in the stash and removes it from the path. The removal process can be implemented efficiently by flipping only one bit in the bucket meta-data.

To perform eviction, the client prepares a list of blocks to be pushed down in the eviction path by scanning the meta-data of buckets in the path. The client picks one block in the stash (if any) that can be pushed to the deepest level of the tree and then traverses from the root to the leaf node. In each level, the client drops the on-hold block and picks at most one block to be put into a deeper level. For each data access, the client invokes two eviction procedures, with the eviction path being selected randomly or deterministically, as in [71].

Circuit-ORAM has a smaller bucket size Z than Path-ORAM (*i.e.*, 2 *vs.* 4) and, therefore, it incurs less server storage overhead. However, Circuit-ORAM incurs approximately 1.25× more I/O accesses than Path-ORAM since each access operation requires two eviction operations. Similar to Path-ORAM, the stash size $|\mathbf{S}|$ in Circuit-ORAM was proven to be upper-bounded by the security parameter, *i.e.*, $|\mathbf{S}|= O(\lambda)$ blocks.

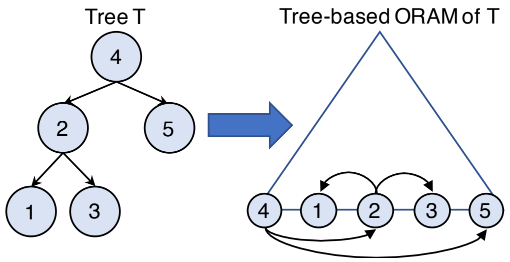


Figure 4.2: Oblivious data structure for a tree.

4.3.4 Write-Only ORAM

In contrast to the generic ORAM, where both read and write operations are hidden, Blass *et al.* [23] 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 the location of every block. Given a block to be written, 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 IND-CPA encrypted to hide which slot is updated. By selecting λ sufficiently large (*e.g.*, $\lambda = 80$), 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 $\mathbf{S}$ of size $\mathcal{O}(\log N)$, where N is the total number of data blocks, to temporarily store blocks that cannot be rewritten when all read slots are full.

4.3.5 Oblivious Data Structures

Oblivious Data Structure (ODS) proposed by Wang *et al.* [182] leverages "pointer techniques" to reduce the storage cost of position map components in non-recursive ORAM schemes to $\mathcal{O}(1)$, if the data to be accessed have some specific structures (*e.g.*, linked-list, grid, tree, etc.). For instance, given a binary search-sorted array as illustrated in Figure 4.2, the ORAM block is augmented with $k+1$ additional slots that hold the position of the block along with the positions and identifiers of its children as $b := (id, \mathsf{data}, \mathsf{pos}, \mathsf{childmap})$, where id is the block identifier, data is the block data, pos is its position in ORAM structure, and $\mathsf{childmap}$ is a miniature position map with entries (id_i, pos_i) for k children. To ensure that the $\mathsf{childmap}$ is up to date, a child block must be accessed through at most one parent at any given time. If a block does not have a parent (*e.g.*, the root of a tree), its position will be stored in the client. A parent block should never be written back to the server without updating positions of its children blocks.

4.3.6 Multi-Party Computation

Secret sharing scheme allows a secret value to be shared and computed securely among multiple untrusted parties. We briefly introduce several efficient secret sharing schemes as follows.

4.3.6.1 Shamir Secret Sharing

We recall (t, ℓ)-threshold Shamir Secret Sharing (SSS) scheme [160], which comprises two algorithms $\mathsf{SSS} = (\mathsf{Create}, \mathsf{Recover})$ as presented in Figure 4.3. To share a secret $\alpha \in \mathbb{F}_p$ among ℓ parties, a dealer generates a random polynomial f, where $f(0) = \alpha$ and evaluates $f(x_i)$ for party P_i for $1 \leq i \leq \ell$, where $x_i \in \mathbb{F}_p^*$ is a deterministic non-zero element of $\mathbb{F}_p$ that uniquely identifies party P_i and it is considered public information ($\mathsf{SSS.Create}$ algorithm). $f(x_i)$ is referred to as the share of party P_i, and it is denoted by $[\![\alpha]\!]_i$. To reconstruct the secret α, the shares of at least $t + 1$ parties have to be combined via Lagrange interpolation ($\mathsf{SSS.Recover}$ algorithm).

We extend the notion of secret share for a value into the share for a vector in a natural way as follows: 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 P_i, which is a vector whose elements are the shares of the elements in $\mathbf{v}$. Similarly, given a matrix $\mathbf{I}$, $[\![\mathbf{I}]\!]$ denotes the share of $\mathbf{I}$, which is also a matrix with each cell $[\![\mathbf{I}[i, j]]\!]$ being the share of the cell

```
SSS.Create$(\alpha, t)$:
$(a_1, \ldots, a_t) \xleftarrow{\$} \mathbb{F}_p$
for $i = 1, \ldots, \ell$ do
    $[\![\alpha]\!]_i \leftarrow \alpha + \sum_{j=1}^{t} a_j \cdot x_i^j$ # $x_i \in \mathbb{F}_p^*$: public identifier of party $P_i$
return $([\![\alpha]\!]_1, \ldots, [\![\alpha]\!]_\ell)$

SSS.Recover$(\mathcal{A}, t)$:
Randomly pick $t + 1 \leq \ell$ shares $\{[\![\alpha]\!]_{x_1}, \ldots, [\![\alpha]\!]_{x_{t+1}}\}$ in $\mathcal{A}$
$g(x) \leftarrow$ LagrangeInterpolation $\left(\{(x_i, [\![\alpha]\!]_{x_i})\}_{i=1}^{t+1}\right)$
$\alpha \leftarrow g(0)$
return $\alpha$
```

Figure 4.3: Shamir secret sharing scheme.

$\mathbf{I}[i, j]$. In some cases, to ease readability, we drop the subscript i, when the party is understood from the context.

Shamir [160] showed that SSS is information-theoretic secure and t-private in the sense that no set of t or less shares reveals any information about the secret. More precisely, $\forall m, m' \in \mathbb{F}_p$, $\forall \mathcal{I} \subseteq \{1, \ldots, \ell\}$ such that $|\mathcal{I}| \leq t$ and for any set $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{I}|}\}$ where $a_i \in \mathbb{F}_p$, the probability distributions of $\{s_{i \in \mathcal{I}} : (s_1, \ldots, s_\ell) \leftarrow \mathsf{SSS.Create}(m, t)\}$ and $\{s'_{i \in \mathcal{I}} : (s'_1, \ldots, s'_\ell) \leftarrow \mathsf{SSS.Create}(m', t)\}$ are identical and uniform:

$$\Pr(\{s_{i \in \mathcal{I}}\} = \mathcal{A}) = \Pr(\{s'_{i \in \mathcal{I}}\} = \mathcal{A}).$$

Ben-Or *et al.* [19] showed that SSS can be used to obtain t-private protocols. Lemma 2 summarizes the homomorphic properties of SSS and it was first described in [19].

Lemma 2 (SSS Homomorphic Properties [19]). Let $[\![\alpha]\!]_i^{(t)}$ be the Shamir share of value $\alpha \in \mathbb{F}_p$ with privacy level t for P_i. SSS offers additively and multiplicatively homomorphic properties:

- *Addition of two shares*

$$[\![\alpha_1]\!]_i^{(t)} + [\![\alpha_2]\!]_i^{(t)} = [\![\alpha_1 + \alpha_2]\!]_i^{(t)}. \tag{4.1}$$

- *Multiplication w.r.t a scalar* $c \in \mathbb{F}_p$

$$c \cdot [\![\alpha]\!]_i^{(t)} = [\![c \cdot \alpha]\!]_i^{(t)}. \tag{4.2}$$

- *Partial share multiplication*

$$[\![\alpha_1]\!]_i^{(t)} \cdot [\![\alpha_2]\!]_i^{(t)} = [\![\alpha_1 \cdot \alpha_2]\!]_i^{(2t)}. \tag{4.3}$$

The two-share partial multiplication (Equation 4.3) in Lemma 2 results in a share of $\alpha_1 \cdot \alpha_2$, which is t-private and represented by a $2t$-degree polynomial. It was first observed in [19] that the resulting polynomial is not uniformly distributed. In order to achieve the uniform distribution and computation consistency over $[\![\alpha_1 \cdot \alpha_2]\!]$, it is required to reduce the degree of the polynomial representation of $[\![\alpha_1 \cdot \alpha_2]\!]$ from $2t$ to t and re-share the polynomial. This multiplication operation with degree reduction can be achieved via the secure following multiplication protocol.

Gennaro *et al.* [68] presented a Secure Multi-pary Multiplication (SMM) protocol for two Shamir secret-shared values among multiple parties. Given $\alpha_1, \alpha_2 \in \mathbb{F}_p$ shared by (t, ℓ)-threshold SSS as $[\![\alpha_1]\!]_i^{(t)}$ and $[\![\alpha_2]\!]_i^{(t)}$ for $1 \leq i \leq \ell$ respectively, $2t+1$ parties P_i among ℓ parties would like to compute the multiplication of α_1, α_2 without revealing the value of α_1 and α_2. The protocol requires a Vandermonde matrix $\mathbf{V}_{\{x_i\}}$ of size $(2t+1) \times (2t+1)$ having the following structure.

```
• Input: $P_i$ owns $[\![\alpha_1]\!]_i^{(t)}$, $[\![\alpha_2]\!]_i^{(t)}$ and wants to compute $[\![\alpha_1 \cdot \alpha_2]\!]_i^{(t)}$
• Output: Each $P_i$ obtains $[\![\beta]\!]_i^{(t)}$, where $\beta = \alpha_1 \cdot \alpha_2$
for each $P_i \in \{P_1, \ldots, P_{2t+1}\}$ do
    $[\![\beta]\!]_i^{(2t)} \leftarrow [\![\alpha_1]\!]_i^{(t)} \cdot [\![\alpha_2]\!]_i^{(t)}$
    $([\![\beta]\!]_j^{(t)})_{j=1}^{\ell} \leftarrow \mathsf{SSS.Create}([\![\beta]\!]_i^{(2t)}, t)$
    Distribute $[\![\beta]\!]_j^{(t)}$ to all $P_j \in \{P_1, \ldots, P_{2t+1}\} \setminus P_i$
for each $P_i \in \{P_1, \ldots, P_{2t+1}\}$ do
    $[\![\beta]\!]_i^{(t)} \leftarrow \sum_{j=1}^{2t+1} \mathbf{V}^{-1}[1, j] \cdot [\![\beta]\!]_j^{(t)}$
```

Figure 4.4: Secure multi-party multiplication protocol on SSS shares.

$$\mathbf{V}_{\{x_1,\ldots,x_{2t+1}\}} = \begin{bmatrix} x_1^0 & x_1^1 & \ldots & x_1^{2t} \\ x_2^0 & x_2^1 & \ldots & x_2^{2t} \\ \vdots & \vdots & \ddots & \vdots \\ x_{2t+1}^0 & x_{2t+1}^1 & \ldots & x_{2t+1}^{2t} \end{bmatrix}, \tag{4.4}$$

where $x_i \in \mathbb{F}_p$ are unique identifiers of participating party P_i. We refer to $\mathbf{V}^{-1}$ as the inverse of Vandermonde matrix. Each party P_i locally multiplies $[\![\alpha_1]\!]_i^{(t)}$ and $[\![\alpha_2]\!]_i^{(t)}$ yielding $[\![\alpha_1 \cdot \alpha_2]\!]_i^{(2t)}$, and creates shares of $[\![\alpha_1 \cdot \alpha_2]\!]_i^{(2t)}$ by a new random polynomial of degree t for $2t+1$ parties and distributes them to other $2t$ parties. Finally, each party locally performs the dot product between the received shares and $\mathbf{V}_{\{x_i\}}^{-1}[1, *]$ to obtain a new share of $\alpha_1 \cdot \alpha_2$, which is now represented by a polynomial of degree t as $[\![\alpha_1 \cdot \alpha_2]\!]_i^{(t)}$. Figure 4.4 presents this multiplication protocol.

Lemma 3 (SMM Protocol Privacy [68]). The SMM protocol in [68] (denoted as $\star$ operator) offers homomorphic property for full multiplication between two SSS-shares, whose result is t-private as:

$$[\![\alpha_1 \cdot \alpha_2]\!]_i^{(t)} = [\![\alpha_1]\!]_i^{(t)} \star [\![\alpha_2]\!]_i^{(t)} \tag{4.5}$$

```
$(\langle s \rangle_0, \ldots, \langle s \rangle_{\ell-1}) \leftarrow \mathsf{AuthCreate}(\alpha, s, \ell)$:
$([\![s]\!]_0, \ldots, [\![s]\!]_{\ell-1}) \leftarrow \mathsf{ASSS.Create}(s, \ell)$
$([\![\alpha s]\!]_0, \ldots, [\![\alpha s]\!]_{\ell-1} \leftarrow \mathsf{ASSS.Create}(\alpha s, \ell)$
return $(\langle s \rangle_0, \ldots, \langle s \rangle_{\ell-1})$, where $\langle s \rangle_i \leftarrow ([\![s]\!]_i, [\![\alpha s]\!]_i)$

$s \leftarrow \mathsf{AuthRecover}(\alpha, (\langle s \rangle_0, \ldots, \langle s \rangle_{\ell-1}))$:
$s \leftarrow \mathsf{ASSS.Recover}([\![s]\!]_0, \ldots, [\![s]\!]_{\ell-1})$
$\sigma \leftarrow \mathsf{ASSS.Recover}([\![\alpha s]\!]_0, \ldots, [\![\alpha s]\!]_{\ell-1})$
if $\alpha s \neq \sigma$ then return $\perp$
return $s$
```

Figure 4.5: Authenticated additive secret sharing.

4.3.6.2 Additive Secret Sharing

Additive secret sharing scheme (ASSS) comprises two algorithms $\mathsf{ASSS} = (\mathsf{Create}, \mathsf{Recover})$ as follows.

- $(s_0, \ldots, s_{\ell-1}) \leftarrow \mathsf{ASSS.Create}(s, \ell)$: Given a secret $s \in \mathbb{F}_p$ and a number of parties ℓ as input, it outputs random values s_i as the shares for ℓ parties such that $s = \sum_i s_i$. We denote the additive share of a value s for party P_i as $[\![s]\!]_i$, *i.e.*, $[\![s]\!]_i = s_i$.

- $s \leftarrow \mathsf{ASSS.Recover}(s_0, \ldots, s_{\ell-1})$: Given ℓ shares as input, it returns the secret as $s \leftarrow \sum_i s_i$.

Additive secret sharing is t-*private* in the sense that no set of t or fewer shares reveals any information about the secret. More formally, $\forall s, s' \in \mathbb{F}_p$, $\forall \mathcal{L} \subseteq \{0, \ldots, \ell-1\}$ such that $|\mathcal{L}| \leq t$ and for any $\mathcal{S} = \{s_0, \ldots, s_{|\mathcal{L}|-1}\}$ where $s_i \in \mathbb{F}_p$, the probability distributions of $\{s_{i \in \mathcal{L}} : (s_0, \ldots, s_{\ell-1}) \leftarrow \mathsf{ASSS.Create}(s, \ell)\}$ and $\{s'_{i \in \mathcal{L}} : (s'_0, \ldots, s'_{\ell-1}) \leftarrow \mathsf{ASSS.Create}(s', \ell)\}$ are identical and uniform.

Additive secret sharing offers additive homomorphic properties as follows. Given additive shares $[\![s_1]\!]$ and $[\![s_2]\!]$ and $c \in \mathbb{F}_p$, each party can locally compute the additive share of addition and scalar multiplication as $[\![s_1 + s_2]\!] \leftarrow [\![s_1]\!] + [\![s_2]\!]$ and $[\![cs]\!] \leftarrow c[\![s]\!]$.

Replicated Secret Sharing (RSS) scheme enables homomorphic multiplication over additive shares with information-theoretic security [105]. In the three-party setting, each party $S_i \in \{S_0, S_1, S_2\}$ stores two additive shares of a secret $s \in \mathbb{F}_p$, $\llbracket s \rrbracket_i$ and $\llbracket s \rrbracket_{i+1}$[4]. To compute $\llbracket uv \rrbracket$ from $\llbracket u \rrbracket$ and $\llbracket v \rrbracket$, RSS proceeds as follows. First, each party S_i (locally) computes $x_i \leftarrow \llbracket u \rrbracket_i \llbracket v \rrbracket_i + \llbracket u \rrbracket_i \llbracket v \rrbracket_{i+1} + \llbracket u \rrbracket_{i+1} \llbracket v \rrbracket_i$ and represents x_i with the addition of random values as $x_i = r_0^{(i)} + r_1^{(i)} + r_2^{(i)}$. Each S_i retains $(r_i^{(i)}, r_{i+1}^{(i)})$ and sends $(r_{i-1}^{(i)}, r_i^{(i)})$ and $(r_i^{(i)}, r_{i+1}^{(i)})$ to other parties S_{i-1} and S_{i+1}, respectively. Finally, each S_i obtains the shares of multiplication result by (locally) computing $\llbracket uv \rrbracket_i \leftarrow r_i^{(0)} + r_i^{(1)} + r_i^{(2)}$ and $\llbracket uv \rrbracket_{i+1} \leftarrow r_{i+1}^{(0)} + r_{i+1}^{(1)} + r_{i+1}^{(2)}$.

We recall the authenticated secret sharing in [51], in which each secret s is attached with an information-theoretic Message Authenticated Code (MAC) computed as αs, where α is a global MAC key owned by the dealer. We denote the authenticated share of a secret s as $\langle \cdot \rangle$, which contains the additive share of s and the additive share of αs as $\langle s \rangle = (\llbracket s \rrbracket, \llbracket \alpha s \rrbracket)$, where $\llbracket \alpha s \rrbracket$ is created in the same manner as $\llbracket s \rrbracket$. Figure 4.5 presents the algorithms to create authenticated shares and recover the secret. We present a homomorphic multiplication protocol with malicious security, which follows the pre-computation model [51, 112] using Beaver multiplication triples [14] of the form (a, b, c), where $c = ab$. In this setting, each untrusted party S_i owns a share of the MAC key as $\llbracket \alpha \rrbracket_i$. In the offline phase, all untrusted parties harness homomorphic encryption and zero-knowledge protocols [51, 112] to compute the authenticated share of the Beaver triple and its MAC in such a way that no party learns about (a, b, c) and α. To this end, each S_i obtains $(\langle a \rangle_i, \langle b \rangle_i, \langle c \rangle_i)$, where $\langle a \rangle_i = (\llbracket a \rrbracket_i, \llbracket \alpha a \rrbracket_i)$ and so forth. In the online phase, given $\langle u \rangle = (\llbracket u \rrbracket, \llbracket \alpha u \rrbracket)$ and $\langle v \rangle = (\llbracket v \rrbracket, \llbracket \alpha v \rrbracket)$ and all parties want to compute $\langle uv \rangle$, each S_i first (locally) computes $\llbracket \epsilon \rrbracket_i \leftarrow \llbracket u \rrbracket_i - \llbracket a \rrbracket_i$, and $\llbracket \rho \rrbracket_i \leftarrow \llbracket v \rrbracket_i - \llbracket b \rrbracket_i$. All parties come together to open ϵ and ρ by each S_i broadcasting $\llbracket \epsilon \rrbracket_i$ and $\llbracket \rho \rrbracket_i$. Finally, each S_i

[4] We note that the subscript index in this case is modulo 3.

```
$(\rho_1, \ldots, \rho_\ell) \leftarrow \mathsf{PIR}^{\mathsf{xor}}.\mathsf{CreateQuery}(j)$: Create select query for a database size $n$
Initialize binary string $e \leftarrow 0^n$ and set $e[j] \leftarrow 1$
for $i = 1, \ldots, \ell - 1$ do
    $\rho_i \leftarrow \{0,1\}^n$
$\rho_\ell \leftarrow \rho_1 \oplus \ldots \oplus \rho_{\ell-1} \oplus\ e$
return $(\rho_1, \ldots, \rho_\ell)$
```

```
$r_i \leftarrow \mathsf{PIR}^{\mathsf{xor}}.\mathsf{Retrieve}(\rho_i, \mathbf{B})$: Retrieve an item in $\mathbf{B}$
Parse $\mathbf{B}$ as $(b_1, \ldots b_n)$
Initialize $r_i \leftarrow \{0\}^m$
for $j = 1, \ldots, n$ do
    if $\rho_i[j] = 1$ then
        $r_i \leftarrow r_i \oplus b_j$
return $r_i$
```

```
$b \leftarrow \mathsf{PIR}^{\mathsf{xor}}.\mathsf{Reconstruct}(\mathcal{R})$: Reconstruct the item
$b \leftarrow r_1 \oplus \ldots \oplus r_\ell$, where $r_i \in \mathcal{R}$ for $1 \leq i \leq \ell$
return $b$
```

Figure 4.6: XOR-based PIR.

(locally) computes the authenticated share of the multiplication as $\langle uv \rangle_i = ([\![uv]\!]_i, [\![\alpha uv]\!]_i)$, where $[\![uv]\!]_i \leftarrow [\![c]\!]_i + \epsilon[\![b]\!]_i + \rho[\![a]\!]_i + \epsilon\rho$ and $[\![\alpha uv]\!]_i \leftarrow [\![\alpha c]\!]_i + \epsilon[\![\sigma b]\!]_i + \rho[\![\sigma b]\!]_i + \epsilon\rho[\![\alpha]\!]_i$. At the end of the protocol, all parties can verify the integrity of opened values as follows. Let x_j be an opened value and $[\![\alpha x_j]\!]_i$ is the share of its MAC to S_i. Let b_j be a random value that all parties agree on. Each S_i locally computes $x \leftarrow \sum_j b_j x_j$, $[\![y]\!]_i \leftarrow \sum_j b_j [\![\alpha x_j]\!]_i$ and $[\![\omega]\!]_i \leftarrow [\![y]\!]_i - x[\![\alpha]\!]_i$. All parties come together to open ω as $\omega \leftarrow \sum_i [\![\omega]\!]_i$. If $\omega = 0$, all the opened values pass the integrity check.

4.3.7 Private Information Retrieval

Private Information Retrieval (PIR) enables retrieval of a data item from an (unencrypted) public database without revealing which item being fetched. A multi-server PIR [17, 73] is defined as follows.

Definition 6 (Multi-Server PIR [17, 45, 73]). Let $\mathsf{DB} = (b_1, \ldots, b_n)$ be a database consisting of n items being stored in ℓ servers. A multi-server PIR protocol consists of three algorithms: $\mathsf{PIR} = (\mathsf{CreateQuery}, \mathsf{Retrieve}, \mathsf{Reconstruct})$. Given an item b_i in DB to be retrieved, the client creates

```
$([\![\mathbf{e}]\!]_1, \dots, [\![\mathbf{e}]\!]_\ell) \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{CreateQuery}(j)$: Create select queries
Let $\mathbf{e} := (e_1, \dots, e_n)$, where $e_j \leftarrow 1$, $e_i \leftarrow 0$ for $1 \le i \ne j \le n$
for $i = 1, \dots, n$ do
    $([\![e_i]\!]_1, \dots, [\![e_i]\!]_\ell) \leftarrow \mathsf{SSS.Create}(e_i, t)$
for $l = 1, \dots, \ell$ do
    $[\![\mathbf{e}]\!]_l \leftarrow ([\![e_1]\!]_l, \dots, [\![e_n]\!]_l)$
return $([\![\mathbf{e}]\!]_1, \dots, [\![\mathbf{e}]\!]_\ell)$

$[\![b]\!]_i \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Retrieve}([\![\mathbf{e}]\!]_i, \mathbf{B})$: Retrieve the item
$[\![b]\!]_i \leftarrow [\![\mathbf{e}]\!]_i \cdot \mathbf{B}$
return $[\![b]\!]_i$

$b \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\mathcal{B}, t)$: Recover the retrieved item from the set of answers $\mathcal{B}$
$b \leftarrow \mathsf{SSS.Recover}(\mathcal{B}, t)$
return $b$
```

Figure 4.7: SSS-based PIR.

queries $(e_1, \dots, e_\ell) \leftarrow \mathsf{PIR.CreateQuery}(i)$ and distributes e_j to server S_j. Each server responds with an answer $a_j \leftarrow \mathsf{PIR.Retrieve}(e_j, \mathsf{DB})$. Upon receiving ℓ answers, the client computes the value of item b_i by invoking the reconstruction algorithm $b \leftarrow \mathsf{PIR.Reconstruct}(a_1, \dots, a_\ell)$.

The security of the protocol is defined in terms of *correctness* and *privacy*. A multi-server PIR protocol is *correct* if the client computes the correct value of b from any ℓ answers via PIR.Reconstruct algorithm with probability 1. The concept of t-privacy for protocols is applied naturally to the PIR setting and follows directly from the t-privacy of SSS and the fact that among the servers they only have access to t shares of the query vector [73].

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

- *XOR-Based PIR [45]:* It relies on XOR trick to perform the private retrieval, in which the database $\mathbf{B}$ contains n items b_i, each being interpreted as a m-bit string (Figure 4.6).
- *SSS-Based PIR [17, 73]:* It relies on SSS to improve the robustness of multi-server PIR, in which the database $\mathbf{B}$ contains n items b_i, each being interpreted as an element of $\mathbb{F}_p$ (Figure 4.7).

Table 4.1: Summary of $\mathsf{S^3ORAM}$ schemes and some of their counterparts.

Scheme	Bandwidth Overhead†		Block Size*	Server Computation	Client Block Storage‡	# servers
	Client-server	Server-server				
Path-ORAM [169]	$\mathcal{O}(\log N)$	-	$\Omega(1)$	-	$\mathcal{O}(\log N)$	1
Ring-ORAM [150]	$\mathcal{O}(\log N)$	-	$\Omega(1)$	XOR	$\mathcal{O}(\log N)$	1
Onion-ORAM [54]	$\mathcal{O}(1)$	-	$\Omega(\log^5 N)$	Additively HE [50]	$\mathcal{O}(1)$	1
Dist. OblivStore [166]	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$	$\Omega(1)$	Permutation and IND-CPA encryption	$\mathcal{O}(\sqrt{N})$	2
2-Server ORAM [126]	$\mathcal{O}(\log N)$	-	$\Omega(1)$	Permutation and cuckoo hashing	$\mathcal{O}(1)$	2
$\mathsf{S^3ORAM^O}$	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$	$\Omega(\log^2 N)$	Secure addition and multiplication of SSS values	$\mathcal{O}(1)$	3
$\mathsf{S^3ORAM^C}$	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$	$\Omega(\log N)$		$\mathcal{O}(\log N)$	

† *Bandwidth overhead* denotes the number of blocks being transmitted between the client and the server(s) or between the servers. Due to the eviction, the server-server bandwidth overhead of $\mathsf{S^3ORAM^O}$ is $\mathcal{O}(\lambda \log N)$, where λ is the statistical security parameter. Since the eviction is performed every $\lambda/2$ access requests, the amortized server-server bandwidth overhead of $\mathsf{S^3ORAM^O}$ is $\mathcal{O}(\log N)$.

$*$ This indicates the minimal block size needed to absorb the transmission cost of the retrieval query and the eviction data, thereby meeting the expected client-bandwidth overhead. In this table, we consider all the ORAM schemes in the *non-recursive* form, where the position map is stored at the client.

‡ *Client block storage* is defined as the number of data blocks being temporarily stored at the client. This is equivalent to the stash component used in [150, 169], which, therefore, does not include the cost of storing the position map of size $\mathcal{O}(N \log N)$. Notice that all the ORAM schemes in this table, except the one in [126], require the position map component.

4.4 $\mathsf{S^3ORAM}$: A Multi-Server ORAM Framework with Constant Client-Bandwidth

We present $\mathsf{S^3ORAM}$, a new multi-server ORAM framework, which features $\mathcal{O}1$ client bandwidth blowup, low storage and efficient computation at both client- and server-side. Our proposed framework consists of two multi-server active ORAM schemes including $\mathsf{S^3ORAM^O}$ and $\mathsf{S^3ORAM^C}$, in which the former minimizes the client storage requirement while the latter optimizes the computation and storage overhead at the server-side. We first present our main idea and then outline the desirable properties of our proposed framework as follows.

Most efficient ORAM schemes to-date follow the tree paradigm by Shi *et al.* [161]. In this paradigm, there are two main procedures for each ORAM access: retrieval and eviction. Our intuition is to harness the homomorphic properties of Shamir secret sharing along with a secure multi-party multiplication protocol to perform these procedures in an oblivious manner. To achieve $\mathcal{O}(1)$ client-bandwidth overhead, it is imperative to ensure that each procedure only incurs a small constant number of data blocks to be transmitted between the client and the server(s). In the standard (single-server) ORAM setting, we observe that both Onion-ORAM [54] and Circuit-ORAM [179] schemes require low client storage and offer elegant retrieval and eviction strategies that can be further implemented with SSS homomorphic computation to achieve $\mathcal{O}(1)$ client-bandwidth overhead. Therefore, the main idea of $\mathsf{S^3ORAM^O}$ and $\mathsf{S^3ORAM^C}$ schemes in our $\mathsf{S^3ORAM}$ framework is to harness SSS and SMM protocol to perform the retrieval and eviction operations in the line of Onion-ORAM and Circuit-ORAM, respectively, but in a significantly more computation- and client bandwidth-efficient manner. By doing this, $\mathsf{S^3ORAM^O}$ (*resp.* $\mathsf{S^3ORAM^C}$) inherits all desirable properties of Onion-ORAM (*resp.* Circuit-ORAM) regarding the low client storage cost, while achieving $\mathcal{O}(1)$ client-bandwidth overhead *without* the costly homomorphic operations but instead requiring only a lightweight computation and suitability for small block sizes. Table 4.1 outlines a high-level comparison of $\mathsf{S^3ORAM}$ and its counterparts.

Our $\mathsf{S^3ORAM}$ framework offers the following properties.

- *Low client-server communication:* All schemes in $\mathsf{S^3ORAM}$ framework offer $\mathcal{O}(1)$ client bandwidth blowup, compared with $\mathcal{O}(\log N)$ of Path-ORAM [169] and Ring-ORAM [150] (with a fixed number of servers). $\mathsf{S^3ORAM}$ schemes feature a smaller block size (*i.e.*, $\Omega(\log^2 N)$ in $\mathsf{S^3ORAM^O}$, $\Omega(\log N)$ in $\mathsf{S^3ORAM^C}$), than state-of-the-art $\mathcal{O}(1)$ bandwidth blowup ORAM schemes that require Fully or Partially HE operations (*e.g.*, $\Omega(\log^5 N)$ in Onion-ORAM [54], $\Omega(\log^6 N)$ in Bucket-ORAM [61]).

- *Low client and server computation:* $\mathsf{S^3ORAM}$ schemes require the servers to perform only lightweight modular additions and multiplications, which are much more efficient than partial HE operations (*e.g.*, [50]). In particular, we show in §4.4.5 that, the server computation of $\mathsf{S^3ORAM}$ schemes is three orders of magnitude faster than that of Onion-ORAM.

  The client in $\mathsf{S^3ORAM}$ schemes only performs lightweight computations for retrieval and eviction operations. Thus, it is more efficient than Onion-ORAM, which requires a number of HE operations. For example, S^{3}ORAM requires only a few milliseconds compared to minutes of Onion-ORAM to generate an encrypted access query. Moreover, since data blocks in S^{3}ORAM schemes are single-layered "encrypted", the "decryption" process is less costly so that it is faster than other ORAMs (*e.g.*, [54, 166]), whose blocks are multi-layered encrypted.

- *Low end-to-end delay:* Due to low bandwidth and computation overhead, S^{3}ORAM schemes are approximately three orders of magnitude faster than Onion-ORAM, while it is one order of magnitude faster than Path-ORAM in networks with *moderate* client bandwidth. We notice that for S^{3}ORAM to provide all its advantages, it is assumed that good network throughput is available between the servers. In our detailed analysis in §4.4.5.5, we show that if the inter-server bandwidth is limited and the client has access to a high-speed Internet connection, state-of-the-art (single-server) ORAM schemes (*i.e.*, Path-ORAM, Ring-ORAM) are more efficient than S^{3}ORAM (see §4.4.5.5 for a detailed analysis).

- *Low client storage:* S^{3}ORAM$^\mathsf{O}$ scheme features $\mathcal{O}(1)$ blocks of client storage, compared with $\mathcal{O}(\lambda)$ blocks in Path-ORAM/Ring-ORAM, and $\mathcal{O}(\sqrt{N})$ blocks in [168]. S^{3}ORAM$^\mathsf{C}$ scheme achieves the same block of client storage with Path-ORAM/Ring-ORAM (*i.e.*, $\mathcal{O}(\lambda)$)

- *High security*: All S^{3}ORAM schemes achieve information-theoretic statistical security. The statistical bit comes from the tree-paradigm by Shi *et al.* [161]. The information-theoretic property comes from SSS and its multi-party multiplication protocol.

- *Full-fledged implementation and experiments:* We fully implemented S^{3}ORAM$^\mathsf{C}$ and S^{3}ORAM$^\mathsf{O}$ schemes in our S^{3}ORAM framework and evaluated their performance in an actual cloud environment (*i.e.*, Amazon EC2). The detailed experiments showed that both S^{3}ORAM schemes are efficient in practice and they can be deployed on mobile devices with a limited computation capacity and low network connection. We have released the source code of S^{3}ORAM framework for public use and testing.

S^{3}ORAM harnesses the distributed setting to achieve constant client bandwidth overhead with efficient computation and the server-side simultaneously. It should be clear, however, that the use of standard secret sharing techniques and, in particular, Shamir secret sharing, renders our protocol vulnerable to collusion attacks (as it is standard in this setting). It should be observed that this *vulnerability* does not exist in the standard single-server ORAM model. Therefore, as it is also standard in this model, we note that S^{3}ORAM cannot offer any security guarantee if the number of colluding servers exceeds the privacy threshold. Another limitation of S^{3}ORAM is that it only offers security in the semi-honest setting (see [54] for the exemplified active attack).

4.4.1 System and Security Models

4.4.1.1 System Model

Following the literature in distributed secure computation (*e.g.*, [19, 73]), we assume a synchronous network, which consists of a client and $\ell \geq 2t+1$ semi-honest servers $\mathcal{S} = \{S_1, \ldots, S_\ell\}$. It is also assumed that the channels between all the players are pairwise-secure, *i.e.*, no player can tamper with, read, or modify the contents of the communication channel of other players. We assume that all parties behave in an "honest-but-curious" manner in which parties always send messages as expected but try to learn as much as possible from the shared information received or observed. Notice that we *do not* allow parties to provide malicious inputs, *i.e.*, parties are not allowed to behave in a Byzantine manner.

A protocol is t-private [19] (see [73] for similar definitions in the context of distributed PIR) if any set of at most t parties cannot compute after the protocol execution more than they could compute individually from their set of private inputs and outputs. Alternatively, the parties have not "learned" anything. Our protocols in general, offer information-theoretic guarantees unless something is said explicitly to the contrary. This implies that our solutions are secure against computationally unbounded adversaries. As it is standard, we require that all computations by the servers and client be polynomial time and efficient. Finally notice that not only is the interaction between the servers and client performed in such a way that information-theoretic security is guaranteed but also the database being accessed is shared among the servers in a way that no coalition of up to t servers can find anything about the database contents (also in an information-theoretic manner).

4.4.1.2 Security Model

We now define the security of multi-server ORAM in the semi-honest setting proposed in [5] as a straightforward extension of the definition in [5] to the multi-server setting.

Definition 7 (Multi-Server Active ORAM). Let $\mathbf{x} = ((\mathsf{op}_1, \mathsf{id}_1, \mathsf{data}_1), \ldots, (\mathsf{op}_q, \mathsf{id}_q, \mathsf{data}_q))$ be a data request sequence of length q, where $\mathsf{op}_j \in \{\mathsf{Read}, \mathsf{Write}\}$, id_j is the identifier to be read/written and data_j is the data identified by id_j to be read/written. Let $ORAM_j(\mathbf{x})$ represent the ORAM client's sequence of interactions with the server S_i given a data request sequence $\mathbf{x}$.

A multi-server ORAM is correct if for any sequence $\mathbf{x}$, $\{ORAM_1(\mathbf{x}), \ldots, ORAM_\ell(\mathbf{x})\}$ returns data consistent with $\mathbf{x}$ except with a negligible probability.

A multi-server ORAM is t-secure if $\forall \mathcal{I} \subseteq \{1, \ldots, \ell\}$ such that $|\mathcal{I}| \leq t$, for any two data access sequences $\mathbf{x}, \mathbf{y}$ with $|\mathbf{x}| = |\mathbf{y}|$, their corresponding transcripts $\{ORAM_{i\in\mathcal{I}}(\mathbf{x})\}$ and $\{ORAM_{i\in\mathcal{I}}(\mathbf{y})\}$ observed by a coalition of up to t servers $\{S_{i\in\mathcal{I}}\}$ are (perfectly/statistically/computationally) indistinguishable.

Table 4.2: S^{3}ORAM notation.

Symbol	Description
$\mathbf{T}$	S^{3}ORAMtree structure.
Z	Bucket size.
$\mathbf{T}[i], \mathbf{T}[i][j]$	i-th bucket of S^{3}ORAMtree $\mathbf{T}$ and j-th slot in the i-th bucket of $\mathbf{T}$.
$\lvert b\rvert, b, c$	Block size, block and block chunk, respectively.
N, m	Number of blocks and number of chunks in a block.
H	Height of the S^{3}ORAMtree.
pm	Position map.
$(\mathsf{pid}, \mathsf{pIdx}) \leftarrow \mathsf{pm}[\mathsf{id}]$	Precise location (*i.e.*, path ID and path index) of the block id.
$\mathcal{I} \leftarrow \mathcal{P}(\mathsf{pid})$	Set of indexes of buckets residing in the path pid.

4.4.2 The Proposed S^{3}ORAM Framework

S^{3}ORAM follows the typical procedure of tree-based ORAMs [161]. Specifically, given a block to be accessed, the client first retrieves it from the outsourced ORAM structure via a secure retrieval operation. The retrieved block is then assigned to a random path, and written back to the root bucket. Finally, an eviction operation is performed in order to percolate data blocks to lower levels in the ORAM structure. *The intuition behind* S^{3}ORAM *access protocol is as follows:* (1) We integrate SSS with a multi-server PIR protocol to perform a private retrieval operation with some homomorphic properties; (2) We leverage these homomorphic properties of SSS and a SMM protocol to perform block permutation and to preserve t-privacy level of ORAM structure in the eviction phase, without relying on costly partial HE operations. Notice that the idea of using PIR to implement the ORAM retrieval phase was first suggested in [130], and later in some subsequent works such as [5, 52, 54, 77]. In Table 4.2, we outline the notation used in the S^{3}ORAM schemes and throughout the rest of the dissertation.

4.4.2.1 Data Structure

S^{3}ORAM schemes follow the tree paradigm proposed by Shi *et al.* [161] (see §4.3.1), in which the outsourced database is split into size-equal blocks and then organized to a balanced binary tree

```
S^3ORAM.Setup(DB):
1: Split DB into blocks $(b_1,\ldots,b_N)$ with corresponding IDs $(\mathsf{id}_1,\ldots,\mathsf{id}_N)$
2: $\mathbf{T}[i][j] \leftarrow \{0\}^{|b|}$ for $1 \le i < 2^{H+1}$ and $1 \le j \le Z$
3: for $i = 1,\ldots,N$ do
4:     $z_i \xleftarrow{\$} \{1,\ldots,2^H\}$
5:     Put $b_i$ into an empty slot indexed $y$ of the leaf bucket in path $z_i$
6:     $\mathsf{pm}[\mathsf{id}_i] \leftarrow (z_i, H \cdot Z + y)$
7: for $i = 1,\ldots,2^{H+1}-1$ do
8:     for $j = 1,\ldots,Z$ do
9:         $(c_{i,j}^{(1)},\ldots,c_{i,j}^{(m)}) \leftarrow \mathbf{T}[i,j]$, where $c_{i,j}^{(k)} \in \mathbb{F}_p$
10:        $(\llbracket c_{i,j}^{(k)}\rrbracket_1,\ldots,\llbracket c_{i,j}^{(k)}\rrbracket_\ell) \leftarrow \mathsf{SSS.Create}(c_{i,j}^{(k)},t)$ for $1 \le k \le m$
11:        $\llbracket\mathbf{T}[i,j]\rrbracket_l \leftarrow (\llbracket c_{i,j}^{(1)}\rrbracket_l,\ldots,\llbracket c_{i,j}^{(m)}\rrbracket_l)$ for $1 \le l \le \ell$
12: return $(\llbracket\mathbf{T}\rrbracket_1,\ldots,\llbracket\mathbf{T}\rrbracket_\ell)$ # Send $\llbracket\mathbf{T}\rrbracket_i$ to $S_i$ for $1 \le i \le \ell$
```

Figure 4.8: $\mathsf{S}^3\mathsf{ORAM}$ setup algorithm.

($\mathbf{T}$) with a height of H. Each node in $\mathbf{T}$ is called a *bucket* with Z slots so that it can store up to Z data blocks. Thus, $\mathbf{T}$ can store up to $N < Z \cdot 2^H$ data blocks.

At the client-side, the client maintains a position map component (pm) to keep track of the assigned path (pid) for each data block in the tree. Additionally, the client stores the location of each data block in its assigned path. Hence, pm is of structure $\mathsf{pm} := (\mathsf{id}, \langle \mathsf{pid}, \mathsf{pIdx} \rangle)$, where id is the block ID, $1 \le \mathsf{pid} \le 2^H$ is the assigned path of the block, and $1 \le \mathsf{pIdx} \le Z \cdot (H+1)$ is the location of the block in its path. The client also maintains a so-called stash component (denoted as $\mathbf{S}$) to temporarily store accessed block(s) from the tree.

In the $\mathsf{S}^3\mathsf{ORAM}$ framework, the tree structure is SSS-shared among ℓ servers. Figure 4.8 presents the Setup algorithm to construct data structures in $\mathsf{S}^3\mathsf{ORAM}$ schemes given a database input DB. First, the client organizes DB into N data blocks, and then initializes every slot in each bucket of the tree ($\mathbf{T}$) with a 0's string of length $|b|$ (lines 1-2). The client arranges all blocks into $\mathbf{T}$, wherein each block (b_i) is independently assigned to a random leaf bucket of $\mathbf{T}$. Notice that $|b|$ can be larger than $\lceil \log_2 p \rceil$ and therefore, it might not be suitable for arithmetic computation over $\mathbb{F}_p$. To address this, the client splits the data in each slot of $\mathbf{T}$ into equal-sized chunks $c_j \in \mathbb{F}_p$ (line

```
$\mathsf{S^3ORAM.Access}(\mathsf{op}, \mathsf{id}, b^*)$:
1: $b \leftarrow \mathsf{S^3ORAM.Retrieve}(\mathsf{id})$
2: $\mathsf{pm}[\mathsf{id}].\mathsf{pid} \xleftarrow{\$} \{1, \ldots, k^{H+1}\}$
3: if $\mathsf{op} = \mathsf{write}$ then
4:     $b \leftarrow b^*$
5: $\mathbf{S} \leftarrow \mathbf{S} \cup b$
6: Execute $\mathsf{S^3ORAM.Evict}()$
7: return $b$
```

Figure 4.9: General access procedure in S^{3}ORAM schemes.

9)[5]. Finally, the client creates shares of $\mathbf{T}$ via SSS.Create algorithm for each chunk in each slot in $\mathbf{T}$ (line 10). The S^{3}ORAM distributed data structure consists of ℓ shares of $\mathbf{T}$ as $\{[\![\mathbf{T}]\!]_1, \ldots, [\![\mathbf{T}]\!]_\ell\}$.

Figure 4.9 presents the general access operation of S^{3}ORAMschemes following the tree-ORAM paradigm. Basically, there are two main subroutines in the S^{3}ORAM.Access algorithm: Retrieve (line 1) and Evict (line 6). The former is to obliviously retrieve the block of interest from the ORAM-tree stored on the cloud, while the latter is to obliviously write the retrieved block back to the ORAM-tree. Once the block is retrieved, it is assigned to a new random path (line 2), updated if needed (line 3), and then stored in the stash (line 5) to be pushed back later via the Evict protocol.

In our S^{3}ORAM framework, we select the eviction path deterministically, which follows the reverse lexicographical order proposed in [71]. Specifically, given a binary tree of height H, where edges in each level are indexed by either 0 (left) or 1 (right), the collection of edges of the eviction path at the EvictCtr-th eviction operation is calculated by the following formula.

$$v = \mathsf{DigitReverse}_2(\mathsf{EvictCtr} \mod 2^H), \tag{4.6}$$

where DigitReverse$_2$ denotes the order-reversal of the binary string representation of the decimal integer input.

[5] We assume implicitly that we choose an appropriate prime p such that every string c_j when interpreted as an element of $\mathbb{F}_p$ is less than p.

In the following, we present the main scheme in our $\mathsf{S^3ORAM}$framework called $\mathsf{S^3ORAM^O}$, which features the low client storage overhead. We describe another $\mathsf{S^3ORAM}$ scheme called $\mathsf{S^3ORAM^C}$, which offers efficient computation and low server storage overhead with the cost of client storage afterward.

4.4.2.2 $\mathsf{S^3ORAM^O}$*:* $\mathsf{S^3ORAM}$ *with Low Client Storage*

We introduce $\mathsf{S^3ORAM^O}$, an $\mathsf{S^3ORAM}$ scheme that does not require the client to maintain the stash component, thereby saving a factor of $\mathcal{O}(\lambda)$ client storage overhead. To achieve this, $\mathsf{S^3ORAM^O}$ follows the Triplet Eviction strategy in [54]. To enable $\mathcal{O}(1)$ client-bandwidth blowup, $\mathsf{S^3ORAM^O}$ harnesses homomorphic properties of SSS, which allows the client to "instruct" the servers to perform efficient retrieval and eviction operations in a secure manner without having to download and upload $\mathcal{O}(\log N)$ data blocks.

In the following, we describe in detail the retrieval and eviction protocol of $\mathsf{S^3ORAM^O}$ scheme.

To achieve $\mathcal{O}(1)$ client bandwidth blowup, the retrieval protocol in $\mathsf{S^3ORAM^O}$ scheme requires an efficient PIR protocol to privately retrieve the block of interest. We first describe the PIR protocol based on SSS as follows.

Our objective is to privately retrieve a block of interest residing in the queried path on the $\mathsf{S^3ORAM^O}$ tree. Recall that in the single-server HE-based ORAM schemes (*e.g.*, [10, 54]), the PIR query is encrypted with additive/fully HE. In $\mathsf{S^3ORAM^O}$, the tree is SSS-shared among ℓ servers, which features highly efficient additive and multiplicative homomorphic properties. We observe that the multi-server PIR scheme in [17, 73] relies on SSS to create PIR queries and, therefore, it can serve as a suitable private retrieval tool to be used for $\mathsf{S^3ORAM^O}$ scheme. We describe SSS-based PIR scheme in Figure 4.10, and further outline it as follows:

```
$\mathsf{PIR.CreateQuery}(j)$:
1: Let $\mathbf{e} := (e_1, \ldots, e_n)$, where $e_j \leftarrow 1$, $e_i \leftarrow 0$ for $1 \leq i \neq j \leq n$
2: for $i = 1, \ldots, n$ do
3:     $(\llbracket e_i \rrbracket_1^{(t)}, \ldots, \llbracket e_i \rrbracket_\ell^{(t)}) \leftarrow \mathsf{SSS.Create}(e_i, t)$
4: $\llbracket \mathbf{e} \rrbracket_i^{(t)} := (\llbracket e_1 \rrbracket_i^{(t)}, \ldots, \llbracket e_n \rrbracket_i^{(t)})$, for $1 \leq i \leq \ell$
5: return $(\llbracket \mathbf{e} \rrbracket_1^{(t)}, \ldots, \llbracket \mathbf{e} \rrbracket_\ell^{(t)})$

$\mathsf{PIR.Retrieve}(\llbracket \mathbf{e} \rrbracket_i^{(t)}, \llbracket \mathsf{DB} \rrbracket_i^{(t)})$:
6: $\llbracket b \rrbracket_i^{(2t)} \leftarrow \llbracket \mathbf{e} \rrbracket_i^{(t)} \cdot \llbracket \mathsf{DB} \rrbracket_i^{(t)}$
7: return $\llbracket b \rrbracket_i^{(2t)}$

$\mathsf{PIR.Reconstruct}(\llbracket b \rrbracket_1^{(2t)}, \ldots, \llbracket b \rrbracket_\ell^{(2t)})$:
8: $b \leftarrow \mathsf{SSS.Recover}(\llbracket b \rrbracket_1^{(2t)}, \ldots, \llbracket b \rrbracket_\ell^{(2t)}, 2t)$
9: return $b$
```

Figure 4.10: SSS-based PIR scheme.

```
$\mathsf{S^3ORAM^O.read(id)}$:
Client:
1: $(s, j) \leftarrow \mathsf{pm[id]}$
2: $(\llbracket \mathbf{e} \rrbracket_1^{(t)}, \ldots, \llbracket \mathbf{e} \rrbracket_\ell^{(t)}) \leftarrow \mathsf{PIR.CreateQuery}(j)$
3: Send $(s, \llbracket \mathbf{e} \rrbracket_i^{(t)})$ to server $S_i$, for $1 \leq i \leq \ell$
Server: each $S_i \in \{S_1, \ldots, S_\ell\}$ receiving $(s, \llbracket \mathbf{e} \rrbracket_i^{(t)})$ do
4: $\mathcal{I} \leftarrow \mathcal{P}(s)$
5: for $j = 1, \ldots, m$ do
6:     Let $\llbracket \mathbf{c}_j \rrbracket_i^{(t)}$ contain $j$-th chunk of $Z$ slots in $\llbracket \mathbf{T}[i'] \rrbracket_i^{(t)}$, $\forall i' \in \mathcal{I}$
7:     $\llbracket c_j \rrbracket_i^{(2t)} \leftarrow \mathsf{PIR.Retrieve}(\llbracket \mathbf{e} \rrbracket_i^{(t)}, \llbracket \mathbf{c}_j \rrbracket_i^{(t)})$
8: Send $(\llbracket c_1 \rrbracket_i^{(2t)}, \ldots, \llbracket c_m \rrbracket_i^{(2t)})$ to client
Client: On receive $(\{\llbracket c_1 \rrbracket_i^{(2t)}\}_{i=1}^{\ell}, \ldots, \{\llbracket c_m \rrbracket_i^{(2t)}\}_{i=1}^{\ell})$
9: $c_j \leftarrow \mathsf{PIR.Reconstruct}(\llbracket c_j \rrbracket_1^{(2t)}, \ldots, \llbracket c_j \rrbracket_\ell^{(2t)})$ for $1 \leq j \leq m$
10: $b \leftarrow (c_1, \ldots, c_m)$
11: return $b$
```

Figure 4.11: $\mathsf{S^3ORAM^O}$ retrieval subroutine.

Assume that each server S_i stores a share of the database DB containing n blocks denoted as $\llbracket \mathsf{DB} \rrbracket_i$, which can be interpreted as a vector with each i-th component being the share of the i-th item in DB. Let j be the index of the block in DB to be privately retrieved. The client executes the $\mathsf{PIR.CreateQuery}$ algorithm, which creates an n-dimensional unit vector with all zero coordinates except the j-th coordinate being set to 1 (line 1) and then, secret-shares it with SSS (lines 2–3). The client then distributes these shares to the corresponding servers, each answering with the result of the dot product between the received share vector and its share of DB by executing the $\mathsf{PIR.Retrieve}$

```
$\mathsf{S^3ORAM^O.Evict}()$:
$(c_1, \ldots, c_m) \leftarrow b$, where $b$ is the block that has just been retrieved
$([\![c_j]\!]_1, \ldots, [\![c_j]\!]_\ell) \leftarrow \mathsf{SSS.Create}(c_j, t)$ for $1 \leq j \leq m$
Write $([\![c_1]\!]_i, \ldots, [\![c_m]\!]_i)$ to slot $[\![\mathbf{T}[1, n_r + 1]]\!]_i$ in server $S_i$ for $1 \leq i \leq \ell$
$n_r \leftarrow n_r + 1 \mod A$ # $n_r$ is initialized with 0
if $n_r = 0$ then
    $v \leftarrow \mathsf{DigitReverse}_2(\mathsf{EvictCtr} \mod 2^H)$
    Execute $\mathsf{S^3ORAM^O.EvictAlongPath}(v)$ protocol
    $\mathsf{EvictCtr} \leftarrow \mathsf{EvictCtr} + 1 \mod 2^H$ # EvictCtr is initialized with 0
```

Figure 4.12: $\mathsf{S^3ORAM^O}$ eviction subroutine.

algorithm (line 6). Finally, the client executes the PIR.Reconstruct algorithm, which invokes the SSS.Recover algorithm over ℓ answers to recover the desired block (line 8). Since DB in this context is SSS-secret shared instead of plaintext as in [17, 73], our PIR.Reconstruct algorithm requires *at least* $2t + 1$ shares (instead of $t + 1$) to recover the item correctly.

We present the retrieval protocol in $\mathsf{S^3ORAM^O}$ in Figure 4.11, which employs three algorithms of the above SSS-based PIR scheme. Given the block to be read, the client first determines its location in the $\mathsf{S^3ORAM^O}$ tree via the position map pm (line 1) and then, privately retrieves it using the SSS-based PIR protocol. In this case, the server interprets all slots in the retrieval path as the database input DB in the PIR.Retrieve algorithm. Hence, the size of DB and the length of the query vector is $n = Z \cdot (H + 1)$. Since there are m separate chunks in each slot, the servers execute the PIR.Retrieve algorithm m times with the same PIR query but over different DB_j, where each DB_j contains the j-th chunk of all slots in the retrieval path (lines 5–7). Finally, the client obtains the desired block by recovering all chunks upon receiving their corresponding shares using the PIR.Reconstruct algorithm (line 9).

To eliminate the need of maintaining the stash component at the client-side, $\mathsf{S^3ORAM^O}$ follows the Triplet Eviction strategy proposed in [54]. The $\mathsf{S^3ORAM^O.Evict}$ algorithm in Figure 4.12 presents the eviction procedure in $\mathsf{S^3ORAM^O}$ scheme. Specifically, after the block is privately retrieved via the $\mathsf{S^3ORAM^O.read}$ protocol, the client creates new SSS-shares for it (lines 1–2), and then writes the share to an empty slot in the root bucket of the corresponding server (lines 3). After $A \leq Z$ successive retrievals, the client selects a deterministic eviction path following the reverse lexicographical order (line 5), and executes the $\mathsf{S^3ORAM^O.EvictAlongPath}$ protocol, to obliviously percolate the blocks from upper levels (*e.g.*, root bucket) to deeper levels (*e.g.*, leaf buckets).

According to the Triplet Eviction policy, for each level in the eviction path, all blocks from the source bucket ($\mathbf{T}[i]$) will be obliviously moved to all its children (*i.e.*, $\mathbf{T}[2i]$, $\mathbf{T}[2i+1]$). We follow the same terminology used in [54] to denote the buckets involved in each Triplet Eviction operation: If the child of the source bucket resides in the eviction path, it is called the *destination* bucket while the other child is called the *sibling* bucket (see Figure 4.13 for clarification). In our $\mathsf{S^3ORAM^O}$ scheme, the move is performed by computing the matrix product, in which the client creates permutation matrices and requests the servers to jointly perform the matrix product between such matrices and vectors containing data along the eviction path in the $\mathsf{S^3ORAM^O}$ tree. We present the algorithmic description of this strategy in the $\mathsf{S^3ORAM^O.EvictAlongPath}$ protocol in Figure 4.14 with details as follows.

Let $[\![\mathbf{u}]\!]$ be a $2Z$-dimensional share vector formed by concatenating all data in the source bucket and the destination bucket. The client creates a permutation matrix $\mathbf{I} \in \{0,1\}^{2Z \times Z}$ (line 2) such that the matrix product between $[\![\mathbf{u}]\!]$ and $\mathbf{I}$ will result in a Z-dimensional vector $[\![\mathbf{v}]\!]$, in which data at position i in $[\![\mathbf{u}]\!]$ is moved to position j in $[\![\mathbf{v}]\!]$. That is, $\mathbf{I}$ is a matrix, where $\mathbf{I}[i,j] \leftarrow 1$ if the block at position i in $[\![\mathbf{u}]\!]$ is expected to move to position j in $[\![\mathbf{v}]\!]$ (line 7). As a result, $\mathbf{I}[i+Z,i] \leftarrow 1$ if the block currently at position i in $[\![\mathbf{v}]\!]$ remains (line 12). To hide the location information of real blocks after permutation, the client "encrypts" every single element of $\mathbf{I}$ with SSS resulting in a share matrix $[\![\mathbf{I}]\!] \in \mathbb{F}_p^{2Z \times Z}$ (line 13). Note that the matrix product between these two shares

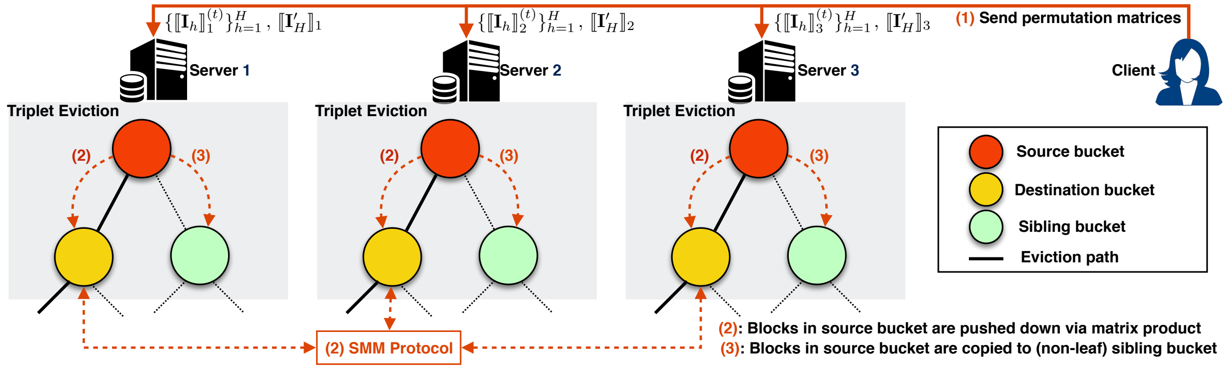

$\{[\![\mathbf{I}_h]\!]_1^{(t)}\}_{h=1}^H$, $[\![\mathbf{I}'_H]\!]_1$
$\{[\![\mathbf{I}_h]\!]_2^{(t)}\}_{h=1}^H$, $[\![\mathbf{I}'_H]\!]_2$
$\{[\![\mathbf{I}_h]\!]_3^{(t)}\}_{h=1}^H$, $[\![\mathbf{I}'_H]\!]_3$
(1) Send permutation matrices
Server 1
Server 2
Server 3
Client
Triplet Eviction
Triplet Eviction
Triplet Eviction
(2)
(3)
(2)
(3)
(2)
(3)
Source bucket
Destination bucket
Sibling bucket
Eviction path
(2) SMM Protocol
(2): Blocks in source bucket are pushed down via matrix product
(3): Blocks in source bucket are copied to (non-leaf) sibling bucket

Figure 4.13: The triplet eviction using SSS and SMM protocol.

results in a share vector with each element being represented by a degree-$2t$ polynomial. To maintain the consistency and privacy of the $\mathsf{S^3ORAM^O}$ tree structure, servers will jointly perform the SMM protocol to reduce the degree of the polynomial of each component in $[\![\mathbf{v}]\!]$ from $2t$ to t (line 22 and line 26).

We can apply the same trick as above to obliviously move real blocks in source buckets to their sibling buckets. However, since the non-leaf sibling buckets are guaranteed to be empty due to previous evictions passing on them (see Lemma 4), this process can be further optimized as discussed in [54] as follows. For each non-leaf sibling bucket in the eviction path, the client simply requests servers to copy all the data in the source bucket to the sibling bucket (line 19) and then, updates locally the path location of blocks in the position map (pm) accordingly (line 9–10). For the leaf sibling bucket, since it is not guaranteed to be empty at any time, we use the matrix permutation to move blocks from the source bucket to it as described above. This optimization can halve the client-server and server-server bandwidth cost as well as the server computation. Generally, we can see that our eviction approach requires only one client-server communication and guarantees that all data after eviction are consistently "encrypted" by degree-t polynomials. Figure 4.13 visualizes this new SSS-based Triplet Eviction strategy.

```
$\mathsf{S^3ORAM^O.EvictAlongPath}(v)$:
Let $(u_1, \ldots, u_H)$ be the (ordered) indexes of source buckets along the eviction path $v$
Client:
1: for $h = 1, \ldots, H$ do
2:     Let $\mathbf{I}_h$ be a $2Z \times Z$ matrix, set $\mathbf{I}_h[*,*] \leftarrow 0$
3:     for each real block with id in the source bucket $\mathbf{T}[u_h]$ do
4:         if id can legally reside in the destination bucket of $\mathbf{T}[u_h]$ then
5:             $(\mathsf{pid}, \mathsf{pIdx}) \leftarrow \mathsf{pm}[\mathsf{id}]$
6:             Let $y$ $(1 \le y \le Z)$ be the index of an empty slot in the destination bucket of $\mathbf{T}[u_h]$
7:             $\mathbf{I}_h[x][Z+y] \leftarrow 1$, where $x \leftarrow \mathsf{pIdx} \mod Z$
8:             $\mathsf{pm}[\mathsf{id}'].\mathsf{pIdx} \leftarrow Z \cdot h + y$ # Update the new location of the block in the path
9:         else# id can legally reside in the sibling bucket of $\mathbf{T}[u_h]$
10:            $\mathsf{pm}[\mathsf{id}].\mathsf{pIdx} \leftarrow \mathsf{pm}[\mathsf{id}].\mathsf{pIdx} + Z$
11:    for each real block with $\mathsf{id}'$ in the destination bucket of $\mathbf{T}[u_h]$ do
12:        $\mathbf{I}_h[x+Z][x] \leftarrow 1$, where $x \leftarrow \mathsf{pm}[\mathsf{id}'].\mathsf{pIdx} \mod Z$
13:    $[\![\mathbf{I}_h[x,y]]\!]_1^{(t)}, \ldots, [\![\mathbf{I}_h[x,y]]\!]_\ell^{(t)} \leftarrow \mathsf{SSS.Create}(\mathbf{I}_h[x,y], t)$ for $1 \le x \le 2Z$, $1 \le y \le Z$
14: Repeat lines 2–13 (excluded lines 9-10) to create $[\![\mathbf{I'}_H]\!]$, the share of permutation matrix for source to sibling
    bucket at the leaf level $(h = H)$
15: Send $([\![\mathbf{I}'_H]\!]_i^{(t)}, [\![\mathbf{I}_1]\!]_i^{(t)}, \ldots, [\![\mathbf{I}_H]\!]_i^{(t)})$ to $S_i$, for $1 \le i \le \ell$
Server: each $S_i \in \{S_1, \ldots, S_\ell\}$ receiving $([\![\mathbf{I}'_H]\!]_i^{(t)}, [\![\mathbf{I}_1]\!]_i^{(t)}, \ldots, [\![\mathbf{I}_H]\!]_i^{(t)})$ do
18: for $h = 1, \ldots, H$ do
19:     Copy all data from source bucket $[\![\mathbf{T}[u_h]]\!]_i^{(t)}$ to its non-leaf sibling bucket
20:     for $j = 1, \ldots, m$ do
21:         Let $[\![\mathbf{c}_{h,j}]\!]_i^{(t)}$ be a vector containing $j$-th chunks of $[\![\mathbf{T}[u_h]]\!]_i^{(t)}$ and its destination bucket
22:         $[\![\hat{\mathbf{c}}_{h,j}]\!]_i^{(t)} \leftarrow [\![\mathbf{c}_{h,j}]\!]_i^{(t)} \star [\![\mathbf{I}_h]\!]_i^{(t)}$
23:         Update $j$-th chunks of the destination bucket of $[\![\mathbf{T}[u_h]]\!]_i^{(t)}$ with $[\![\hat{\mathbf{c}}_{h,j}]\!]_i^{(t)}$
24: for $j = 1, \ldots, m$ do
25:     Let $[\![\mathbf{c}'_{H,j}]\!]_i^{(t)}$ be a vector containing $j$-th chunks of source bucket $[\![\mathbf{T}[u_H]]\!]_i^{(t)}$ and its (leaf) sibling bucket
26:     $[\![\hat{\mathbf{c}}'_{H,j}]\!]_i^{(t)} \leftarrow [\![\mathbf{c}'_{H,j}]\!]_i^{(t)} \star [\![\mathbf{I}'_H]\!]_i^{(t)}$
27:     Update $j$-th chunks of the sibling bucket of $[\![\mathbf{T}[u_H]]\!]_i^{(t)}$ with $[\![\hat{\mathbf{c}}'_{H,j}]\!]_i^{(t)}$
```

Figure 4.14: $\mathsf{S^3ORAM^O}$ triplet eviction with SSS scheme and SMM protocol.

We analyze the cost of $\mathsf{S^3ORAM^O}$ pertaining to the block size ($|b|$), number of blocks (N), and statistical security parameter (λ). We consider other system parameters (*e.g.*, prime field $\mathbb{F}_p$, number of servers ℓ) to be fixed.

In the $\mathsf{S^3ORAM^O}$ retrieval phase, each PIR query being sent to ℓ servers is of size $(Z \cdot (H+1) \cdot \lceil \log_2 p \rceil)$ bits. The client exchanges one block of size $|b|$ with each server. The Triplet Eviction is performed after every A subsequent retrievals. In this operation, the client sends $H+1$ permutation matrices to ℓ servers. Each matrix is of size $2Z^2 \cdot \lceil \log_2 p \rceil$ bits. The servers exchange the shares of $H+1$ buckets with each other, each being of size $Z \cdot |b|$ bits. Therefore, given

$H = \mathcal{O}(\log N), Z = A = \mathcal{O}(\lambda)$ and ℓ, p are constants, the *amortized* client-server communication complexity is $\mathcal{O}(|b| + \lambda \cdot \log N)$. The *amortized* server-server communication overhead is $\mathcal{O}(|b| \cdot \log N)$.

The client bandwidth blowup is defined as the *ratio* between the cost of client-server communication by using ORAM to access the block *vs.* the base case where the block is insecurely accessed without ORAM. Our analyzed communication complexity of $\mathsf{S^3ORAM^O}$ above indicates that the size of the PIR query and the permutation matrices is *independent* of the block size parameter $|b|$. Therefore, the $\mathcal{O}(1)$ client bandwidth blowup can be achieved in $\mathsf{S^3ORAM^O}$ by selecting a suitable value of $|b|$. That is, by selecting $|b| = \Omega(\lambda \cdot \log N)$[6], $\mathsf{S^3ORAM}$ achieves $\mathcal{O}(1)$ client bandwidth blowup.

In the retrieval phase, the servers compute the dot product between the $Z \cdot (H+1)$-dimensional PIR query vector and the block vector containing $Z \cdot (H+1)$ blocks of size $|b|$. In the Triplet Eviction phase, the servers compute $H+1$ times the matrix product between a vector containing $2Z$ blocks of size $|b|$ and a permutation matrix of size $2Z \times Z$. The matrix product incurs re-sharing and computing the degree reduction in the SMM protocol on $Z \cdot (H+1)$ blocks each being of size $|b|$. In total, the *amortized* server computation complexity is $\mathcal{O}(|b| \cdot \lambda \cdot \log N)$.

The client executes the $\mathsf{SSS.Create}$ algorithm $Z \cdot (H+1)$ times and $2Z^2 \cdot (H+1)$ times to create the PIR query and $H+1$ permutation matrices, respectively. The client executes the $\mathsf{SSS.Recover}$ and $\mathsf{SSS.Create}$ algorithms to reconstruct and re-share a block of size $|b|$, respectively. Thus, the *amortized* client computation complexity is $\mathcal{O}(|b| + \log N)$.

$\mathsf{S^3ORAM^O}$ layout is a full binary tree of height H, which has a total of $Z \cdot (2^{H+1} - 1)$ slots and can store up to $N \leq A \cdot 2^{H-1}$ real blocks. Given $A = Z = \Theta(\lambda)$ for statistical security (see

[6] In the ORAM community, $\lambda = \mathcal{O}(\log N)$ is commonly used. With this assumption, the block size in $\mathsf{S^3ORAM^O}$ is $\Omega(\log^2 N)$

Lemma 4), the server storage blowup cost is $\mathcal{O}(1)$. Notice that the share of the value has the same size as the value (*i.e.*, no ciphertext expansion as in Onion-ORAM), the server storage of $\mathsf{S}^3\mathsf{ORAM}$ is constant and does not increase after a sequence of access operations.

Similar to Onion-ORAM, $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}$ does not require the stash component since the retrieved block is immediately written back to the root bucket. Hence, the client block storage in $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}$ is $\mathcal{O}(1)$. The client locally stores the position map whose cost is $\mathcal{O}(N \cdot (\log N + \log\log N))$.

For theoretical interest, $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}$ can achieve (in total) $\mathcal{O}(1)$ client storage by storing the position map in smaller ORAMs using the recursion technique in [168] and the bucket metadata structure in [54]. Specifically, for each bucket in the $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}$ tree, we create a metadata that stores the current index (pIdx) and the assigned path (pid) of blocks residing in it. For each $\mathsf{S}^3\mathsf{ORAM}$ access, the metadata of buckets along the retrieval/eviction path will be read first to get the path and the location of blocks of interest. This information will be used to create the PIR query and permutation matrices. Next, we construct a series of $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}$ structures $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}_0, \ldots, \mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}_{\log_r N}$, where $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}_0$ stores database blocks and each block j in $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}_{i+1}$ stores the path information (pid) of the blocks $(j-1)r, \ldots, jr$ in $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}_i$ and $r \geq 2$ is the compression ratio. We refer the reader to [54, 168] for the detailed descriptions.

For simplicity, we assume that $r = 2$ and let $H = \log N$ be the height of $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}_0$. In $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}_i$ $(i \geq 1)$, the size of meta-data is $\lambda(H - i)$, the block size is $2(H - i + 1)$, and the path length is $H - i$. There are $\log N$ recursive levels so that the total bandwidth overhead for each recursive $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}$ retrieval is $\lambda \sum_{i=0}^{H-1} i^2 + \sum_{i=1}^{H} 2(H - i + 1) = \mathcal{O}(\lambda \log^3 N)$. Due to amortization, the asymptotic cost of eviction is similar to the retrieval as analyzed above. Therefore, to achieve $\mathcal{O}(1)$ client bandwidth blowup, the block size of $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}_0$ needs to be $\Omega(\lambda \cdot \log^3 N)$. So, using the recursion technique to get rid of the client position map increases the regular block size a factor of $\mathcal{O}(\log^2 N)$ and $\mathcal{O}(\log N)$ communication *rounds*.

The regular block size in recursive $\mathsf{S^3ORAM^O}$ is a factor of $\log^2 N$ times larger than other (recursive) tree-based ORAM schemes featuring $\mathcal{O}(\log N)$ bandwidth (*e.g.*, Path ORAM, Ring-ORAM, Tree-ORAM) and (at least) $\log N$ times smaller than (recursive) tree-based ORAM with $\mathcal{O}(1)$ bandwidth (*e.g.*, Onion-ORAM [54], Bucket-ORAM [61], OVS [10]) due to the following reasons. As analyzed above, to keep the original asymptotic communication overhead intact when applying the recursion technique, the regular block size must be large enough to absorb the cost of transmitting the blocks and the meta-data components from $\mathcal{O}(\log N)$ small (recursive) ORAM structures. In ORAM schemes with $\mathcal{O}(\log N)$ bandwidth, since the size of small blocks in their recursive structures is $\mathcal{O}(\log N)$, the regular block size is $\Omega(\log^2 N)$ to absorb the cost of downloading $\mathcal{O}(\log^2 N)$ small blocks (there is no meta-data component in these schemes). On the other hand, the regular block size of $\mathcal{O}(1)$-bandwidth ORAM schemes does not increase when applying the recursion, since it is already larger than the total amount needed to absorb the cost of downloading the blocks and the meta-data of small ORAM structures (*e.g.*, $\Omega(\log^5 N)$-$\Omega(\log^6 N)$ block size *vs.* $\Omega(\log^3 N)$ needed).

Given that the recursion technique significantly increases the regular block size, it is recommended to maintain the position map locally assuming that its size is small enough. This choice allows the implementor to gain the full performance advantages that $\mathsf{S^3ORAM}$ offers in practice.

4.4.2.3 $\mathsf{S^3ORAM^C}$*: $\mathsf{S^3ORAM}$ with Low Server Computation*

In this section, we present $\mathsf{S^3ORAM^C}$, a $\mathsf{S^3ORAM}$ scheme that achieves lower computational complexity than $\mathsf{S^3ORAM^O}$ due to its smaller bucket size parameter Z (*e.g.*, $\mathcal{O}(1)$ *vs.* $\mathcal{O}(\lambda)$). The price to pay for such achievement is that it requires maintaining the stash at the client-side to temporarily store blocks that cannot be pushed back to the tree due to the small bucket size. The intuition of $\mathsf{S^3ORAM^C}$ is to implement the access protocol of Circuit-ORAM proposed by Wang *et al.* [179] using the homomorphic properties of SSS as follows.

$\mathsf{S^3ORAM^C}$ has the same retrieval procedure like $\mathsf{S^3ORAM^O}$ scheme, where we leverage SSS-based PIR Scheme to privately retrieve the block in the retrieval path of the $\mathsf{S^3ORAM^C}$ tree (Figure 4.15).

```
$\mathsf{S^3ORAM^C.read(id)}$
  $b \leftarrow \mathsf{S^3ORAM^O.read(id)}$
  return $b$
```

Figure 4.15: $\mathsf{S^3ORAM^C}$ retrieval subroutine.

$\mathsf{S^3ORAM^C}$ implements the eviction principle in Circuit-ORAM scheme with additive and multiplicative homomorphic properties of SSS. Similar to $\mathsf{S^3ORAM^O}$ scheme, $\mathsf{S^3ORAM^C}$ selects a deterministic eviction path following the reverse lexicographical order (Equation 4.6) proposed in [71] (Figure 4.16), which was proven to achieve the negligible overflow probability with a lower bucket size parameter compared with the random path (*e.g.*, 2 *vs.* 3). We note that the detail descriptions of $\mathsf{PrepareDeepest}(v)$ and $\mathsf{PrepareTarget}(v)$ subroutines in Figure 4.16 can be found in [179].

Intuitively, the client first scans the position map to prepare the target array that indicates which blocks to be pushed down to which levels in the eviction path. Afterward, the client goes through each level of the eviction path, picks the desired block and drops it to the target level.

```
$\mathsf{S^3ORAM^C.Evict()}$:
  1: $v \leftarrow \mathsf{DigitReverse}_2(\mathsf{EvictCtr} \mod 2^H)$
  2: Execute $\mathsf{S^3ORAM^C.EvictAlongPath}(v)$ protocol
  3: $\mathsf{EvictCtr} \leftarrow \mathsf{EvictCtr} + 1 \mod 2^H$
  4: Repeat lines 1-3
```

Figure 4.16: $\mathsf{S^3ORAM^C}$ eviction subroutine.

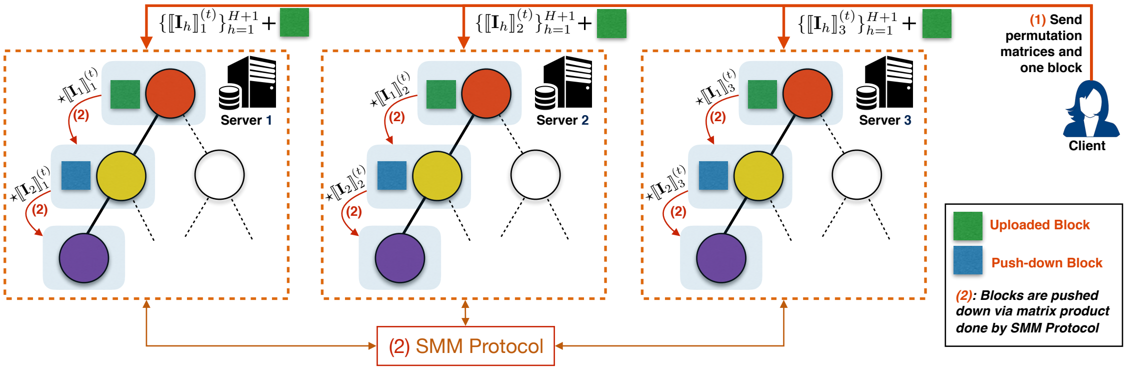

$\{[\![\mathbf{I}_h]\!]_1^{(t)}\}_{h=1}^{H+1}+$ 🟩
$\{[\![\mathbf{I}_h]\!]_2^{(t)}\}_{h=1}^{H+1}+$ 🟩
$\{[\![\mathbf{I}_h]\!]_3^{(t)}\}_{h=1}^{H+1}+$ 🟩
(1) Send permutation matrices and one block
Client
Server 1
Server 2
Server 3
$\star[\![\mathbf{I}_1]\!]_1^{(t)}$
(2)
$\star[\![\mathbf{I}_2]\!]_1^{(t)}$
(2)
$\star[\![\mathbf{I}_1]\!]_2^{(t)}$
(2)
$\star[\![\mathbf{I}_2]\!]_2^{(t)}$
(2)
$\star[\![\mathbf{I}_1]\!]_3^{(t)}$
(2)
$\star[\![\mathbf{I}_2]\!]_3^{(t)}$
(2)
🟩 Uploaded Block
🟦 Push-down Block
(2): Blocks are pushed down via matrix product done by SMM Protocol
(2) SMM Protocol

Figure 4.17: S^{3}ORAM$^{\mathsf{O}}$ eviction based on [179] using SSS and SMM protocol.

Notice that at any time, the client holds and drops at most one block. This policy is guaranteed by computing a target array that indicates whether to pick/drop the block in each level by scanning the position map. We refer the reader to [179] for the detailed description and explanation.

Figure 4.17 visualizes the high-level idea of the eviction in S^{3}ORAM$^{\mathsf{C}}$, which implements the push-down strategy in [179] using SSS and SMM protocol. Figure 4.18 describes the detailed algorithm with the high-level idea as follows. For each level (h) in the eviction path, the client creates a permutation matrix ($\mathbf{I}_h$) of size $(Z+1)\times(Z+1)$. We use the last column of the matrix ($\mathbf{I}_h[*][Z+1]$) to indicate the block to be picked, while the other columns $\mathbf{I}_h[*][j]$ ($1\leq j\leq Z$) is to indicate the block to be moved to or hold at j-th slot of the h-leveled bucket. The data vector $[\![\mathbf{c}_h]\!]$, which will be computed the matrix product with $\mathbf{I}_h$, is of size $Z+1$ containing the holding block ($[\![\mathbf{c}_h[1]]\!]$) and the data from Z slots of the h-leveled bucket ($[\![\mathbf{v}_h[2]]\!]\ldots,[\![\mathbf{v}_h[Z+1]]\!]$). So, the client sets $\mathbf{I}_h[x][Z+1]\leftarrow 1$ to pick the block at slot x (line 15), and $\mathbf{I}_h[1][x]\leftarrow 1$ to drop the holding block to the x-th slot of the h-leveled bucket (line 10). If the currently holding block is moved to the next level (*i.e.*, no pickup/drop-off at this level), the client sets $\mathbf{I}_h[1][Z+1]\leftarrow 1$ (line 13). Similar to S^{3}ORAM$^{\mathsf{O}}$ scheme, the client sets $\mathbf{I}_h[x+1][x]\leftarrow 1$ to keep blocks indexed x in the h-leveled bucket in position (lines 17-20). Finally, the client creates the SSS-shares for such permutation matrices

```
$\mathsf{S^3ORAM^C.EvictAlongPath}(v)$:
Let $(u_1,\dots,u_{H+1})$ be the (ordered) bucket indexes along the eviction path $v$ from the root to the leaf level
Client:
 1: (deepest, deepestIdx) $\leftarrow$ PrepareDeepest$(v)$; target $\leftarrow$ PrepareTarget$(v)$
 2: hold $\leftarrow \perp$, dest $\leftarrow \perp$, $(c_1,\dots,c_m) \leftarrow 0^{|b|}$
 3: if target$[0] \neq \perp$ then # target[0] and deepestIdx[0] denote the stash component
 4:     hold $\leftarrow$ deepestIdx$[0]$, dest $\leftarrow$ target$[0]$
 5:     $(c_1,\dots,c_m) \leftarrow \mathbf{S}$[hold], $\mathbf{S}$[hold] $\leftarrow \{\}$
 6: for $h = 1,\dots,H+1$ do
 7:     Let $\mathbf{I}_h$ be a $(Z+1)\times(Z+1)$ matrix, set $\mathbf{I}_h[*,*] \leftarrow 0$.
 8:     if hold $\neq \perp$ then
 9:         if $i =$ dest then # Drop the holding block to this level
10:             $\mathbf{I}_h[1][x] \leftarrow 1$ where $x$ is the index of an empty slot in the bucket $\mathbf{T}[u_h]$
11:             hold $\leftarrow \perp$, dest $\leftarrow \perp$
12:         else# Move the holding block to the next level
13:             $\mathbf{I}_h[1][Z+1] \leftarrow 1$
14:     if target$[i] \neq \perp$ then # Pick a block at this level
15:         $\mathbf{I}_h[x][Z+1] \leftarrow 1$ where $x \leftarrow$ deepestIdx[h]
16:         hold $\leftarrow x$, dest $\leftarrow$ target$[i]$
17:     for each real block id in $\mathbf{T}[u_h]$ do # Hold the position of other real blocks at this level
18:         $x \leftarrow$ pm[id].pIdx  mod $Z$
19:         if  $x \neq$ deepestIdx$[h]$ then
20:             $\mathbf{I}_h[x+1][x] \leftarrow 1$
21: $[\![\mathbf{I}_h[*,*]]\!]_1^{(t)},\dots,[\![\mathbf{I}_h[*,*]]\!]_\ell^{(t)} \leftarrow$ SSS.Create$(\mathbf{I}_h[*,*],t)$ for $1 \le h \le H+1$
22: $([\![c_j]\!]_1^{(t)},\dots,[\![c_j]\!]_\ell^{(t)}) \leftarrow$ SSS.Create$(c_j,t)$ for $1 \le j \le m$
23: Send $\left(\langle [\![c_1]\!]_i^{(t)},\dots,[\![c_m]\!]_i^{(t)}\rangle, \langle [\![\mathbf{I}_1]\!]_i^{(t)},\dots,[\![\mathbf{I}_{H+1}]\!]_i^{(t)}\rangle\right)$ to $S_i$, for $1 \le i \le \ell$
Server: each $S \in \{S_1,\dots,S_\ell\}$ receiving $(\langle [\![c_1]\!],\dots,[\![c_m]\!]\rangle, \langle [\![\mathbf{I}_1]\!],\dots,[\![\mathbf{I}_{H+1}]\!]\rangle)$ do
18: $[\![x]\!]_j \leftarrow [\![c]\!]_j$ for $1 \le i \le m$
19: for $h = 1,\dots,H+1$ do
20:     for $j = 1,\dots,m$ do
21:         Let $[\![\mathbf{c}_{h,j}]\!]$ be a $Z$-dimensional vector containing $j$-th chunks of bucket $[\![\mathbf{T}[u_h]]\!]$
22:         $[\![\hat{\mathbf{c}}_{h,j}]\!] := ([\![x_j]\!],[\![\mathbf{c}_{h,j}]\!])$ # Concatenate $[\![x_j]\!]$ with $[\![\mathbf{c}_{h,j}]\!]$  resulting in a $(Z+1)$-dimensional vector
23:         $[\![\hat{\mathbf{c}}'_{h,j}]\!] \leftarrow [\![\hat{\mathbf{c}}_{h,j}]\!] \star [\![\mathbf{I}_h]\!]$
24:         $([\![\mathbf{c}'_{h,j}]\!],[\![x_j]\!]) := [\![\hat{\mathbf{c}}'_{h,j}]\!]$ # Assign the last component of vector $[\![\hat{\mathbf{c}}']\!]_{h,j}$ to $[\![x_j]\!]$
25:         Update $j$-th chunks of bucket $[\![\mathbf{T}[u_h]]\!]$ with $[\![\mathbf{c}'_{h,j}]\!]$
```

Figure 4.18: $\mathsf{S^3ORAM^C}$ eviction protocol based on [179].

(line 21) and for the block being picked-up in the stash (if any) (line 22), and distributes the shares to the corresponding servers (line 23). Similar to $\mathsf{S^3ORAM^O}$, for each level in the eviction path, the servers jointly perform the matrix product between the share of data vector and the share of permutation matrix via local addition and the secure multiplication protocol (line 23), and update the bucket with the newly computed vector (line 25).

Similar to $\mathsf{S^3ORAM^O}$, we analyze the cost of $\mathsf{S^3ORAM^C}$ regarding the block size ($|b|$), number of blocks (N), and statistical security parameter (λ), while other system parameters (*e.g.*, prime field $\mathbb{F}_p$, number of servers ℓ) are treated as constants. $\mathsf{S^3ORAM^O}$ has the same tree layout, an identical retrieval phase and a similar eviction procedure with the $\mathsf{S^3ORAM^O}$ scheme. $\mathsf{S^3ORAM^C}$ only differs from $\mathsf{S^3ORAM^O}$ in terms of the bucket size parameter (Z) and the eviction frequency, which happens after *every* retrieval instead of A as in $\mathsf{S^3ORAM^O}$. $\mathsf{S^3ORAM^C}$ also incurs at most three blocks (one for retrieval and two for eviction) to be transmitted in each ORAM access. Given $Z = \mathcal{O}(1)$ in $\mathsf{S^3ORAM^C}$, we summarize the asymptotic cost of $\mathsf{S^3ORAM^O}$ as follows.

The client-server communication complexity is $\mathcal{O}(|b|+\log N)$. The server-server communication overhead is $\mathcal{O}(|b|\cdot\log N)$. To achieve $\mathcal{O}(1)$ client-server bandwidth blowup, the minimal block size is $\Omega(\log N)$, which is a factor of λ times smaller than that of $\mathsf{S^3ORAM^O}$.

The server computation is $\mathcal{O}(|b|\cdot\log N)$. The client computation complexity is $\mathcal{O}(|b|+\log N)$.

$\mathsf{S^3ORAM^C}$ layout is a full binary tree of height H, which has a total of $Z\cdot(2^{H+1}-1)$ slots and can store up to $N \leq 2^H$ real blocks. Since $Z = \mathcal{O}(1)$, the server storage blowup in $\mathsf{S^3ORAM^C}$ is $\mathcal{O}(1)$ similar to $\mathsf{S^3ORAM^O}$ asymptotically, but its constant overhead factor is smaller (*i.e.*, 2 *vs.* 8). The client requires to maintain the stash, which costs $\mathcal{O}(\lambda)$ to achieve negligible overflow probability. The position map costs $\mathcal{O}(N(\log N + \log\log N))$. Therefore, the total client storage is $\mathcal{O}(\lambda + N(\log N + \log\log N))$. It is possible to achieve $\mathcal{O}(1)$ client storage by storing the stash (via SSS shares) on the servers and using the recursion technique to keep the position map in smaller ORAMs. However, this will significantly increase the computation and communication overhead for oblivious access to the stash and the position map, respectively.

4.4.3 Security Analysis

In this section, we analyze the security of two $\mathsf{S^3ORAM}$ schemes. First, $\mathsf{S^3ORAM^O}$ follows the Triplet Eviction strategy originally proposed in Onion-ORAM [54]. Therefore, it achieves the same failure probability with Onion-ORAM. We refer the reader to [54] for the detail of the proof.

Lemma 4 ($\mathsf{S^3ORAM^O}$ Bucket Overflow Probability). If $Z \geq A$ and $N \leq A \cdot 2^{H-1}$, the probability that a bucket overflows after a Triplet Eviction operation is bounded by $e^{-\frac{(2Z-A)^2}{6A}}$, where $Z = A = \Theta(\lambda)$.

Proof. We refer the reader to [54]. □

It is easy to see that Lemma 4 implies the following fact.

Corollary 3 (Non-Leaf Destination Bucket Load). All non-leaf destination buckets are always empty after the Triplet Eviction takes place, except with a negligible probability.

We present the main security of $\mathsf{S^3ORAM^O}$ as follows.

Theorem 3 ($\mathsf{S^3ORAM^O}$ Security). $\mathsf{S^3ORAM^O}$ is correct and information-theoretically (statistically) t-secure by Definition 7.

Proof. $\mathsf{S^3ORAM^O}$ is correct iff (i) the $\mathsf{S^3ORAM^O.read}(\cdot)$ protocol returns the correct value of the retrieved block and (ii) the $\mathsf{S^3ORAM^O.Evict}(\cdot)$ function is consistent.

For each data request x, let b be the block to be retrieved and j be the location of b in its path (*i.e.*, $j := \mathsf{pm}[\mathsf{id}].\mathsf{pIdx}$ where id is the identifier of b). The share of the PIR query for server S_i is of form: $[\![\mathbf{e}]\!]_i^{(t)} = ([\![e]\!]_1^{(t)}, \ldots, [\![e]\!]_n^{(t)})$, where $n = Z \cdot (H+1)$ and $e_i = 0$ for $1 \leq i \neq j \leq n, e_j = 1$. Let $[\![\mathbf{c}_u]\!] = ([\![c_{u1}]\!], \ldots, [\![c_{un}]\!])$ be the vector consisting of the share of u-th chunks taken from Z slots in every bucket residing in the retrieval path. For $1 \leq u \leq m$, the answer of each server S_i is of form:

$$\begin{aligned} [\![\mathbf{e}]\!]_i^{(t)} \cdot [\![\mathbf{c}_u]\!]_i^{(t)} &= \sum_{k=1}^{n} \left([\![e_k]\!]^{(t)} \cdot [\![c_{u,k}]\!]^{(t)} \right) \\ &= \sum_{k=1}^{n} [\![e_k \cdot c_{u,k}]\!]^{(2t)} && \text{by Equation 4.3} \\ &= [\![c_{u,j}]\!]^{(2t)} && \text{by Equation 4.1} \end{aligned}$$

By SSS scheme, at least $2t+1$ shares are required to recover the secret hidden by a random $2t$-degree polynomial. Our system model presented in §4.4.1 follows this and, therefore, the client always computes the correct value of chunk c_t by $c_t \leftarrow \mathsf{SSS.Recover}([\![c_t]\!]_1^{(2t)}, \ldots, [\![c_t]\!]_\ell^{(2t)}, 2t)$. Since all chunks of b are correctly computed, b is properly retrieved with the probability 1.

Corollary 3 shows that the root bucket is empty after the triplet eviction. The client writes the retrieved block to an empty slot in the root bucket sequentially (line 3, Figure 4.12). Since $Z \geq A$, the root always has enough empty slots to contain all the blocks to be written before the triplet eviction happens, thereby avoiding the overwritten and inconsistency issues. After A accesses, the client executes the triplet eviction algorithm (Figure 4.14) to move blocks from upper levels (*e.g.*, root) to deeper levels (*e.g.*, leaf). Corollary 3 also shows that non-leaf sibling buckets are empty due to previous triplet evictions and, therefore, they can contain all data moved from their source bucket without creating any inconsistency issue. Real blocks from source buckets are moved to destination buckets via matrix products. These computations are correct due to homomorphic properties of two-share addition and multiplication offered by SSS and the SMM protocol, respectively, which were proven correct in [68].

We now prove the security of $\mathsf{S^3ORAM^O}$ as follows.

Given a request sequence $\mathbf{x}$ of length q, where $x_j = (\mathsf{op}_j, \mathsf{id}_j, \mathsf{data}_j)$ as in Definition 7, let $\mathsf{S^3ORAM}_i^{\mathsf{O}}(\mathbf{x})$ be the $\mathsf{S^3ORAM^O}$ client's sequence of interactions with the server S_i including a sequence of *retrievals* (Figure 4.11), *write-to-root* (line 3, Figure 4.12) and *triplet eviction* operations (Figure 4.14). We have that the *write-to-root* operation is deterministic, which is performed right after the retrieval. In this operation, the previously retrieved block is written to a publicly known slot in the root bucket as shown above. The *triplet eviction* is also deterministic, which is performed after every A successive accesses regardless of any data being requested. Since all these operations (*i.e.*, *retrieval*, *write-to-root*, *triplet eviction*) are independent of each other, they can be considered as separate sequences observed by S_i in $\mathsf{S^3ORAM}_i^{\mathsf{O}}(\mathbf{x})$ as follows

$$\mathsf{S^3ORAM}_i^{\mathsf{O}}(\mathbf{x}) = \begin{cases} \overrightarrow{R}_i(\mathbf{x}) & = & (R_i^{(x_1)}, \ldots, R_i^{(x_q)}) \\ \overrightarrow{W}_i(\tilde{\mathbf{x}}) & = & (W_i^{(\tilde{x}_1)}, \ldots, W_i^{(\tilde{x}_q)}), \\ \overrightarrow{E}_i(\bar{\mathbf{x}}) & = & (E_i^{(\bar{x}_1)}, \ldots, E_i^{(\bar{x}_{q/A})}) \end{cases} \tag{4.7}$$

where $\overrightarrow{R}_i(\tilde{\mathbf{x}})$, $\overrightarrow{W}_i(\tilde{\mathbf{x}})$ and $\overrightarrow{E}(\bar{\mathbf{x}})$ denote the *retrieval*, *write-to-bucket* and *triplet eviction* sequences, given a data access sequence $\mathbf{x}$, respectively.

Assume that there is a coalition of up t servers $\{S_{i\in\mathcal{I}}\}$ sharing their own transcripts with each other. Let $\mathcal{I} \subseteq \{1, \ldots, \ell\}$ such that $|\mathcal{I}| \leq t$. The view of $\{S_{i\in\mathcal{I}}\}$ can be derived from Equation 4.7 as

$$\{\mathsf{S^3ORAM}_{i\in\mathcal{I}}^{\mathsf{O}}(\mathbf{x})\} = \begin{cases} \{\overrightarrow{R}_{i\in\mathcal{I}}(\mathbf{x})\} & = & (\{R_{i\in\mathcal{I}}^{(x_1)}\}, \ldots, \{R_{i\in\mathcal{I}}^{(x_q)}\}) \\ \{\overrightarrow{W}_{i\in\mathcal{I}}(\tilde{\mathbf{x}})\} & = & (\{W_{i\in\mathcal{I}}^{(\tilde{x}_1)}\}, \ldots, \{W_{i\in\mathcal{I}}^{(\tilde{x}_q)}\}), \\ \{\overrightarrow{E}_{i\in\mathcal{I}}(\bar{\mathbf{x}})\} & = & (\{E_{i\in\mathcal{I}}^{(1)}\}, \ldots, \{E_{i\in\mathcal{I}}^{(q/A)}\}) \end{cases}$$

We show that for any two access sequences $\mathbf{x}$ and $\mathbf{x}'$ of the same length (*i.e.*, $|\mathbf{x}| = |\mathbf{x}'|$), the pairs $\langle\{\overrightarrow{R}_{i\in\mathcal{I}}(\mathbf{x})\}, \{\overrightarrow{W}_{i\in\mathcal{I}}(\tilde{\mathbf{x}})\}, \{\overrightarrow{E}_{i\in\mathcal{I}}(\mathbf{x})\}\rangle$ and $\langle\{\overrightarrow{R}_{i\in\mathcal{I}}(\mathbf{x}')\}, \{\overrightarrow{W}_{i\in\mathcal{I}}(\tilde{\mathbf{x}}')\}, \{\overrightarrow{E}_{i\in\mathcal{I}}(\bar{\mathbf{x}}')\}\rangle$ are identically distributed.

For each access request $x_j \in \mathbf{x}$, $\{S_{i\in\mathcal{I}}\}$ observes a transcript $\{R_{i\in\mathcal{I}}^{(x_j)}\}$ consisting of a retrieval path $\mathcal{P}_{x_j}$ (access pattern), which is identical for all servers (line 4, Figure 4.11) and all data generated by the SSS-based PIR scheme (lines 5-7).

The access pattern of $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$ is identical to all other secure tree-based ORAM schemes. Specifically, each block in $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$ is assigned to a leaf bucket selected randomly and independently from each other. Once a block is accessed, it is assigned to a new bucket leaf selected randomly and independently. Such random assignment along with the selected bucket size parameter (Z) may result in the bucket(s) in the $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$ tree being overflowed with a negligible probability thereby, impacts the security (see Lemma 4). Therefore, access patterns generated by any data request sequences of the same length are *statistically* indistinguishable. We next analyze the probability distribution of data observed at the server side in each $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$ retrieval as follows. For each retrieval, the client sends to the servers PIR queries generated by $\mathsf{PIR.CreateQuery}$ algorithm. Such queries are SSS shares and, therefore, is t-private. The inner product is also t-private due to Lemma 2 with addition and partial multiplicative homomorphic properties by Equation 4.3 and Equation 4.1, respectively. So, any data generated in $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$ retrievals are identically distributed in the presence of t colluding servers.

By these properties, for any data request sequence $\mathbf{x}$, the corresponding transcripts (including access patterns) generated in the $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$ retrieval phase are information-theoretically (statistically) indistinguishable from random access sequence in the presence of up to t colluding servers.

Data are written to slots in the root bucket according to the sequential order and, therefore, the write pattern is deterministic and public. Such written data are SSS-shared with new random polynomials so that they are t-private. Therefore, the *write-to-root* transcripts are identically distributed.

The access patterns of $\{E^{(j)}_{i\in\mathcal{I}}\}$ and $\{E^{(j')}_{i\in\mathcal{I}}\}$ are public because the triplet eviction is deterministic, which follows reverse lexicographical order like Onion-ORAM (*e.g.*, [54]), We show that data generated in such triplet evictions are identically distributed as follows. For each triplet eviction, the client sends $(H+1)$ permutation matrices, which are SSS-shares and, therefore, they are all t-private and uniformly distributed. Data in sibling buckets are t-private and uniformly distributed because they are merely copied from source buckets deterministically (line 19, Figure 4.14). The matrix product (line 22, Figure 4.14) is also t-private due to the security of SMM protocol by Lemma 3 . Therefore, given two request sequences $\mathbf{x}$, $\mathbf{y}$ with $|\mathbf{x}|=|\mathbf{y}|$, the corresponding deterministic triplet eviction sequences observed by $\{S_{i\in\mathcal{I}}\}$ are

$$\{\overrightarrow{E}_{i\in\mathcal{I}}(\bar{\mathbf{x}})\} = (\{E^{(\bar{x}_1)}_{i\in\mathcal{I}}\},\ldots,\{E^{(\bar{x}_{q/A})}_{i\in\mathcal{I}}\})$$

$$\{\overrightarrow{E}_{i\in\mathcal{I}}(\bar{\mathbf{y}})\} = (\{E^{(\bar{y}_1)}_{i\in\mathcal{I}}\},\ldots,\{E^{(\bar{y}_{q/A})}_{i\in\mathcal{I}}\})$$

where $(\bar{x}_j,\bar{y}_j)\in\{0,\ldots,H\}$ for $1\le j\le q/A$. Since data yielded in $\{E^{(\bar{j}_j)}_{i\in\mathcal{I}}\}$ and $\{E^{(\bar{x}_{j'})}_{i\in\mathcal{I}}\}$ are identically distributed for all $(j,j')\in\{\bar{x}_1,\ldots,\bar{x}_{q/A}\}\cup\{\bar{y}_1,\ldots,\bar{y}_{q/A}\}$ as shown above, $\{\overrightarrow{E}_{i\in\mathcal{I}}(\bar{\mathbf{x}})\}$ and $\{\overrightarrow{E}_{i\in\mathcal{I}}(\bar{\mathbf{y}})\}$ are identically distributed.

Given any data request sequence, $\mathsf{S^3ORAM^O}$ generates (*i*) access patterns statistically indistinguishable from a random request sequence of the same length, and (*ii*) identically (uniform) distributed data in the presence of up to t colluding servers. This indicates that $\mathsf{S^3ORAM^O}$ scheme achieves information-theoretic statistical t-security according to Definition 7. □

$\mathsf{S^3ORAM^C}$ follows the Circuit-ORAM eviction strategy [5] so that it inherits the same failure probability as Circuit-ORAM as follows.

Lemma 5 ($\mathsf{S^3ORAM^C}$ *Stash Overflow Probability*)*. Let the bucket size* $Z \geq 2$*. Let* $\mathsf{st}(\mathsf{S^3ORAM^C}[\mathsf{s}])$ *be a random variable denoting the stash size of* $\mathsf{S^3ORAM^C}$ *scheme after an access sequence* s*. Then, for any access sequence* s*,* $\Pr[\mathsf{st}(\mathsf{S^3ORAM^C}[s]) \geq R] \leq 14 \cdot e^{-R}$.

Proof. We refer the reader to [179]. □

The security of $\mathsf{S^3ORAM^C}$ is given in the following theorem.

Theorem 4 ($\mathsf{S^3ORAM^C}$ *Security*)*.* $\mathsf{S^3ORAM^C}$ *is correct and information-theoretically (statistically) t-secure by Definition 7.*

Proof. The correctness and security proof of $\mathsf{S^3ORAM^C}$ can be easily derived from that of $\mathsf{S^3ORAM^O}$ scheme so that we will not present it in detail due to the significant overlap with the proof of Theorem 3. Intuitively, $\mathsf{S^3ORAM^C}$ leverages the same principles as $\mathsf{S^3ORAM^O}$, *i.e.*, SSS-based PIR scheme and permutation matrix, to implement the retrieval and eviction phases, which were proven correct and consistent due to homomorphic properties of SSS and SMM protocol. We also proved that all the data generated by these operations are t-private. The access pattern in $\mathsf{S^3ORAM^C}$ is *statistically* indistinguishable due to its negligible stash overflow probability by Lemma 5. All these properties indicate that $\mathsf{S^3ORAM^C}$ scheme achieves information-theoretic statistical t-security by Definition 7. □

4.4.4 Generalization of $\mathsf{S^3ORAM}$ on k-ary Tree

It is possible to execute ORAM over a general k-ary tree layout to achieve a sub-logarithmic asymptotic overhead (*i.e.*, $\mathcal{O}(\log_k N)$, where k is a free parameter). Our proposed $\mathsf{S^3ORAM}$ schemes can also be easily extended to work over a general k-ary ORAM layout. However, we later show that increasing the value of k does not bring much benefit to the actual performance of tree-based

ORAM schemes. We present $\mathsf{S}^3\mathsf{ORAM}$schemes on the general k-ary tree layout, and then provide the analytical analysis to show that their cost achieves the best at $k \in \{2, 3\}$ as follows.

4.4.4.1 k-ary $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$

We can leverage the concepts of SSS homomorphic computation and the permutation matrix presented in §4.4.2.2 to implement the eviction strategy in [5], which is the generalization of the Triplet Eviction used in $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$. Generally speaking, this strategy requires to organize each bucket in the $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$ k-ary tree layout into k slides, each being of size as a function of the security parameter (*i.e.*, $\mathcal{O}(\lambda)$). In other words, $Z = \mathcal{O}(k \cdot \lambda)$. Each bucket at the leaf level is connected with a so-called auxiliary bucket of size $\mathcal{O}(\lambda)$. The eviction path for k-ary tree is determined by modifying Equation 4.6 to output the order-reversal of base-k digits instead of the binary string. Once the eviction path is determined, we travel from the root to the leaf and obliviously move all blocks from the (non-leaf) source bucket to a deterministic slide of *all* of its children. At the leaf level, we obliviously move all blocks from the leaf bucket to its corresponding auxiliary bucket. All these oblivious moves can be implemented using the SSS matrix product principle described in §4.4.2.2. Notice that in this context, the retrieval phase in $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$ scheme remains unchanged.

4.4.4.2 k-ary $\mathsf{S}^3\mathsf{ORAM}^\mathsf{C}$

$\mathsf{S}^3\mathsf{ORAM}^\mathsf{C}$ scheme in §4.4.2.3 supports the k-ary tree layout naturally without modifying the retrieval and the eviction subroutines. We only need to change Equation 4.6 as similar to the k-ary $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$ scheme as discussed above to get the eviction path in the k-ary tree layout. It also only requires to adjust the bucket size parameter (Z) to be a function of the tree degree to achieve a negligible stash overflow probability. In other words, $Z = \mathcal{O}(k)$ for the statistical security.

We now treat k as a parameter in the asymptotic cost. In this context, the bucket size parameter (Z) is $\mathcal{O}(\kappa \cdot \lambda)$ instead of $\mathcal{O}(\lambda)$. The k-ary $\mathsf{S^3ORAM^O}$ layout is a tree of height $\mathcal{O}(\log_k N)$. The eviction in each level move all blocks from the source bucket of size $\mathcal{O}(\kappa \cdot \lambda)$ to a slide (sized $\mathcal{O}(\lambda)$) of its k children buckets. Thus, the (amortized) client-server and server-server *bandwidth* is $\mathcal{O}(|b|+\lambda \cdot k \cdot \log_k N)$ and $\mathcal{O}(|b| \cdot \lambda \cdot k \cdot \log_k N)$, respectively. The (amortized) server and client *computation* is $\mathcal{O}(|b| \cdot \lambda \cdot k \cdot \log_k N)$ and $\mathcal{O}(|b|+k \cdot \log_k N)$, respectively.

Similarly, we can easily derive the cost of k-ary $\mathsf{S^3ORAM^C}$ scheme, where the bucket size now becomes $Z = \mathcal{O}(k)$. So, the client-server- and server-server-bandwidth of k-ary $\mathsf{S^3ORAM^C}$ are $\mathcal{O}(|b|+k \cdot \log_k N)$ and $\mathcal{O}(|b| \cdot k \cdot \log_k N)$, respectively. The server- and client-computation are $\mathcal{O}(|b| \cdot k \cdot \log_k N)$ and $\mathcal{O}(|b|+k \cdot \log_k N)$, respectively.

So, given that $|b|, \lambda, N$ are unchanged in this context, the computation/bandwidth overhead of k-ary $\mathsf{S^3ORAM}$ schemes can be written as a function of k as

$$f(k) = \alpha + \beta \cdot k \cdot \log_k N = \alpha + \beta \cdot \frac{k}{\ln k} \cdot \ln N, \tag{4.8}$$

where $\alpha \in \{0, |b|\}, \beta \in \{1, |b|, \lambda, \lambda \cdot |b|\}$. Since $k \geq 2$ and $k \in \mathbb{N}$, it is easy to see that $f(k)$ is minimal at $k = 3$.

For $k = 2$, our $\mathsf{S^3ORAM^O}$ using the Triplet Eviction outperforms a constant factor of two compared with using the generalization strategy in [5]. This is because in this case, each source bucket only has one sibling bucket. Due to Equation 4.6, once a bucket is being treated as the sibling bucket, it will be later considered as the destination bucket before being treated as the sibling bucket again. Meanwhile, once the (non-leaf) bucket is treated the destination bucket, it is always guaranteed to be empty after the eviction (see Corollary 3). In other words, the bucket is always empty before being considered as the sibling bucket and, therefore, given a fixed size of $\mathcal{O}(\lambda)$, it always has enough slots to keep the expected load below its capacity. The eviction in [5] does not exploit this special role-switching when $k = 2$, but focuses on the general case for any $k > 2$, where one bucket must serve as the sibling bucket $k - 1$ times before being empty. As a result, each bucket must have k slides each being of size $\mathcal{O}(\lambda)$, and the eviction only touches one slide of the bucket to achieve the sub-logarithmic overhead. Therefore, when $k = 2$, its (eviction and retrieval) overhead is doubled compared with that of the Triplet Eviction. Moreover, given the fact that the generalization eviction with $k = 3$ only gains 6% improvement over $k = 2$, while its $k = 2$ case is two times less efficient than the Triplet Eviction as shown above, we conclude that the $\mathsf{S^3ORAM^O}$ achieves the best performance with $k = 2$ and the Triplet Eviction strategy.

For $\mathsf{S^3ORAM^C}$ scheme, it achieves the best performance at $k = 3$ according to Equation 4.8. This is because it supports k-ary layout naturally without modifying eviction and retrieval subroutine but only adjusting the bucket size parameter. We further demonstrate empirical results to support such analytical analyses in §4.4.5.6.

4.4.5 Experimental Evaluation

4.4.5.1 Implementation

We fully implemented two $\mathsf{S^3ORAM}$ schemes in C++ consisting of roughly 5,000 lines of code. We used two external libraries in our implementation: (1) The Shoup's NTL library v9.10.0[7] for the pseudo-random number generation and arithmetic operations due to its low-level optimization for modular multiplication and cross product functions; (ii) the ZeroMQ library[8] for the network communication. Our implementation supports parallelization via multi-threading to

[7] Available at http://www.shoup.net/ntl/download.html

[8] Available at http://zeromq.org

take full advantage of multi-core CPUs at the server side. We also implemented k-ary tree layout generalization for both $\mathsf{S^3ORAM}$ schemes. The implementation of our $\mathsf{S^3ORAM}$ framework is publicly available at https://github.com/thanghoang/S3ORAM.

We first describe the configuration and methodology to conduct our experiments as follows.

4.4.5.2 Configuration and Methodology

We used a 2015 Macbook Pro laptop as the client, which was equipped with an Intel Core i5-5287U CPU @ 2.90GHz and 16 GB RAM. On the server-side, we used Amazon EC2 with `c4.4xlarge` type to deploy three server instances. Each server was running Ubuntu 16.04 and equipped with 16 vCPUs Intel Xeon E5-2666 v3 @2.9 GHz, 30 GB RAM and 1TB SSD.

We located three servers to be geographically close to each other (same region) as well as to our client machine, which results in the network latency between them being approximately 15 ms. The servers were connected to each other via a dedicated network whose throughput for both download and upload is approximately 1 Gbps. The client used Wi-Fi connecting to the Internet via a home data plan, which offers the latency of 20 ms and the download/upload throughput of 55/6 Mbps to the servers.

We evaluated the performance of all ORAM schemes with a randomly generated database of size from 0.5 GB to 40 GB and block sizes from 4 KB to 1024 KB.

We selected Path-ORAM [169] and Onion-ORAM [54] as the main counterparts of $\mathsf{S^3ORAM}$ framework since the former is the most optimal $\mathcal{O}(\log N)$-bandwidth ORAM (without server computation) while the latter achieves $\mathcal{O}(1)$ bandwidth blowup (with server computation). We also chose Ring-ORAM [150] as it is an efficient $\mathcal{O}(\log N)$-bandwidth ORAM scheme with server computation.

We consider all ORAM schemes (including our $\mathsf{S^3ORAM}$ framework) under their *non-recursive form*, where the position map is stored locally at the client. This is because storing the position map at the server will incur $\mathcal{O}(\log N)$ number of communication rounds of accessing $\mathcal{O}(\log N)$ smaller $\mathsf{S^3ORAM}$, which may result in high overhead. In practice, it is likely that the position map is small enough to be stored locally at the client. Moreover, since $\mathsf{S^3ORAM}$ is only secure in the *semi-honest setting*, we only compared its performance with the semi-honest version of Path-ORAM, Ring-ORAM and Onion-ORAM. We did not consider alternatives that (*i*) failed to achieve $\mathcal{O}(1)$ client communication blowup but incurred more delay (*e.g.*, [52, 130]), (*ii*) were shown to be insecure (*e.g.*, [131, 132]), or (*iii*) incurred more cost than the selected ORAM counterparts above regarding to our configuration and experimental settings (*e.g.*, [10]). We also did not explicitly compare the performance of $\mathsf{S^3ORAM}$ against the multi-server ORAM scheme in [166] because of the major difference in terms of client block storage between the two schemes ($\mathcal{O}(1)$ *vs.* $\mathcal{O}(\sqrt{N})$). Given a very large outsourced database, the storage required by [166] might not be suitable for resource-limited devices such as a mobile phone. Moreover, if $\mathcal{O}(\sqrt{N})$ block storage is acceptable, then the lower bound in [5] might imply a better ORAM strategy than our $\mathsf{S^3ORAM}$ schemes, in which leveraging only PIR technique suffices to achieve $\mathcal{O}(1)$ client bandwidth blowup.

We present the parameter choice and methodology to measure the performance of $\mathsf{S^3ORAM}$ schemes and their counterparts as follows.

- $\mathsf{S^3ORAM}$: For the $\mathsf{S^3ORAM^O}$ scheme, we selected the bucket size $Z = 74$ and $A = Z/2 = 37$ and to achieve the negligible overflow probability of 2^{-80} by Lemma 4. We measured the cost for each $\mathsf{S^3ORAM^O}$ access as the retrieval delay plus the write-to-root delay plus the *amortized* delay of the eviction operation. For the $\mathsf{S^3ORAM^C}$ scheme, we selected the bucket size $Z = 2$

suggested in [179] for the negligible stash overflow probability by Lemma 5. We also investigated the performance of S^{3}ORAM framework with k-ary structure, where $k > 2$. In this case, we fixed the database size and varied the tree height (H) and the bucket size (Z) parameters. (see §4.4.5.6 for the detailed configuration).

- Path-ORAM: We selected the bucket size $Z = 4$ to achieve the negligible stash overflow probability of 2^{-80}. We measured the delay of Path-ORAM as the time to download and upload $4 \cdot \log_2 N$ blocks plus the delay of IND-CPA decryption and re-encryption of these blocks at the client. We used `libtomcrypt`[9] to implement AES-CTR as the IND-CPA encryption.

- Ring-ORAM: We selected Ring-ORAM parameters (*i.e.*, $Z = 16$, $S = 25$ and $A = 20$) as stated in [150] for a negligible stash overflow probability of 2^{-80}. We measured the delay of Ring-ORAM as the total time of (*i*) one block transmission, (*ii*) XOR and IND-CPA encryption/decryption operations at the client, (*iii*) XOR operations at the server and (*iv*) the amortized cost of eviction and early shuffles based on the formula $(H + 1)(2Z + S)/A \cdot (1 + \mathsf{PoissCDF}(S, A))$ given in [150].

- Onion-ORAM: We selected the size of RSA modulus to be 1024 bits for AHE according to [9]. We selected the bucket size and the eviction frequency of Onion-ORAM as $Z = 74$ and $A = Z/2 = 37$ for the negligible bucket overflow probability of 2^{-80}. We measured the overall delay of Onion-ORAM as the time to (1) perform homomorphic computations at the client and server and (2) transfer $\mathcal{O}(1)$ blocks and PIR queries, plus the amortized cost of eviction operation. Since Onion-ORAM is extremely computationally costly, measuring its delay even on a medium database takes an insurmountable amount of time. Therefore, we only measured its delay on a small database (*i.e.*, 1 MB) first, and then extrapolate the delay for larger database sizes.

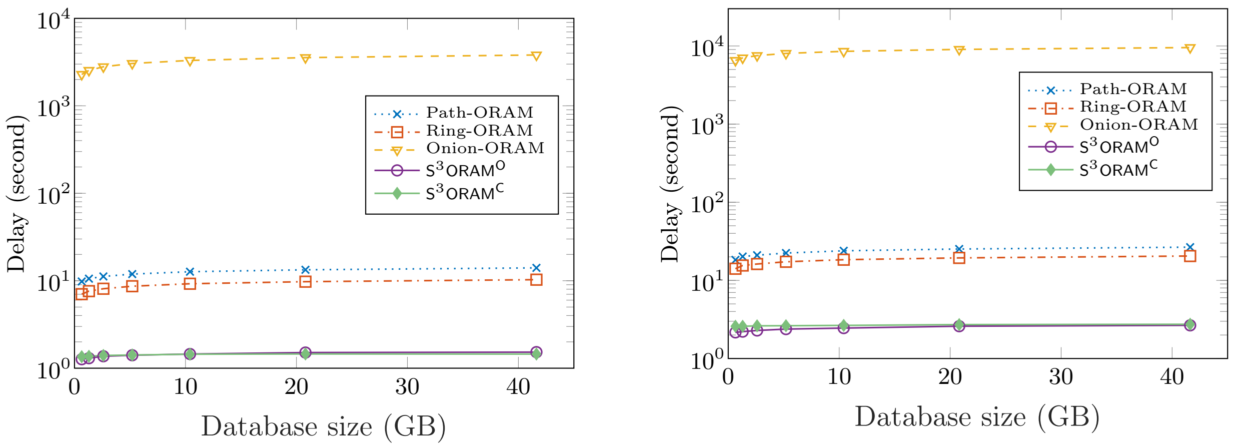
[9] Available at https://github.com/libtom/libtomcrypt

Table 4.3: Amount of data to be sent by the client and processed by the server(s).

# Blocks	Retrieval Phase				Eviction Phase			
	Query Size (KB)		# Computed Blocks		Permutation Matrix Size (KB)		# Computed Blocks	
	$\mathsf{S^3ORAM^O}$	$\mathsf{S^3ORAM^C}$	$\mathsf{S^3ORAM^O}$	$\mathsf{S^3ORAM^C}$	$\mathsf{S^3ORAM^O}$	$\mathsf{S^3ORAM^C}$	$\mathsf{S^3ORAM^O}$	$\mathsf{S^3ORAM^C}$
10^3	4.05	0.17	518	20	598.93	1.55	76,664	198
10^4	6.36	0.23	814	28	941.19	2.11	120,472	270
10^5	8.09	0.28	1,036	34	1,197.88	2.53	153,328	324
10^6	9.82	0.33	1,258	40	1,454.56	2.95	186,184	378
10^7	12.14	0.39	1,554	48	1,796.81	3.52	229,992	450
10^8	13.88	0.44	1,776	54	2,053.50	3.94	262,848	504
10^9	15.61	0.48	1,998	60	2,310.19	4.36	295,704	558

4.4.5.3 End-to-End Delay

We first present the analytical communication and computation overhead of $\mathsf{S^3ORAM}$ schemes with databases containing a various number of data blocks in Table 4.3. We can see that the size of the retrieval query and permutation matrices by the client as well as the amount of the data to be computed by the server are much lower than $\mathsf{S^3ORAM^O}$ for each access. This is mainly because $\mathsf{S^3ORAM^C}$ has a much smaller bucket size than $\mathsf{S^3ORAM^O}$ (*i.e.*, 2 *vs.* 74). However, the eviction in $\mathsf{S^3ORAM^O}$ is only performed after every 37 accesses compared with one in $\mathsf{S^3ORAM^C}$ scheme. In fact, this accumulative strategy allows $\mathsf{S^3ORAM^O}$ to be less impacted by the network congestion control (*i.e.*, TCP slow-start) and the client-server/ server-server communication latency than $\mathsf{S^3ORAM^C}$. Moreover, $\mathsf{S^3ORAM^O}$ incurs only one block to be uploaded per access compared with two in $\mathsf{S^3ORAM^O}$ scheme. All these factors result in the *amortized* end-to-end delay of $\mathsf{S^3ORAM^O}$ scheme being comparable with the actual delay of $\mathsf{S^3ORAM^C}$ as shown in Figure 4.19, even though its analytical overhead looks worse than that of $\mathsf{S^3ORAM^C}$. We also show in Figure 4.19 the simulated delay of $\mathsf{S^3ORAM}$ counterparts with different database sizes (from 0.5 to 40 GB) and block sizes (128 KB and 256 KB). Both $\mathsf{S^3ORAM}$ schemes took only 1.3-1.4 (*resp.* 2.1-2.7) seconds to access a 128 KB (*resp.* 256 KB) block in the database of size up to 40 GB. This resulted in $\mathsf{S^3ORAM}$ schemes being approximately 9.3 and 6.4 times faster than Path-ORAM and Ring-ORAM, where they took 7-14 (*resp.* 14-26) seconds for each 128 KB (*resp.* 256 KB) block access. Compared with



(a) Block size = 128 KB

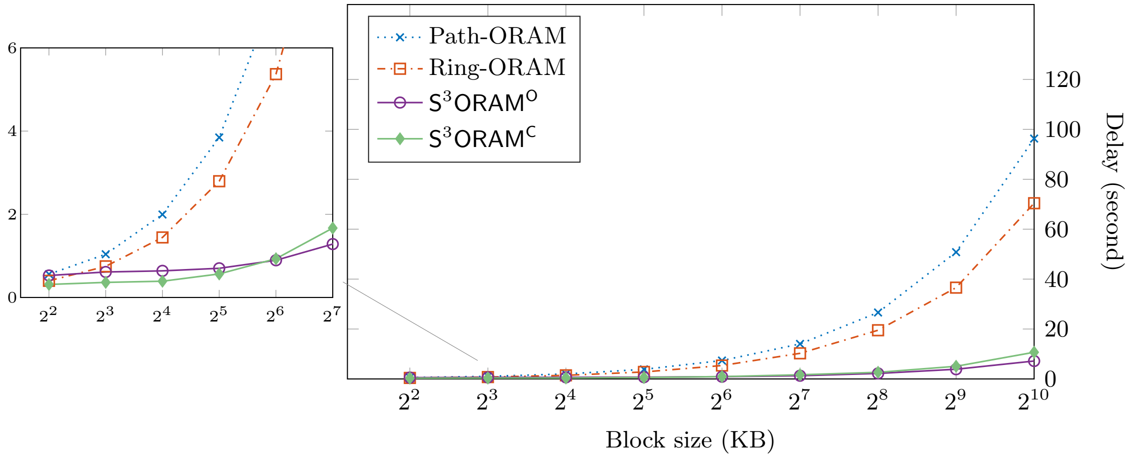
(b) Block size = 256 KB

Figure 4.19: Delay of S^{3}ORAM and its counterparts on a laptop with home network.

Onion-ORAM, our S^{3}ORAM schemes were three orders of magnitude faster. This is mainly due to the fact that S^{3}ORAM schemes only rely on simple arithmetic operations (*e.g.*, modular addition/ multiplication), while Onion-ORAM leverages Partially/Fully HE (see §4.4.5.4). One might also observe from Figure 4.19 that choosing a larger block size has a small impact on the delay of S^{3}ORAM schemes. This is clearly illustrated in Figure 4.20, where we present the impact of block size on the end-to-end delay of S^{3}ORAM schemes compared with their counterparts. Given any block size ranging from 4 KB to 1024 KB, S^{3}ORAM schemes always maintain a constant factor of 9.3 and 6.4 times faster than Ring-ORAM and Path-ORAM, respectively. This presents an advantage to S^{3}ORAM schemes over their counterparts for applications requiring large block sizes such as image or video storage services.

4.4.5.4 Cost Breakdown Analysis

In this section, we dissect the overall delay of S^{3}ORAM to explore the factors that contribute the most to the total delay. Figure 4.21 shows the detailed cost factors of two S^{3}ORAM schemes



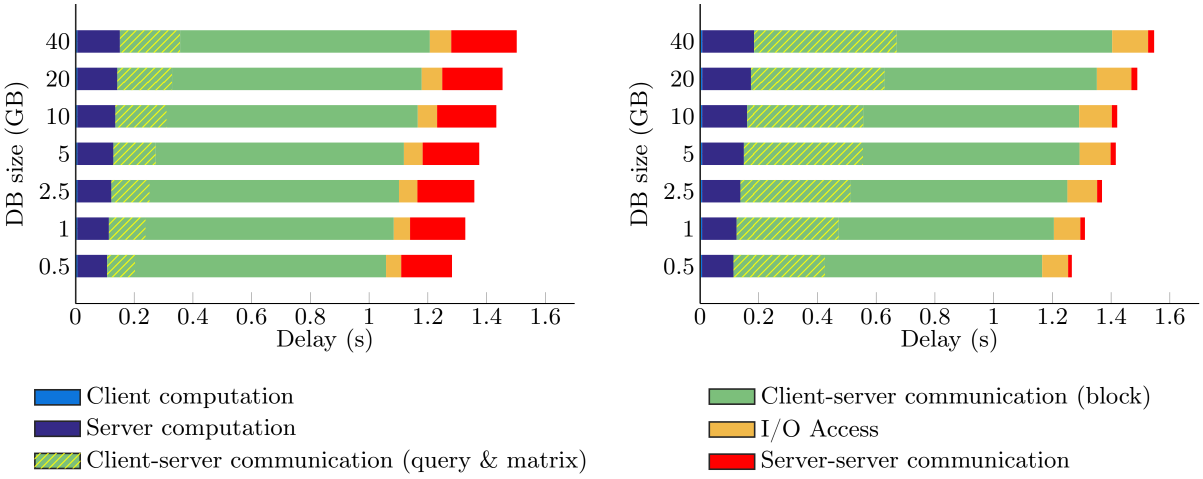
*We excluded Onion-ORAM since its plot is far beyond the limit of y-axis.

Figure 4.20: Delay of $\mathsf{S^3ORAM}$ for varying block sizes for a 40GB DB.

according to 0.5-40 GB DB with 128 KB blocks. There are five factors that affect the overall delay of $\mathsf{S^3ORAM}$ schemes as follows.

1. *Client computation:* In both $\mathsf{S^3ORAM}$ schemes, the client computed the SSS shares of the retrieval query and the permutation matrices, recovered the requested block and re-shared the block with SSS. All these incurred only some modular addition and multiplication operations. These computations are extremely lightweight so that the client computation contributed only a minimal amount to the total delay (*i.e.*, $< 1\%$), which is hard to observe in both Figure 4.21a and Figure 4.21b.

2. *Server computation:* In both $\mathsf{S^3ORAM}$ schemes, the servers computed the ORAM tree data with the retrieval query via the dot product and with permutation matrices via the matrix product, which also incurred a series of modular addition and multiplication operations. However, unlike the client computation, the cost of these operations at the server-side depends on the block size. As a result, the server computation contributed a higher amount to the total delay (*i.e.*, 7-11%) than the client computation. Compared between two $\mathsf{S^3ORAM}$ schemes, we can see



(a) $\mathsf{S^3ORAM^C}$

(b) $\mathsf{S^3ORAM^O}$

Figure 4.21: Detailed cost breakdown of $\mathsf{S^3ORAM}$ on a laptop with home network.

that $\mathsf{S^3ORAM^O}$ had a higher server computation delay than $\mathsf{S^3ORAM^C}$ scheme. This is because the block size of $\mathsf{S^3ORAM^O}$ is much larger than $\mathsf{S^3ORAM^C}$ (*i.e.*, 74 *vs.* 2), which significantly impacts the SSS-based PIR computation in the retrieval phase.

3. *Client-server communication:* In both $\mathsf{S^3ORAM}$ schemes, this operation contributed the most to the total delay (over 90%). For each $\mathsf{S^3ORAM}$ access, the client downloaded one block from the servers and uploaded 1-2 blocks along with one retrieval query and some permutation matrices. We can observe from Figure 4.21 that the time to upload the retrieval query and permutation matrices (*yellow-patterned green bars*) was much faster than the time to download and upload a 128 KB data block (*unpatterned green bars*). This clearly reflects the theoretical insight of $\mathsf{S^3ORAM}$ schemes, where the client communication overhead is constant and mostly dominated by the data block with the poly-logarithmic size. We can also observe that $\mathsf{S^3ORAM^O}$ (Figure 4.21b) took a longer time to transmit the retrieval query and permutation matrices than $\mathsf{S^3ORAM^C}$ (Figure 4.21a). This is because the bucket size parameter in $\mathsf{S^3ORAM^O}$ scheme is much larger than in $\mathsf{S^3ORAM^C}$ as explained above, which impacts the size of the retrieval

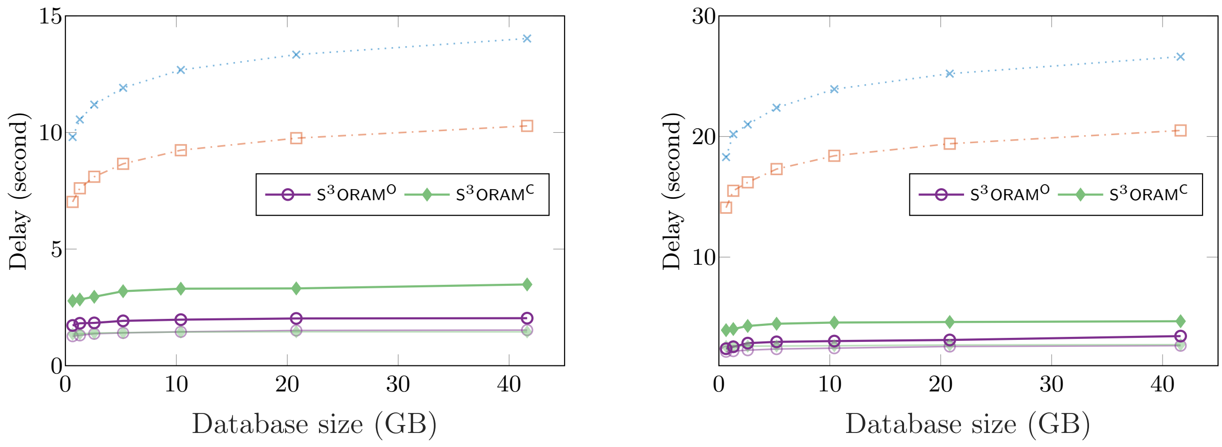
vector and the eviction matrices. On the other hand, the block transmission time in $\mathsf{S^3ORAM^C}$ was doubly slower than in $\mathsf{S^3ORAM^C}$. This is because the eviction in $\mathsf{S^3ORAM^C}$ requires to transmit *two* blocks for each access, compared with only *one* in $\mathsf{S^3ORAM^O}$.

4. *I/O access:* Due to the cache miss issue and the infrastructure of the selected Amazon EC2 instances (*i.e.*, `c4.4xlarge`), the disk I/O access caused a considerable delay especially in $\mathsf{S^3ORAM^O}$ scheme. Specifically, we stored the $\mathsf{S^3ORAM}$ tree in a network storage unit called "Elastic Block Storage" (EBS), which was connected to Amazon EC2 computing unit with a maximum throughput of 160 MB/s. This resulted in the I/O access being limited by this throughput, and therefore, causing a high delay. To reduce the I/O access overhead, one solution is to store the $\mathsf{S^3ORAM}$ tree structure on a local storage unit with high throughput (*e.g.*, NVMe). Another solution is to apply a caching strategy, where h-top levels of the $\mathsf{S^3ORAM}$ tree are stored directly on RAM. As explained above, $\mathsf{S^3ORAM^O}$ has a larger bucket size than $\mathsf{S^3ORAM^C}$ so that its reported I/O delay was higher than $\mathsf{S^3ORAM^C}$.

5. *Server-server communication:* This overhead was caused by the SMM protocol when the servers performed the matrix product operation in the eviction phase. In $\mathsf{S^3ORAM^O}$ scheme, the reported communication delay between the servers was very low, and significantly faster than in $\mathsf{S^3ORAM^C}$ scheme. This is because of the amortization in $\mathsf{S^3ORAM^O}$ scheme, where the eviction was performed after every $A = 37$ subsequent retrievals. This context allowed the network latency (*i.e.*, 15 ms) to be amortized and minimized the impact of the TCP slow start scheme. In $\mathsf{S^3ORAM^C}$, the eviction must be performed right after each retrieval so that its reported delay was significantly impacted by those factors.

4.4.5.5 The Impact of Network Quality

We first investigated the impact of inter-server network quality on the performance of $\mathsf{S}^3\mathsf{ORAM}$ schemes. In this setting, we set up three Amazon EC2 servers to be geographically distant to each other (in the form of a triangle between California, Ohio and Central Canada regions). The average network round-trip latency and throughput between the servers were 78 ms and 295 Mbps, respectively. The round-trip latency between the client and the farthest server was 80 ms, while the client throughput to all servers remained unchanged (*i.e.*, 55/6 Mbps of download/upload speed). Figure 4.22 presents the delay of $\mathsf{S}^3\mathsf{ORAM}$ in this setting compared with the previous one. We can see that $\mathsf{S}^3\mathsf{ORAM}$ schemes performed 0.3–2 s (2× at most) slower than in the original setting, where all servers were in the same region and close to the client. This slowdown is mostly due to (*i*) the higher latency and lower throughput of the inter-server network link and (*ii*) the latency when the client communicates with the farthest server. However, as shown in Figure 4.22, $\mathsf{S}^3\mathsf{ORAM}$ still outperformed the performance of Path-ORAM and Ring-ORAM in the original setting (*i.e.*, server was placed close to the client). This is because the server-server communication only contributed a small fraction in the total delay, especially in $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$ scheme (due to the amortization) as already analyzed in §4.4.5.4. On the other hand, $\mathsf{S}^3\mathsf{ORAM}$ incurred only one communication round between the client and the servers so that the impact of the client's high round-trip latency was minimal. Moreover, the client throughput remained unchanged and therefore, it did not impact much on the delay of $\mathsf{S}^3\mathsf{ORAM}$ schemes in this context. We observed that $\mathsf{S}^3\mathsf{ORAM}^\mathsf{C}$ was more impacted by the inter-server high network round-trip latency than $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$. This is because $\mathsf{S}^3\mathsf{ORAM}^\mathsf{C}$ performed eviction right after each access, where the servers communicated with each other in $\mathcal{O}(\log N)$ rounds. Meanwhile, these rounds were performed once every $A = 37$ accesses in $\mathsf{S}^3\mathsf{ORAM}^\mathsf{O}$ and therefore, their total latency was amortized.

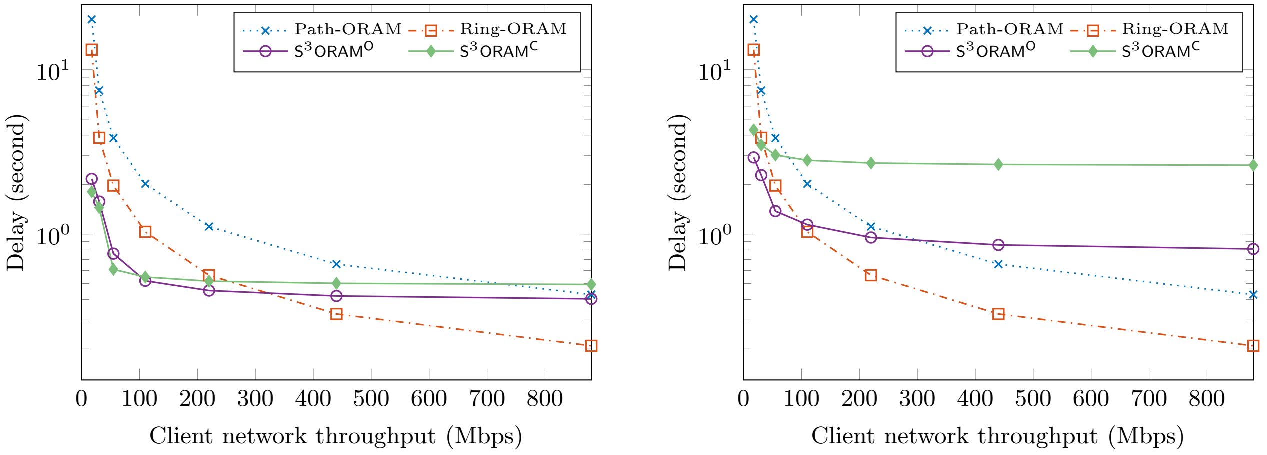


(a) Block size = 128 KB

(b) Block size = 256 KB

Figure 4.22: The delay of S^{3}ORAM schemes with geographically distant servers.

Given that the low network quality at both client and inter-server sides did not impact much on the delay of S^{3}ORAM schemes, we now show that if the client can have a high-speed network setting, our S^{3}ORAM framework might no longer be an ideal choice. We conducted an experiment to demonstrate that ORAM schemes featuring $\mathcal{O}(\log N)$ bandwidth overhead are better than S^{3}ORAM after a certain threshold of network bandwidth. Figure 4.23 presents simulated performance of S^{3}ORAM schemes and their counterparts with different client network bandwidth settings regarding 40 GB database containing 128 KB blocks. With 1 Gbps inter-server network throughput (servers were at the same region), Path-ORAM and Ring-ORAM surpassed S^{3}ORAM schemes for a client network throughput of approximately 720 Mbps and 380 Mbps, respectively (Figure 4.23a). Given servers were set up geographically distant to each other, the corresponding numbers were 80-300 Mbps and 50-100 Mbps (Figure 4.23b). This is because Path-ORAM and Ring-ORAM feature $\mathcal{O}(\log N)$ bandwidth overhead so that they receive a more benefit from the high network speed. On the other hand, S^{3}ORAM schemes feature $\mathcal{O}(1)$ bandwidth overhead and therefore, get less benefit.



(a) 1 Gbps inter-server network throughput (b) 295 Mbps inter-server network throughput

Figure 4.23: The impact of client network throughput.

4.4.5.6 The Impact of k-ary Tree Layout

We performed an empirical analysis to confirm our finding in §4.4.4 that increasing the degree of the ORAM tree receives very little benefit if not worse than the default setting (*i.e.*, binary tree).

Figure 4.24 presents the actual end-to-end delay of S^{3}ORAM schemes with varied tree degrees under the fixed 1TB DB with 128 KB blocks configuration. We can see that the actual delay of k-ary S^{3}ORAM schemes likely matched with the expected overhead (*the dash-dotted line*). As discussed, the performance of S^{3}ORAM^{O} following the generalized eviction in [5] achieved the best at $k = 3$

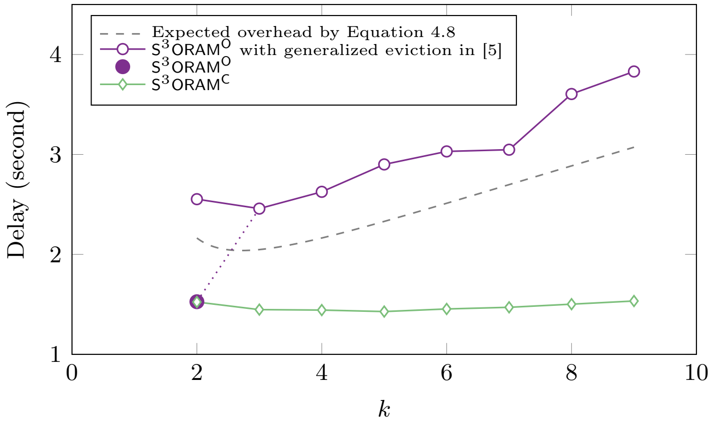


Figure 4.24: The impact of tree degree.

(*the solid purple line*). Remark that for the special case where $k = 2$, such generalized eviction did not take into account the bucket load characteristic after each eviction for optimization. Meanwhile, this characteristic was fully exploited in the Triplet Eviction strategy, which allowed reducing the end-to-end delay by half (*the solid purple point with the dashed purple line*). In summary, considering a little gain that $k = 3$ can offer and the optimization that can be done with $k = 2$, we can see that $\mathsf{S^3ORAM^O}$ scheme achieved the best performance at the default setting (*i.e.*, binary tree layout with the Triplet Eviction). In $\mathsf{S^3ORAM^C}$ scheme, the actual performance followed closely to the analytical result, which achieved the best performance at around $k \in \{3, 4, 5\}$. However, the gain was not so considerable compared with $k = 2$ (*i.e.*, $< 6.5\%$). At $k > 5$, the delay of $\mathsf{S^3ORAM^C}$ scheme started to increase and became worst than $k = 2$.

4.4.5.7 Storage Overhead

At the client-side, $\mathsf{S^3ORAM^O}$ does not require the stash component similar to Onion-ORAM. On the other hand, $\mathsf{S^3ORAM^C}$ requires the stash of size $\mathcal{O}(\lambda \cdot |b|)$ similar to Path-ORAM and Ring-ORAM. Therefore, given a database containing 512 KB blocks, $\mathsf{S^3ORAM^C}$ scheme needs around 32-33 MB of the client storage for the stash, while $\mathsf{S^3ORAM^O}$ requires nothing. The storage cost for the position map component in both (non-recursive) $\mathsf{S^3ORAM}$ schemes is slightly higher than their non-recursive counterparts. For instance, with a 16 TB database of 512-KB blocks ($D = 33, 554, 432$), $\mathsf{S^3ORAM}$ schemes cost 119 MB while the others (*e.g.*, Onion-ORAM, Ring-ORAM, Path-ORAM) cost 100 MB. This is because we store not only the path information but also the specific location of each block in its assigned path.

At the server side, *each* server storage overhead in $\mathsf{S^3ORAM^O}$ scheme increases by a factor of eight (*i.e.*, $(8D-A)\cdot|b|$ bits) by Lemma 4. The server storage overhead for $\mathsf{S^3ORAM^C}$ scheme increases by a factor of two (*i.e.*, $2D \cdot |b|$), which is equal to Circuit-ORAM. Recall that all $\mathsf{S}^3\mathsf{ORAM}$ schemes need at least three servers. The server storage for Path-ORAM and Ring-ORAM is $4D \cdot |b|$ bits and $6D \cdot |b|$ bits, respectively. The server storage for Onion-ORAM is similar to $\mathsf{S}^3\mathsf{ORAM}^{\mathsf{O}}$ for one server but will increase after a sequence of access operation due to the ciphertext expansion of Additively HE.

We analytically compare $\mathsf{S}^3\mathsf{ORAM}$ schemes with state-of-the-art multi-server ORAM schemes for data outsourcing. The most notable ORAM relevant to our framework is Multi-Cloud Oblivious Storage (MCOS) [166] as it also features $\mathcal{O}(1)$ client communication overhead at the cost of $\mathcal{O}(\log N)$ server-server bandwidth overhead like $\mathsf{S}^3\mathsf{ORAM}$. MCOS is better than $\mathsf{S}^3\mathsf{ORAM}$ in the several aspects as follows. First, it does not require a minimal block size to achieve the constant client communication overhead, while $\mathsf{S}^3\mathsf{ORAM}$ requires $\Omega(\log^2 N)$ - $\Omega(\log^3 N)$ block size. Second, it needs two servers to operate while $\mathsf{S}^3\mathsf{ORAM}$requires at least three servers. The main downside of MCOS over $\mathsf{S}^3\mathsf{ORAM}$is that it requires the client to store $\mathcal{O}(\sqrt{N})$ data blocks compared with $\mathcal{O}(1) - \mathcal{O}(\log N)$. For instance, with 256 TB database with 2^{32} blocks, the client storage is 15 GB (*vs.* $\{0, 8\}$ MB in $\mathsf{S}^3\mathsf{ORAM}$). Another distributed ORAM relevant to $\mathsf{S}^3\mathsf{ORAM}$ is the two-server ORAM scheme by Lu and Ostrovsky *et al.* [126]. Due to the hierarchical ORAM paradigm [75], the main advantage of this scheme is that the client does not need to maintain the position map and the stash components as in partition-based and tree-based ORAM schemes including $\mathsf{S}^3\mathsf{ORAM}$ and MCOS. However, it incurs $\mathcal{O}(\log N)$ client communication overhead as opposed to $\mathsf{S}^3\mathsf{ORAM}$ and MCOS. As a result, it can operate on any block size and all the servers do not need to communicate with each other.

4.5 MACAO: A Multi-Server ORAM Framework with Active Security

While S^{3}ORAM enables desirable performance properties for the client such as low communication overhead and low delay, it only offers security against semi-honest adversary. In practice, it is likely the *active* adversary may be present, who may deviate from the protocol to compromise the user privacy and data integrity. To address this issue, we design MACAO, a comprehensive MAliciously-secure and Client-efficient Active ORAM framework. MACAO harnesses suitable secret sharing techniques, efficient eviction strategy along with information-theoretic Message Authentication Code (MAC), which (*i*) offers integrity check, (*ii*) prevents malicious behaviors and (*iii*) achieves a comparable efficiency to state-of-the-art ORAM schemes simultaneously. Our MACAO framework comprises two main multi-server ORAM schemes: MACAO$_{\mathsf{rss}}$ and MACAO$_{\mathsf{spdz}}$. We design MACAO$_{\mathsf{rss}}$ based on replicated secret sharing [105], which requires three servers and there is no collusion among the servers (privacy level $t = 1$). On the other hand, MACAO$_{\mathsf{spdz}}$ is built on SPDZ secret sharing [51] following the preprocessing model, which can operate in the ℓ-server setting ($\ell \geq 2$) with the optimal level of privacy (*i.e.*, $t = \ell - 1$). We construct a series of authenticated PIR protocols based on RSS and SPDZ and prove that they are secure against the malicious adversary. Additionally, we propose several optimization tricks to reduce the bandwidth overhead at the cost of reducing information-theoretic to computational security. Table 4.4 outlines some key characteristics of MACAO compared with state-of-the-art ORAM schemes.

In summary, our main contributions are as follows.

- *Multi-server active ORAM with security against active adversaries:* MACAO offers data confidentiality and integrity, access pattern obliviousness in the presence of malicious adversaries.

Table 4.4: Summary of state-of-the-art ORAM schemes.

Scheme	Bandwidth Overhead†		Block Size*	Client Block Storage‡	# servers§	Security	Comp. over Enc. Data
	Client-server	Server-server					
Ring-ORAM [150]	$\mathcal{O}(\log N)$	-	$\Omega(1)$	$\mathcal{O}(\log N)$	1	Semi-Honest	✗
CKN+18 [40]	$\mathcal{O}(\log N)$	-	$\Omega(\log^2 N)$	$\mathcal{O}(1)$	3	Semi-Honest	✗
GKW18 [77]	$\mathcal{O}(\log N)$	-	$\Omega(1)$	$\mathcal{O}(\log N)$	2	Semi-Honest	✗
S^{3}ORAM[89]	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$	$\Omega(\log^2 N)$	$\mathcal{O}(1)$	$2t+1$	Semi-Honest	✓
Path-ORAM [169]	$\mathcal{O}(\log N)$	-	$\Omega(1)$	$\mathcal{O}(\log N)$	1	Malicious	✗
Circuit-ORAM [179]	$\mathcal{O}(\log N)$	-	$\Omega(1)$	$\mathcal{O}(\log N)$	1	Malicious	✗
SS13 [166]	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$	$\Omega(\log^2 N)$	$\mathcal{O}(\sqrt{N})$	2	Malicious	✗
LO13 [126]	$\mathcal{O}(\log N)$	-	$\Omega(1)$	$\mathcal{O}(1)$	2	Malicious	✗
Onion-ORAM [54]	$\mathcal{O}(1)$	-	$\Omega(\log^6 N)$	$\mathcal{O}(1)$	1	Malicious	✓
MACAO (MACAO$_{\mathsf{rss}}$)	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$	$\Omega(\log N)$	$\mathcal{O}(\log N)$	3	Malicious	✓
MACAO (MACAO$_{\mathsf{spdz}}$)					$t+1$		

†*Bandwidth overhead* denotes the number of blocks being transmitted between the client and the server(s) or between the servers.
*This indicates the minimal block size needed to absorb the transmission cost of the retrieval query and the eviction instructions, thereby achieving the desirable client-bandwidth overhead.
‡*Client block storage* is defined as the number of data blocks being temporarily stored at the client. This is equivalent to the stash component used in [150, 169], which, therefore, does not include the cost of storing the position map of size $\mathcal{O}(N \log N)$. Notice that all the ORAM schemes in this table, except [77, 126], *require* such a position map component. However, we can apply recursive technique in [161] to store the position map on the server at the cost of increasing a small number of communication rounds [161, 166].
§S^{3}ORAM and MACAO$_{\mathsf{spdz}}$ offer the property that allows a certain number of colluding servers in the system (privacy level $t \geq 1$) by increasing the number of servers. Other multi-server ORAM schemes do not offer this scalability ($t = 1$) efficiently, and require a fixed number of servers.

MACAO enables the client to detect, with high probability, if the malicious server(s) has tampered with the inputs/outputs of the protocol.

- *Oblivious distributed file system applications and secure computation:* Our MACAO framework relies on secret sharing as the core building block, which offers additive and multiplicative homomorphic properties. Therefore, after a block is accessed, it can be computed further directly on the server(s). This property permits MACAO to serve as a core building block towards designing a full-fledged Oblivious Distributed File System (ODFS) with secure computation capacity.

- *Full-fledged implementation and performance evaluation:* We fully implemented MACAO framework and compared its performance with state-of-the-art ORAM schemes on real-cloud platforms (*i.e.*, Amazon EC2). Our experimental results confirmed the efficiency of MACAO, in which it is up to seven times faster than single-server ORAMs. We provide detail cost analysis of MACAO schemes in §4.5.3.5.

In addition, it is important to point out that in the context of a multi-server active ORAM scheme with malicious security, we also achieve the following important properties, previously only attained in passive schemes in the semi-honest setting.

- *Low client storage and communication overhead:* MACAO offers $\mathcal{O}(1)$ client-bandwidth overhead, compared with $\mathcal{O}(\log N)$ of the most efficient passive ORAM schemes (*e.g.*, [150, 169]). Moreover, MACAO features a smaller block size than other active ORAM schemes that also achieve the constant client-bandwidth blowup (*e.g.*, S^{3}ORAM [89], Onion-ORAM [54], Bucket-ORAM [61]). Observe that while asymptotically comparable to [54, 61], in practice MACAO schemes are more efficient since they feature a smaller block size.

- *Low computational overhead at both client and server sides:* In MACAO, the client and server(s) only perform bit-wise and arithmetic operations (*e.g.*, addition, multiplication) during the *online* access. This is more efficient than other ORAM schemes requiring heavy computation due to partially/fully HE [54]. MACAO offers up to three orders of magnitude improvement over Onion-ORAM [54] thanks to the fact that all HE operations in MACAO are pre-computed in the *offline* phase between the servers and *independent* of the client and on-line (read or eviction) access phase. Therefore, the online access latency of MACAO$_{\mathsf{spdz}}$ is not impacted by the delay of HE. On the other hand, MACAO$_{\mathsf{rss}}$ does not require any pre-computation or HE operations. Due to the efficient computation at both client and server sides and the low client-bandwidth overhead, MACAO achieves low end-to-end delay to access a large block in a large database in real-world settings.

As a final remark, observe that we focus on oblivious access in the single-client setting, where the client is fully trusted. This is in contrast to some of the distributed ORAM research targeting the fully distributed model, where there is no trusted party (*i.e.*, client) at all [56, 59, 114, 181]. The problem of multi-client access to ORAM as in [125, 128, 129, 158, 186] is also outside of the scope of this study.

4.5.1 System and Security Models

4.5.1.1 System Model

Our system model consists of a client and ℓ servers $(S_0, \ldots, S_{\ell-1})$. We assume that the channels between all the players are pairwise-secure. That is, no player can tamper with, read, or modify the contents of the communication channel of other players. We define a multi-server ORAM scheme as follows.

Definition 8 (Multi-Server ORAM). A Multi-server ORAM scheme is a tuple of two PPT algorithms $\mathsf{ORAM} = (\mathsf{Setup}, \mathsf{Access})$ as follows.

- $\overrightarrow{\mathbf{T}} \leftarrow \mathsf{Setup}(\mathsf{DB}, 1^\lambda)$: Given database DB and security parameter λ as input, it outputs a distributed data structure $\overrightarrow{\mathbf{T}}$.
- $\mathsf{data}' \leftarrow \mathsf{Access}(\mathsf{op}, \mathsf{id}, \mathsf{data})$: Given an operation type $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, an ID id of the block to be accessed, a data data, it outputs a block content data' to the client.

4.5.1.2 Security Model

The client is the only trusted party. The servers are untrusted and can behave maliciously, in which they can tamper with the inputs and/or outputs of the ORAM protocol. Our security model captures the privacy and verifiability of the honest client in the presence of a malicious adversary corrupting a number of servers in the system. The privacy property ensures that the adversary cannot infer the client access pattern or database content. The verifiability ensures that the client is assured to gain access to the trustworthy data from the server with integrity guarantee, and they can detect and abort the protocol if one of the servers cheats. Following the simulation-based security model in multi-party computation [32] and single-server ORAM [54], we define the security of multi-server ORAM in the malicious setting by augmenting the $\mathsf{S^3ORAM}$ security model [89] to account for malicious adversaries as follows.

Definition 9 (Simulation-Based Multi-Server ORAM Security with Verifiability). We first define the ideal and real worlds as follows.

Let $\mathcal{F}_{\mathsf{oram}}$ be an ideal functionality, which maintains the latest version of the database on behalf of the client, and answers the client's requests as follows.

- *Setup:* An environment $\mathcal{Z}$ provides a database DB to the client. The client sends DB to the ideal functionality $\mathcal{F}_{\mathsf{oram}}$. $\mathcal{F}_{\mathsf{oram}}$ notifies the simulator $\mathcal{S}_{\mathsf{oram}}$ the completion of the setup operation and the DB size, but not the DB content. $\mathcal{S}_{\mathsf{oram}}$ returns ok or abort to $\mathcal{F}_{\mathsf{oram}}$. $\mathcal{F}_{\mathsf{oram}}$ then returns ok or $\perp$ to the client accordingly.

- *Access:* For each access, the environment $\mathcal{Z}$ specifies an operation $\mathsf{op} \in \{\mathsf{read}(\mathsf{id}, \perp), \mathsf{write}(\mathsf{id}, \mathsf{data})\}$ as the client's input, where id is the ID of the block to be accessed and data is the block data to be updated. The client sends op to $\mathcal{F}_{\mathsf{oram}}$. $\mathcal{F}_{\mathsf{oram}}$ notifies the simulator $\mathcal{S}_{\mathsf{oram}}$ (without revealing the operation op to $\mathcal{S}_{\mathsf{oram}}$). If $\mathcal{S}_{\mathsf{oram}}$ returns ok to $\mathcal{F}_{\mathsf{oram}}$, $\mathcal{F}_{\mathsf{oram}}$ sends $\mathsf{data}' \leftarrow \mathsf{DB}[\mathsf{id}]$ to the client, and updates $\mathsf{DB}[\mathsf{id}] \leftarrow \mathsf{data}$ accordingly if $\mathsf{op} = \mathsf{write}$. The client then returns the block data data' to the environment $\mathcal{Z}$. If $\mathcal{S}_{\mathsf{oram}}$ returns abort to $\mathcal{F}_{\mathsf{oram}}$, $\mathcal{F}_{\mathsf{oram}}$ returns $\perp$ to the client.

In the real world, an environment $\mathcal{Z}$ gives the client a database DB. The client executes Setup protocol with servers $(S_0, \ldots, S_{\ell-1})$ on DB. For each access, $\mathcal{Z}$ specifies an input $\mathsf{op} \in \{\mathsf{read}(\mathsf{id}, \perp$

$), \mathsf{write}(\mathsf{id}, \mathsf{data})\}$ to the client. The client executes Access protocol with servers $(S_0, \ldots, S_{\ell-1})$. The environment $\mathcal{Z}$ gets the view of the adversary $\mathcal{A}$ after every operation. The client outputs to the environment $\mathcal{Z}$ the data of block with ID id being accessed or abort (indicating abort).

We say that a protocol $\Pi_{\mathcal{F}}$ securely realizes the ideal functionality $\mathcal{F}_{\mathsf{oram}}$ in the presence of a malicious adversary corrupting t servers iff for any PPT real-world adversary that corrupts up to t servers, there exists a simulator $\mathcal{S}_{\mathsf{oram}}$, such that for all non-uniform, polynomial-time environment $\mathcal{Z}$, there exists a negligible function negl such that

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F}_{\mathsf{oram}},\mathcal{S}_{\mathsf{oram}},\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda).$$

4.5.2 The Proposed MACAO Framework

In this section, we describe our ORAM framework in detail. We first present how state-of-the-art active ORAM schemes are vulnerable against the active adversary. We then develop sub-protocols that are used to build our framework.

4.5.2.1 ORAM in the Malicious Setting

In passive ORAM schemes (*e.g.*, [161, 169, 179]), the server only acts as a storage-only service, which processes data sending and receiving requests by the client. Therefore, any malicious behavior can be easily detected by creating a MAC for each ORAM block being requested [169]. Malicious security is more difficult to achieve in the active ORAM setting, where the client delegates the computation to the server for reduced bandwidth overhead. Next, we review the attack introduced in [54] to illustrate the vulnerability of state-of-the-art active ORAM schemes in the malicious setting.

Most efficient active ORAMs (*e.g.*, [52, 89, 150]) follow the tree-ORAM paradigm and harness PIR techniques to implement the retrieval phase efficiently. As outlined in §4.3.7, to privately retrieve the block indexed i, the client creates a PIR query, which is a *unit* vector $\mathbf{v}$ where all elements are set to zero, except the one at index i being set to 1. Such a query is either encrypted with HE or secret-shared. According to the PIR, the server computes a homomorphic inner product between $\mathbf{v}$ and the vector containing ORAM blocks on the retrieval path. So, if the adversary modifies ORAM blocks that will be likely multiplied with the ciphertext/share of the zero components in the retrieval vector, the final inner product will still be correct. In this case, the client is unable to tell whether the adversary has modified the ORAM structure, but the malicious server has learned if the vector component was zero or not, thus violating the privacy of the query. In the tree paradigm, the upper levels of the tree will likely contain blocks that have been recently accessed. By modifying data blocks in upper levels, the malicious server can learn whether the same blocks have been accessed again with high probability. To prevent this, Devadas *et al.* [54] suggested the client to download a large portion of data blocks, and apply the same homomorphic computation (like what server did) on them (as what should be done at the server), to verify if computation at the server is consistent with the one computed locally. This technique, however, incurs high bandwidth and computation overhead at the client.

4.5.2.2 Our Sub-Protocols

In this section, we design some sub-protocols for our MACAO framework. These sub-protocols offer security against the attacks targeting the inner product between the client queries and the ORAM data as presented above. We present the security proof of sub-protocols in the Appendix.

Inputs: Client has input α and S_i has inputs $(\langle\mathbf{U}\rangle_i, \langle\mathbf{V}\rangle_i, \langle\mathbf{U}\rangle_{i+1}, \langle\mathbf{V}\rangle_{i+1})$, where $\langle\mathbf{U}\rangle_i = (\llbracket\mathbf{U}\rrbracket_i, \llbracket\alpha\mathbf{U}\rrbracket_i)$ and so forth. Each S_i has $\llbracket r\rrbracket_i, \llbracket\hat{r}\rrbracket_i$ as the shares of two random values $r, \hat{r} \in \mathbb{F}_p$.

1. Every S_i locally computes $\mathbf{X}_i \leftarrow \llbracket\mathbf{U}\rrbracket_i \times \llbracket\mathbf{V}\rrbracket_i + \llbracket\mathbf{U}\rrbracket_i \times \llbracket\mathbf{V}\rrbracket_{i+1} + \llbracket\mathbf{U}\rrbracket_{i+1} \times \llbracket\mathbf{V}\rrbracket_i$ and $\mathbf{Y}_i \leftarrow \llbracket\mathbf{U}\rrbracket_i \times \llbracket\alpha\mathbf{V}\rrbracket_i + \llbracket\mathbf{U}\rrbracket_i \times \llbracket\alpha\mathbf{V}\rrbracket_{i+1} + \llbracket\mathbf{U}\rrbracket_{i+1} \times \llbracket\alpha\mathbf{V}\rrbracket_i$.
2. Every S_i represents $\mathbf{X}_i$ and $\mathbf{Y}_i$ as the sum of random matrices as $\mathbf{X}_i = \mathbf{R}_0^{(i)} + \mathbf{R}_1^{(i)} + \mathbf{R}_2^{(i)}$, where $\mathbf{R}_j^{(i)} \overset{\$}{\leftarrow} \mathbb{F}_p^{m\times k}$ and $\mathbf{Y}_i = \mathbf{T}_0^{(i)} + \mathbf{T}_1^{(i)} + \mathbf{T}_2^{(i)}$, where $\mathbf{T}_j^{(i)} \overset{\$}{\leftarrow} \mathbb{F}_p^{m\times k}$. S_i retains $(\mathbf{R}_i^{(i)}, \mathbf{R}_{i+1}^{(i)}, \mathbf{T}_i^{(i)}, \mathbf{T}_{i+1}^{(i)})$ and sends $(\mathbf{R}_{i-1}^{(i)}, \mathbf{R}_i^{(i)}, \mathbf{T}_{i-1}^{(i)}, \mathbf{T}_i^{(i)})$ to S_{i-1}, $(\mathbf{R}_i^{(i)}, \mathbf{R}_{i+1}^{(i)}, \mathbf{T}_i^{(i)}, \mathbf{T}_{i+1}^{(i)})$ to S_{i+1}.
3. Every S_i computes $\langle\mathbf{Q}\rangle_i = (\llbracket\mathbf{Q}\rrbracket_i, \llbracket\alpha\mathbf{Q}\rrbracket_i)$ and $\langle\mathbf{Q}\rangle_{i+1} = (\llbracket\mathbf{Q}\rrbracket_{i+1}, \llbracket\alpha\mathbf{Q}\rrbracket_{i+1})$, where $\llbracket\mathbf{Q}\rrbracket_i \leftarrow \mathbf{R}_i^{(0)} + \mathbf{R}_i^{(1)} + \mathbf{R}_i^{(2)}$, $\llbracket\mathbf{Q}\rrbracket_{i+1} \leftarrow \mathbf{R}_{i+1}^{(0)} + \mathbf{R}_{i+1}^{(1)} + \mathbf{R}_{i+1}^{(2)}$, $\llbracket\alpha\mathbf{Q}\rrbracket_i \leftarrow \mathbf{T}_i^{(0)} + \mathbf{T}_i^{(1)} + \mathbf{T}_i^{(2)}$, $\llbracket\alpha\mathbf{Q}\rrbracket_{i+1} \leftarrow \mathbf{T}_{i+1}^{(0)} + \mathbf{T}_{i+1}^{(1)} + \mathbf{T}_{i+1}^{(2)}$ and $\mathbf{Q} = \mathbf{U} \times \mathbf{V}$.

Output: Each S_i sends $\llbracket r\rrbracket_i, \llbracket\hat{r}\rrbracket_i$ to all other servers to compute $r, \hat{r}$ as $r \leftarrow \sum_i \llbracket r\rrbracket_i$, $\hat{r} \leftarrow \sum_i \llbracket\hat{r}\rrbracket_i$. All servers set $r_t \leftarrow r^t$ and $\hat{r}_t \leftarrow \hat{r}^t$ for $t = 1, \ldots, mp$. Every S_i computes and sends to the client $\llbracket x\rrbracket_i \leftarrow \sum_j \sum_k r_t \llbracket\mathbf{Q}[j,k]\rrbracket_i$, $\llbracket x\rrbracket_{i+1} \leftarrow \sum_j \sum_k \hat{r}_t \llbracket\mathbf{Q}[j,k]\rrbracket_{i+1}$, $\llbracket y\rrbracket_i \leftarrow \sum_j \sum_k r_t \llbracket\alpha\mathbf{Q}[j,k]\rrbracket_i$, $\llbracket y\rrbracket_{i+1} \leftarrow \sum_j \sum_k \hat{r}_t \llbracket\alpha\mathbf{Q}[j,k]\rrbracket_{i+1}$, where $t = jp + k$. If $(\llbracket x\rrbracket_i, \llbracket y\rrbracket_i)$ received from S_i and S_{i-1} are inconsistent or $\alpha x \neq y$, where $x \leftarrow \sum_i \llbracket x\rrbracket_i$ and $y \leftarrow \sum_i \llbracket y\rrbracket_i$, the client sends $\perp$ to all servers and aborts. Otherwise, the client sends ok and every S_i accepts $\langle\mathbf{Q}\rangle_i, \langle\mathbf{Q}\rangle_{i+1}$ as its correct authenticated shares of $\mathbf{U} \times \mathbf{V}$.

Figure 4.25: Authenticated matrix multiplication with RSS.

We construct matrix multiplication protocols using RSS and SPDZ schemes described in §4.3.6.2.

We first extend RSS to be secure against a malicious adversary in the three-server setting. The key idea is to create an information-theoretic MAC for each secret as proposed in [51]. Let u, v be two secrets to be multiplied. The dealer (*i.e.*, the client in our model) creates the authenticated shares of u and v, *i.e.*, $\langle u\rangle = (\llbracket u\rrbracket, \llbracket\alpha\rrbracket)$ and $\langle v\rangle = (\llbracket v\rrbracket, \llbracket\alpha v\rrbracket)$, and distributes them appropriately to servers accordingly. More specifically, every S_i receives $(\langle u\rangle_i, \langle u\rangle_{i+1}, \langle v\rangle_i, \langle v\rangle_{i+1})$. All servers jointly execute the RSS multiplication protocol to compute $\llbracket uv\rrbracket$ over $\llbracket u\rrbracket, \llbracket v\rrbracket$, and $\llbracket\alpha uv\rrbracket$ over $\llbracket\alpha u\rrbracket$, $\llbracket v\rrbracket$ (or $\llbracket\alpha v\rrbracket, \llbracket u\rrbracket$), resulting in $\langle uv\rangle = (\llbracket uv\rrbracket, \llbracket\alpha uv\rrbracket)$. To this end, every S_i sends $\langle uv\rangle_i$, $\langle uv\rangle_{i+1}$ to the client. The client executes $x \leftarrow \mathsf{AuthRecover}(\alpha, (\langle uv\rangle_0, \ldots, \langle uv\rangle_{\ell-1}))$ and aborts if $x = \perp$.

We now develop an authenticated matrix multiplication protocol based on the extended RSS. Given two matrices $\mathbf{U} \in \mathbb{F}_p^{m\times n}$ and $\mathbf{V} \in \mathbb{F}_p^{n\times p}$, $\mathbf{U} \times \mathbf{V}$ incurs $\mathcal{O}(mnp)$ number of pair-wise multiplications. Simply using RSS for each multiplication requires $\mathcal{O}(mnp)$ shares being sent from one

Initialize: The servers invoke the pre-computation to generate sufficient number of authenticated shares of matrix multiplication triples $(\langle \mathbf{A} \rangle, \langle \mathbf{B} \rangle, \langle \mathbf{C} \rangle)$.

Inputs: The client has input α and every S_i has inputs $(\llbracket \alpha \rrbracket_i, \langle \mathbf{U} \rangle_i, \langle \mathbf{V} \rangle_i)$. Each S_i has $\llbracket r \rrbracket_i, \llbracket \hat{r} \rrbracket_i$ as the shares of random values $r, \hat{r} \in \mathbb{F}_p$.

1. Every S_i locally computes $\llbracket \mathbf{E} \rrbracket_i \leftarrow \llbracket \mathbf{U} \rrbracket_i - \llbracket \mathbf{A} \rrbracket_i$, and $\llbracket \mathbf{P} \rrbracket_i \leftarrow \llbracket \mathbf{V} \rrbracket_i - \llbracket \mathbf{B} \rrbracket_i$ and broadcast $\llbracket \mathbf{E} \rrbracket_i$, $\llbracket \mathbf{P} \rrbracket_i$.
2. All servers open $\mathbf{E}$ and $\mathbf{P}$ by every server locally computing $\mathbf{E} \leftarrow \sum_i \llbracket \mathbf{E} \rrbracket_i$, $\mathbf{P} \leftarrow \sum_i \llbracket \mathbf{P} \rrbracket_i$.
3. Every S_i locally computes $\langle \mathbf{Q} \rangle_i = (\llbracket \mathbf{Q} \rrbracket_i, \llbracket \alpha \mathbf{Q} \rrbracket_i)$, where $\llbracket \mathbf{Q} \rrbracket_i \leftarrow \llbracket \mathbf{C} \rrbracket_i + \mathbf{E} \times \llbracket \mathbf{B} \rrbracket_i + \llbracket \mathbf{A} \rrbracket_i \times \mathbf{P} + \mathbf{E} \times \mathbf{P}$ and $\llbracket \alpha \mathbf{Q} \rrbracket_i \leftarrow \llbracket \alpha \mathbf{C} \rrbracket_i + \mathbf{E} \times \llbracket \sigma \mathbf{B} \rrbracket_i + \llbracket \sigma \mathbf{A} \rrbracket_i \times \mathbf{P} + \llbracket \alpha \rrbracket_i \mathbf{E} \times \mathbf{P}$, $\mathbf{P} = \mathbf{U} \times \mathbf{V}$.

Output: Each S_i sends $\llbracket r \rrbracket_i, \llbracket \hat{r} \rrbracket_i$ to all other servers to compute $r, \hat{r}$ as $r \leftarrow \sum_i \llbracket r \rrbracket_i$, $\hat{r} \leftarrow \sum_i \llbracket \hat{r} \rrbracket_i$. All servers set $r_t \leftarrow r^t$ and $\hat{r}_t \leftarrow \hat{r}^t$ for $t = 1, \ldots, mp$. Every S_i locally computes and sends to the client $\llbracket x \rrbracket_i \leftarrow \sum_j \sum_k (r_t \mathbf{E}[j,k] + \hat{r}_t \mathbf{P}[j,k])$ and $\llbracket y \rrbracket_i \leftarrow \sum_j \sum_k (r_t \llbracket \alpha \mathbf{E}[j,k] \rrbracket_i + \hat{r}_t \llbracket \alpha \mathbf{P}[j,k] \rrbracket_i)$, where $\llbracket \alpha \mathbf{E} \rrbracket_i = \llbracket \alpha \mathbf{U} \rrbracket_i - \llbracket \alpha \mathbf{A} \rrbracket_i$, $\llbracket \alpha \mathbf{P} \rrbracket_i = \llbracket \alpha \mathbf{V} \rrbracket_i - \llbracket \alpha \mathbf{B} \rrbracket_i$ and $t = jp + k$. The client computes $x \leftarrow \sum_i \llbracket x \rrbracket_i$, $y \leftarrow \sum_i \llbracket y \rrbracket_i$. If $\alpha x \neq y$, the client sends $\perp$ to all servers and aborts. Otherwise, the client sends ok and every S_i accepts $\langle \mathbf{Q} \rangle_i$ as its correct authenticated share of $\mathbf{U} \times \mathbf{V}$.

Figure 4.26: Authenticated matrix multiplication with SPDZ.

server to the other servers. Instead of doing so, we can perform the local matrix multiplication over the shares and then only re-share the computation result. This strategy saves a factor of n number of shares to be transferred among three servers. Let $\langle \mathbf{U} \rangle = (\llbracket \mathbf{U} \rrbracket, \llbracket \alpha \mathbf{U} \rrbracket)$ and $\langle \mathbf{V} \rangle = (\llbracket \mathbf{V} \rrbracket, \llbracket \alpha \mathbf{V} \rrbracket)$ be the authenticated share of $\mathbf{U}$ and $\mathbf{V}$, respectively. Figure 4.25 presents our matrix multiplication protocol with RSS scheme in the three-server setting with malicious security.

In our framework, we only employ RSS for the specific three-server setting, where no server can collude with each other (level 1 of privacy). Although a higher privacy level can be achieved with the general $(\ell - t)$-threshold RSS, where $\ell = 2t + 1$ is the number of servers and t is the privacy level, it requires $\binom{\ell}{t}$ shares for each secret, $\binom{\ell-1}{t}$ of which are stored in each server. This significantly increases the server storage, I/O access, communication and computation overhead, and therefore it is not desirable. In the following, we construct another matrix multiplication protocol, which is more suitable for applications that need a high privacy level.

Inspired by [133], we develop an efficient authenticated matrix multiplication protocol with SPDZ sharing. As discussed previously, the computation of $\mathbf{U} \times \mathbf{V}$, where $\mathbf{U} \in \mathbb{F}_p^{m \times n}, \mathbf{V} \in \mathbb{F}_p^{n \times p}$

incurs $\mathcal{O}(mnp)$ numbers of multiplication. Simply using the original SPDZ multiplication protocol [51] for the matrix multiplication will increase the communication overhead among the servers significantly[10]. To save a factor of n the bandwidth overhead, instead of using the standard Beaver triples of the form (a, b, ab), we generate matrix multiplication triples $(\mathbf{A}, \mathbf{B}, \mathbf{C})$, where $\mathbf{C} = \mathbf{A} \times \mathbf{B}$. Now we assume that each server S_i stores an authenticated share of the triple as $(\langle \mathbf{A} \rangle_i, \langle \mathbf{B} \rangle_i, \langle \mathbf{C} \rangle_i)$. Figure 4.26 describes the matrix multiplication protocol by SPDZ. Note that in our setting, the client owns the global MAC key α so they they can check the integrity of computation at the end of the protocol. In other words, servers do not need to multiply their share of the MAC key with the random linear combination of opened values.

We construct several PIR protocols in the shared setting with malicious security that will be used in our ORAM framework. In contrast to standard the PIR, where the database is public, the database in our setting is shared with authenticated secret sharing in Figure 4.5 and each server in the system stores one or multiple shares of the database. In this setting, each database item is split into m chunks as $\mathbf{b}_i = (b_{i,1}, \ldots, b_{i,m})$, where $b_{ij} \in \mathbb{F}_p$. So a database with N items can be interpreted as a $m \times N$ matrix $\mathbf{B} = (\mathbf{b}_i) \in \mathbb{F}_p^{m \times N}$. Let $\langle \mathbf{B} \rangle = (\llbracket \mathbf{B} \rrbracket, \llbracket \alpha \mathbf{B} \rrbracket) = \Big((\llbracket \mathbf{b}_i \rrbracket), (\llbracket \alpha \mathbf{b}_i \rrbracket) \Big)$ be the authenticated share of $\mathbf{B}$. Our PIR protocols are as follows.

We extend the original XOR-PIR protocol [45] to privately retrieve a block in the shared setting with malicious security. We consider the three-server case; however it can be extended to the general ℓ-server setting. In this setting, the client creates three authenticated shares for database $\mathbf{B}$ as $(\langle \mathbf{B} \rangle_0, \ldots, \langle \mathbf{B} \rangle_2) \leftarrow \mathsf{AuthCreate}(\alpha, \mathbf{B}, 3)$. Each S_i stores two out of three authenticated shares as $(\langle \mathbf{B} \rangle_i, \langle \mathbf{B} \rangle_{i+1})$. Our main idea is to harness the XOR-PIR protocol in [45] to retrieve each

[10] The authors in [18] observed that the offline phase can be optimized for functions. The approach is however, different. We are not aware of other published results that optimized the offline phase for matrix multiplication.

Parameters: N denotes the number of shared items in the shared database.

Inputs: The client has inputs (idx, α) and each server S_i has inputs $(\langle \mathbf{B}\rangle_i, \langle \mathbf{B}\rangle_{i+1})$.

1. Let $(q_0, \dots, q_{N-1})$ be the indicator for the retrieved block, *i.e.*, $q_{\mathsf{idx}} = 1$ and $q_i = 0$ for $0 \le i \ne \mathsf{idx} < N$. For each authenticated shared database $\langle \mathbf{B}\rangle_i$, stored on S_i and S_{i-1}:
   (a) The client generates a random binary string of length N, $R^{(i)} = (r_0^{(1)}, \dots, r_{N-1}^{(1)})$. The client generates $R^{(i+1)} = (r_0^{(2)}, \dots, r_{N-1}^{(i+1)})$ by flipping the idx-th bit of $R^{(i)}$, *i.e.*, $r_{\mathsf{idx}}^{(i+1)} = \bar{r}_{\mathsf{idx}}^{(i)}$ and $r_j^{(i+1)} = r_j^{(i)}$ for all $j \ne \mathsf{idx}$. The client sends $R^{(i)}$ to S_i and $R^{(i+1)}$ to S_{i+1}.
   (b) S_i computes and responds with $\mathbf{x}_i = \sum_j ([\![\mathbf{b}_j]\!]_i \cdot r_j^{(i)})$ and $\mathbf{y}_i = \sum_j [\![\alpha\mathbf{b}_j]\!]_i \cdot r_j^{(i)}$, where $\sum$ represents the bit-wise XOR and $\cdot$ represents the bit-wise AND. Similarly, S_{i+1} computes and responds with $\mathbf{x}_{i+1} = \sum_j ([\![\mathbf{b}_j]\!]_i \cdot r_j^{(i+1)})$ and $\mathbf{y}_i = \sum_j [\![\alpha\mathbf{b}_j]\!]_i \cdot r_j^{(i+1)}$.
   (c) The client computes $\mathbf{x}'_i = \mathbf{x}_i \oplus \mathbf{x}_{i+1}$ and $\mathbf{y}'_i = \mathbf{y}_i \oplus \mathbf{y}_{i+1}$.
2. The client interprets $\mathbf{x}'_i$ and $\mathbf{y}'_i$ as the i-th authenticated share of the retrieved block $\mathbf{b}$, *i.e.*, $\langle \mathbf{b}\rangle_i = ([\![\mathbf{b}]\!]_i, [\![\alpha\mathbf{b}]\!]_i) = (\mathbf{x}'_i, \mathbf{y}'_i)$. The client executes $\mathbf{b} \leftarrow \mathsf{AuthRecover}(\alpha, (\langle \mathbf{b}\rangle_0, \dots, \langle \mathbf{b}\rangle_2))$.

Output: The client outputs $\mathbf{b}$.

Figure 4.27: Authenticated XOR-PIR on additively shared database.

Parameters: N denotes the number of shared items in the shared database.

Inputs: The client has inputs (idx, α) and each server S_i has inputs $(\langle \mathbf{B}\rangle_i, \langle \mathbf{B}\rangle_{i+1},)$.

1. The client creates an indicator $\mathbf{Q} = (q_0, \dots, q_{N-1})$ for the block to be retrieved, *i.e.*, $q_{\mathsf{idx}} = 1$ and $q_j = 0$ for $0 \le j \ne \mathsf{idx} < N$. The client creates shares of $\mathbf{Q}$ by executing Create algorithm on each q_i, resulting in $[\![\mathbf{Q}]\!]_0, \dots, [\![\mathbf{Q}]\!]_{\ell-1}$. The client sends $[\![\mathbf{Q}]\!]_i$ to S_i for $0 \le i \le 2$.
2. Every S_i executes step 1 of the matrix multiplication protocol based on RSS scheme in Figure 4.25. Specifically, every S_i locally computes and responds with $\mathbf{x}_i = [\![\mathbf{Q}]\!]_i \times [\![\mathbf{B}]\!]_i + [\![\mathbf{Q}]\!]_i \times [\![\mathbf{B}]\!]_{i+1} + [\![\mathbf{Q}]\!]_{i+1} \times [\![\mathbf{B}]\!]_i$ and $\mathbf{y}_i = [\![\mathbf{Q}]\!]_i \times [\![\alpha\mathbf{B}]\!]_i + [\![\mathbf{Q}]\!]_i \times [\![\alpha\mathbf{B}]\!]_{i+1} + [\![\mathbf{Q}]\!]_{i+1} \times [\![\alpha\mathbf{B}]\!]_i$.
3. The client computes $\mathbf{b} \leftarrow \sum_i \mathbf{x}_i$ and $\mathbf{t} \leftarrow \sum \mathbf{y}_i$.

Output: If $\alpha\mathbf{b} = \mathbf{t}$, the client outputs $\mathbf{b}$ as the correct block. Otherwise, the client outputs $\perp$.

Figure 4.28: Authenticated RSS-PIR on additively shared database.

authenticated share of the database block, and then verify the integrity of the block from the shares. Figure 4.27 presents our protocol in details.

We construct a PIR protocol with malicious security based on the RSS matrix multiplication protocol presented in §4.3.6.2. Similar to XOR-PIR, we consider the 3-server setting, where each server S_i stores two authenticated shares $(\langle \mathbf{B}\rangle_i, \langle \mathbf{B}\rangle_{i+1})$ of database $\mathbf{B}$. Figure 4.28 presents the protocol in detail.

Parameters: N denotes the number of shared items in the shared database.

Inputs: The client has inputs (idx, α) and each server S_i has input $\langle \mathbf{B} \rangle_i$.

1. The client creates an indicator $\mathbf{Q} = (q_0, \ldots, q_{N-1})$ for the block to be retrieved, *i.e.*, $q_{\mathsf{idx}} = 1$ and $q_j = 0$ for $0 \leq j \neq \mathsf{idx} < N$. The client creates the authenticated shares of $\mathbf{Q}$ by executing $\mathsf{AuthCreate}$ algorithm on each q_i, resulting in $\langle \mathbf{Q} \rangle_0, \ldots, \langle \mathbf{Q} \rangle_{\ell-1}$. The client sends $\langle \mathbf{Q} \rangle_i$ to S_i
2. The client and all servers jointly execute the SPDZ-based matrix multiplication protocol in Figure 4.26 to compute $\langle \mathbf{Q} \times \mathbf{B} \rangle$. If the client does not abort, every S_i sends the output $\langle \mathbf{Q} \times \mathbf{B} \rangle_i$ to the client.
3. The client executes $\mathbf{b} \leftarrow \mathsf{AuthRecover}(\alpha, \langle \mathbf{Q} \times \mathbf{B} \rangle_0, \ldots, \langle \mathbf{Q} \times \mathbf{B} \rangle_{\ell-1})$.

Output: The client outputs $\mathbf{b}$ as the retrieved block.

Figure 4.29: Authenticated SPDZ-PIR on additively shared database.

We construct a maliciously-secure PIR protocol based on the SPDZ matrix multiplication (Figure 4.29). This protocol works in the general ℓ-server setting, in which every S_i stores a single authenticated share of the database as $\langle \mathbf{B} \rangle_i$.

As outlined in §4.3.3, the core idea of the eviction in [179] is to maximize the number of data blocks that can be pushed down in a single block scan on the eviction path via strategic pick and drop operations. To make these operations oblivious, the client needs to download and upload the entire bucket for each level scan, thereby suffering from the logarithmic communication overhead. Similar to $\mathsf{S^3ORAM}$, MACAO framework harnesses the permutation matrix concept to implement the Circuit-ORAM eviction strategy in a more communication-efficient manner. Figure 4.30 presents our algorithm with the highlights as follows. For each level h on the eviction path v, we create a $(Z+1) \times (Z+1)$ matrix $\mathbf{I}_h$ initialized with zeros. We consider the data to be computed with $\mathbf{I}_h$ as a matrix $\mathbf{U}_h \in \mathbb{F}_p^{(Z+1) \times m}$ containing Z blocks from the bucket at level h and the block supposed to be held by the client[11]. The main objective is to perform $\mathbf{V}_h^\top = \mathbf{U}_h^\top \times \mathbf{I}_h$, where $\mathbf{V}_h \in \mathbb{F}_p^{(Z+1) \times m}$ contains the new data for the bucket at level h and the new block to be picked for deeper levels. Therefore, to drop the holding block to an empty slot indexed x in the bucket at level h, the client sets $\mathbf{I}_h[Z, x] \leftarrow 1$ (line 12). To pick a block at slot x', the client sets $\mathbf{I}_h[x', Z] \leftarrow 1$ (line 17). To skip

[11] Remark that each database block is split into m chunks $c_i \in \mathbb{F}_p$.

```
$(\mathbf{b}, (\mathbf{I}_0, \ldots, \mathbf{I}_H)) \leftarrow \mathsf{CreatePermMat}(v)$:
1: $\mathsf{hold} \leftarrow \perp, \mathsf{dest} \leftarrow \perp$, $b \leftarrow \{0\}^{|b|}$
2: $(\mathsf{deepest}, \mathsf{deepestIdx}) \leftarrow \mathsf{PrepareDeepest}(v)$
3: $\mathsf{target} \leftarrow \mathsf{PrepareTarget}(v)$
4: if $\mathsf{target}[0] \neq \perp$ then
5:     $\mathsf{hold} \leftarrow \mathsf{deepestIdx}[0]$; $\mathsf{dest} \leftarrow \mathsf{target}[0]$
6:     $b \leftarrow \mathbf{S}[\mathsf{hold}]$, $\mathbf{S}[\mathsf{hold}] \leftarrow \{\}$
7: for $h = 0$ to $H$ do
8:     $\mathbf{I}_h[i, j] \leftarrow 0$ for $0 \leq i \leq Z$, $0 \leq j \leq Z$
9:     if $\mathsf{hold} \neq \perp$ then
10:        if $h = \mathsf{dest}$ then # Drop the on-hold block to this level
11:            $\mathsf{hold} \leftarrow \perp$, $\mathsf{dest} \leftarrow \perp$,
12:            $\mathbf{I}_h[Z][x] \leftarrow 1$ where $x_h$ is an empty slot index at level $h$
13:        else# Move the on-hold block to the next level
14:            $\mathbf{I}_h[Z][Z] \leftarrow 1$
15:    if $\mathsf{target}[h] \neq \perp$ then # Pick a block at this level
16:        $\mathbf{I}_h[x'][Z] \leftarrow 1$ where $x' \leftarrow \mathsf{deepestIdx}[h]$
17:        $\mathsf{hold} \leftarrow x'$, $\mathsf{dest} \leftarrow \mathsf{target}[h]$
18:    for each real block id at level $h$ do # Preserve existing blocks
19:        if $\mathsf{pm}[\mathsf{id}].\mathsf{pIdx} \mod Z \neq \mathsf{deepestIdx}[h]$ then
20:            $\mathbf{I}_h[\hat{x}][\hat{x}] \leftarrow 1$ where $\hat{x} \leftarrow \mathsf{pm}[\mathsf{id}].\mathsf{pIdx} \mod Z$
21: return $(\mathbf{b}, (\mathbf{I}_0, \ldots, \mathbf{I}_H))$
```

Figure 4.30: Permutation matrix for Circuit-ORAM eviction.

this level (no drop or pick), the client sets $\mathbf{I}_h[Z, Z] \leftarrow 1$, which moves the holding block to the next level (line 14). To preserve an existing block at this level, the client sets $\mathbf{I}_h[\hat{x}, \hat{x}] \leftarrow 1$ where $\hat{x}$ is the slot index of the block in the bucket (lines 18-20).

4.5.2.3 MACAO Schemes

In this section, we construct two ORAM schemes in our framework called $\mathsf{MACAO}_{\mathsf{rss}}$ and $\mathsf{MACAO}_{\mathsf{spdz}}$ by putting sub-protocols in the previous section altogether. Our constructions follow the general tree-ORAM access structure [161] outlined in §4.3.1, which contains two main subroutines including retrieval and eviction. At the high level idea, our ORAM schemes use multi-server authenticated PIR protocols to implement the retrieval. For the eviction, our ORAM schemes harness the concept of permutation matrix and the homomorphic matrix multiplication protocols.

```
$\mathsf{data}' \leftarrow \mathsf{Access}(\mathsf{op}, \mathsf{id}, \mathsf{data})$:
(pid, pIdx) ← pm[id]
data′ ← Retrieve(pid, pIdx)
if data′ = ⊥ then
    return abort
if op = write then
    $b \leftarrow$ data
pm[id].pid $\xleftarrow{\$} \{0, \ldots, 2^H - 1\}$
Evict()
return data′
```

Figure 4.31: MACAO access structure.

We first give the storage layout at the client- and server-side, and then present our ORAM schemes in detail.

Our constructions follow the tree-ORAM paradigm outlined in §4.3.1. In $\mathsf{MACAO}_{\mathsf{rss}}$ scheme, there are three servers and each server $S_i \in \{S_0, S_1, S_2\}$ stores two authenticated shares of the tree as $(\langle \mathbf{T} \rangle_i, \langle \mathbf{T} \rangle_{i+1})$. In $\mathsf{MACAO}_{\mathsf{spdz}}$ scheme, there are $\ell \geq 2$ servers, where each $S_i \in \{S_0, \ldots, S_{\ell-1}\}$ stores an authenticated share $\langle \mathbf{T} \rangle_i$ and an additive share of the global MAC key, $[\![\alpha]\!]_i$.

The client maintains the position map of the form $\mathsf{pm} := (\mathsf{id}; \mathsf{pid}, \mathsf{pIdx})$, where id is the block ID, $0 \leq \mathsf{pid} < 2^H$ is the path ID and $0 \leq \mathsf{pIdx} < ZH$ is the index of the block in its assigned path. Notice that pm can be stored remotely at the servers by recursive ORAM [169] and meta-data construction as discussed in $\mathsf{S^3ORAM}$ in §4.4 (see §4.5.2.6). For ease of presentation, we assume that pm is locally stored. Since we follow Circuit-ORAM eviction, the client needs to maintain the stash component $(\mathbf{S})$ to temporarily store blocks that cannot be evicted back to the tree. This stash can also be stored at the server-side for reduced storage overhead as will be discussed in the next section. Finally, the client stores the global MAC key α.

Figure 4.31 presents the general access structure of MACAO schemes. In the following, we present in detail the retrieval and eviction phases of each scheme.

$b \leftarrow \mathsf{MACAO_{rss}.Retrieve}(\mathsf{pid}, \mathsf{pIdx})$:

1. **return b**, where $\mathbf{b} \leftarrow$ The client executes either XOR-PIR protocol in Figure 4.27 or RSSS-PIR protocol in Figure 4.28 with 3 servers, in which the client has input (pIdx, α) and each S_i has inputs as the path pid of $\langle \mathbf{T} \rangle_i$, $\langle \mathbf{T} \rangle_{i+1}$.

Figure 4.32: $\mathsf{MACAO_{rss}}$ retrieval.

$\mathsf{MACAO_{rss}.Evict}()$:

Parameters: $\mathsf{EvictCtr}$ denotes the number of eviction operations initialized at 0, H denotes the height of ORAM tree, $\ell = 3$ denotes the number of servers in the system.

Inputs: The client has input α and every S_i has inputs $(\langle \mathbf{T} \rangle_i, \langle \mathbf{T} \rangle_{i+1})$.

Client:

1. $v \leftarrow \mathsf{DigitReverse}_2(\mathsf{EvictCtr} \mod 2^H)$, $\mathsf{EvictCtr} \leftarrow \mathsf{EvictCtr} + 1$
2. $(\mathbf{b}, (\mathbf{I}_0, \ldots, \mathbf{I}_H)) \leftarrow \mathsf{CreatePermMat}(v)$
3. $(\langle \mathbf{b} \rangle_0, \ldots, \langle \mathbf{b} \rangle_{\ell-1}) \leftarrow \mathsf{AuthCreate}(\alpha, \mathbf{b}, \ell)$
4. $([\![\mathbf{I}_h]\!]_0, \ldots, [\![\mathbf{I}_h]\!]_{\ell-1}) \leftarrow \mathsf{Create}(\mathbf{I}_h, \ell)$ for $0 \leq h \leq H$
5. Send $(\langle b \rangle_i, ([\![\mathbf{I}_0]\!]_i, \ldots, [\![\mathbf{I}_H]\!]_i))$ to S_i and S_{i-1} for $0 \leq i < \ell$

Server: For each level h of the eviction path v starting from the root:

6. Every S_i forms authenticated shared matrices $\langle \mathbf{B}_h \rangle_i$, $\langle \mathbf{B}_h \rangle_{i+1}$ by concatenating the bucket $\mathcal{P}(v, h)$ of $\langle \mathbf{T} \rangle_i$, $\langle \mathbf{T} \rangle_{i+1}$ with $\langle \mathbf{b} \rangle_i$, $\langle \mathbf{b} \rangle_{i+1}$, respectively.
7. All parties execute RSS-based matrix multiplication protocol in Figure 4.25, where the client inputs α and every S_i inputs $([\![\mathbf{I}_h]\!]_i, \langle \mathbf{B}_h \rangle_i, [\![\mathbf{I}_h]\!]_{i+1}, \langle \mathbf{B}_h \rangle_{i+1})$. Let $\langle \mathbf{B}_h \times \mathbf{I}_h \rangle_i, \langle \mathbf{B}_h \times \mathbf{I}_h \rangle_{i+1}$ be the output of S_i.
8. Every S_i interprets the last row of $\langle \mathbf{B}_h \times \mathbf{I}_h \rangle_i, \langle \mathbf{B}_h \times \mathbf{I}_h \rangle_{i+1}$ as the on-hold blocks $\langle \mathbf{b} \rangle_i, \langle \mathbf{b} \rangle_{i+1}$ respectively, for the next level $h+1$, and updates the bucket $\mathcal{P}(v, h)$ of $\langle \mathbf{T} \rangle_i, \langle \mathbf{T} \rangle_{i+1}$ with the other rows.

Figure 4.33: $\mathsf{MACAO_{rss}}$ eviction.

Our $\mathsf{MACAO_{rss}}$ scheme operates on the specific three-server setting. $\mathsf{MACAO_{rss}}$ employs either XOR-PIR or RSS-PIR protocol for oblivious retrieval, and RSS matrix multiplication for eviction as follows.

Figure 4.32 presents the retrieval phase of $\mathsf{MACAO_{rss}}$ scheme. Since we perform the PIR on the retrieval path, all the buckets on the path are interpreted as the database input of the PIR protocols. As a result, the length of the PIR query is $Z(H+1)$ and the database input is interpreted as a matrix of size $Z(H+1) \times m$. Notice that we can use XOR-PIR and RSS-PIR protocols interchangeably in this phase. The difference is that XOR-PIR incurs less computation than RSS-PIR (XOR *vs.* arithmetic operations) with the smaller size of the client queries (binary string *vs.* finite field vector). As a trade-off, it doubles the number of data to be downloaded, and the computed blocks on the servers are in the form of XOR shares. This XOR-share format does not allow for further (homomorphic) arithmetic computations after being accessed once.

Figure 4.33 presents the eviction protocol of $\mathsf{MACAO}_{\mathsf{rss}}$ scheme in detail. We follow the deterministic eviction strategy proposed in [71], where the eviction path is selected according to the reverse-lexicographical order of the number of evictions being performed so far (line 1). Intuitively, the client creates the permutation matrices for Circuit-ORAM eviction plans (line 2). The client creates authenticated shares for the block fetched from the stash and the permutation matrices (lines 3, 4), and then distributes the shares to corresponding servers. Notice that it is not necessary to create authenticated shares of permutation matrices because RSS scheme only needs one authenticated share to do the authenticated homomorphic multiplication. Once receiving all shares from the client, all servers execute the RSS-based authenticated matrix multiplication protocol in Figure 4.25 for each level h of the eviction path. The servers form the authenticated shared matrices to be multiplied with the shared permutation matrices by concatenating the buckets at level h of the authenticated shared ORAM trees with the authenticated shared blocks sent by the client. The servers interpret the last row of the resulting matrices as the authenticated shared blocks holding by the client for the next level $(h + 1)$, and update the h-leveled buckets of the shared ORAM tree with the other rows of the resulting matrices.

Our $\mathsf{MACAO}_{\mathsf{spdz}}$ scheme operates on the general ℓ-server setting with the pre-computation model.

As presented in Figure 4.34, $\mathsf{MACAO}_{\mathsf{spdz}}$ employs the SPDZ-PIR protocol, instead of RSS-PIR or XOR-PIR in $\mathsf{MACAO}_{\mathsf{rss}}$, to implement the private retrieval phase.

```
$b \leftarrow \mathsf{MACAO_{rss}.Retrieve(pid, pIdx)}$:
1. return $\mathbf{b}$, where $\mathbf{b} \leftarrow$ The client executes SPDZ-PIR in Figure 4.29 with $\ell$ servers, in which the client has
   input $(\mathsf{pIdx}, \alpha)$ and each $S_i$ has inputs as the path $\mathsf{pid}$ of $\langle \mathbf{T} \rangle_i$.
```

Figure 4.34: $\mathsf{MACAO_{spdz}}$ retrieval.

```
$\mathsf{MACAO_{spdz}.Evict()}$:
Parameters: Same as Figure 4.33, except $\ell \geq 2$.
Inputs: The client has input $\alpha$ and every $S_i$ has inputs $(\llbracket \alpha \rrbracket_i, \langle \mathbf{T} \rangle_i)$.
Client:  $(\mathbf{b}, (\mathbf{I}_0, \ldots, \mathbf{I}_H)) \leftarrow$ Execute lines 1–3 in Figure 4.33.
1. $(\langle \mathbf{I}_h \rangle_0, \ldots, \langle \mathbf{I}_h \rangle_{\ell-1}) \leftarrow \mathsf{AuthCreate}(\mathbf{I}_h, \ell)$ for $0 \leq h \leq H$
2. Send $(\langle b \rangle_i, (\langle \mathbf{I}_0 \rangle_i, \ldots, \langle \mathbf{I}_H \rangle_i))$ to $S_i$ for $0 \leq i < \ell$
Server: For each level $h$ of the eviction path $v$ starting from the root:
3. Every $S_i$ forms authenticated shared matrices $\langle \mathbf{B}_h \rangle_i$ by concatenating the bucket $\mathcal{P}(v, h)$ of $\langle \mathbf{T} \rangle_i$ with $\langle \mathbf{b} \rangle_i$,
   respectively.
4. All parties execute the SPDZ-based matrix multiplication protocol in Figure 4.26, where the client inputs $\alpha$
   and every $S_i$ has inputs $(\llbracket \alpha \rrbracket_i, \langle \mathbf{I}_h \rangle_i, \langle \mathbf{B}_h \rangle_i)$. Let $\langle \mathbf{B}_h \times \mathbf{I}_h \rangle_i$ be the output of $S_i$.
5. Every $S$ interprets the last row of $\langle \mathbf{B}_h \times \mathbf{I}_h \rangle_i$ as the on-hold block $(\langle \mathbf{b} \rangle_i$ for the next level $h + 1$, and updates
   the buckets $\mathcal{P}(v, h)$ of $\langle \mathbf{T} \rangle_i$ with the other rows.
```

Figure 4.35: $\mathsf{MACAO_{spdz}}$ eviction.

Figure 4.35 depicts the eviction phase of $\mathsf{MACAO_{spdz}}$, which is similar to that of $\mathsf{MACAO_{rss}}$, except that it employs matrix multiplication protocol by SPDZ to implement the oblivious pick and drop operations. In this case, the client creates and sends to the servers the *authenticated shares* of the permutation matrices, instead of only the additive shares as in $\mathsf{MACAO_{rss}}$.

4.5.2.4 Security Analysis

We now present the security of MACAO schemes. The MACAO eviction follows the push-down principle proposed in Circuit-ORAM [179] so that it has the same overflow probability as follows.

Lemma 6 (Stash Overflow Probability). Let the bucket size $Z \geq 2$. Let $\mathsf{st}(\mathsf{MACAO[s]})$ be a random variable denoting the stash size of a MACAO scheme after an access sequence s. Then, for any access sequence s, $\Pr[\mathsf{st}(\mathsf{MACAO}[s]) \geq R] \leq 14 \cdot e^{-R}$.

Proof. (see [179]) □

We define the security model for the matrix multiplication with verifiability as follows.

Definition 10 (Matrix Multiplication with Verifiability). We first define the ideal world and real world as follows.

Let $\mathcal{F}_{\mathsf{mult}}$ be an ideal functionality, which performs the matrix multiplication for each client request as follows. In each time step, the environment $\mathcal{Z}$ specifies two matrices $\mathbf{X}$ and $\mathbf{Y}$ as the client's input. The client sends $\mathbf{X}$ and $\mathbf{Y}$ to $\mathcal{F}_{\mathsf{mult}}$. $\mathcal{F}_{\mathsf{mult}}$ notifies the simulator $\mathcal{S}_{\mathsf{mult}}$ (without revealing $\mathbf{X}$ and $\mathbf{Y}$ to $\mathcal{S}_{\mathsf{mult}}$). If $\mathcal{S}_{\mathsf{mult}}$ returns ok to $\mathcal{F}_{\mathsf{mult}}$, $\mathcal{F}_{\mathsf{mult}}$ computes and sends $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$ to the client. The client then returns $\mathbf{Z}$ to the environment $\mathcal{Z}$. If $\mathcal{S}_{\mathsf{mult}}$ returns abort to $\mathcal{F}_{\mathsf{mult}}$, $\mathcal{F}_{\mathsf{mult}}$ returns $\perp$ to the client.

In the real world, $\mathcal{Z}$ specifies an input $(\mathbf{X}, \mathbf{Y})$ to the client. The client executes the matrix multiplication protocol Π with servers $(S_0, \ldots, S_{\ell-1})$. The environment $\mathcal{Z}$ gets the view of the adversary $\mathcal{A}$ after every operation. The client outputs to the environment $\mathcal{Z}$ the output of the protocol Π or abort.

We say that a protocol $\Pi_{\mathcal{F}}$ securely realizes the ideal functionality $\mathcal{F}_{\mathsf{mult}}$ in the presence of a malicious adversary corrupting t servers iff for any PPT real-world adversary that corrupts up to t servers, there exists a simulator $\mathcal{S}_{\mathsf{mult}}$, such that for all non-uniform, polynomial-time environment $\mathcal{Z}$, there exists a negligible function negl such that

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F}_{\mathsf{mult}},\mathcal{S}_{\mathsf{mult}},\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda).$$

Lemma 7. In the 3-server setting, the matrix multiplication protocol via RSS in Figure 4.25 is secure against a malicious adversary corrupting an arbitrary server.

Proof. We prove by constructing a simulator such that the environment $\mathcal{Z}$ cannot distinguish between the real protocol and the ideal functionality. We define the simulator $\mathcal{S}_{\mathsf{mult}}$ in the ideal world and a sequence of hybrid games as follows.

The simulator follows the honest procedure on behalf of the client to multiply two dummy matrices $\mathbf{X} \in \mathbb{F}_p^{n\times m}$ and $\mathbf{Y} \in \mathbb{F}_p^{m\times p}$. During the multiplication, if the client (executed by the simulator) aborts then the simulator sends abort to $\mathcal{F}_{\mathsf{mult}}$ and stops. Otherwise, the simulator returns ok to $\mathcal{F}_{\mathsf{mult}}$ (causing it to output the result to the client).

We define a sequence of hybrid games to show that the following real world and the simulation in the ideal world are statistically indistinguishable:

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F}_{\mathsf{mult}},\mathcal{S}_{\mathsf{mult}},\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda).$$

We define Game 0 as the real game $\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda)$ with an environment $\mathcal{Z}$ and three servers in the presence of an adversary $\mathcal{A}$ presented in Definition 10. In this case, the real matrix multiplication protocol $\Pi_{\mathcal{F}}$ is the one presented in Figure 4.25. Without loss of generality, we assume server S_0 is corrupted.

We define Game 1 as follows. In this game, the client locally computes $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$. Whenever the client executes the protocol $\Pi_{\mathcal{F}}$ with three servers, if abort does not occur, the client uses their locally computed $\mathbf{Z}$ for further processing. The difference between Game 0 and Game 1 happens if at some point, where the client obtains an incorrect computation from the servers, but unable to detect because the adversary generates a valid MAC of the computation (thus the abort does not occur). We claim that Game 0 and Game 1 are statistically indistinguishable. The intuition is to show that if the adversarial server ever cheats by modifying the protocol input during the computation, it will be caught with high probability (thereby forcing the adversary to follow the protocol faithfully).

Let $\langle \mathbf{X} \rangle_i = (\llbracket \mathbf{X} \rrbracket_i, \llbracket \alpha\mathbf{X} \rrbracket_i)$ and $\langle \mathbf{X} \rangle_i = (\llbracket \mathbf{Y} \rrbracket_i, \llbracket \alpha\mathbf{Y} \rrbracket_i)$ be the authenticated shares of $\mathbf{X}$ and $\mathbf{Y}$ for every server S_i, $0 \le i \le 2$. Due to additive secret sharing, we have that $\mathbf{X} = \sum_i \llbracket \mathbf{X} \rrbracket_i$, $\mathbf{Y} = \sum_i \llbracket \mathbf{Y} \rrbracket_i$, $\alpha\mathbf{X} = \sum_i \llbracket \alpha\mathbf{X} \rrbracket_i$ and $\alpha\mathbf{Y} = \sum_i \llbracket \alpha\mathbf{Y} \rrbracket_i$. By replicated secret sharing, $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$ can be expressed as

$$
\begin{aligned}
\mathbf{Z} =& (\llbracket \mathbf{X} \rrbracket_0 + \llbracket \mathbf{X} \rrbracket_1 + \llbracket \mathbf{X} \rrbracket_2) \times (\llbracket \mathbf{Y} \rrbracket_0 + \llbracket \mathbf{Y} \rrbracket_1 + \llbracket \mathbf{Y} \rrbracket_2) \\
=& (\llbracket \mathbf{X} \rrbracket_0 \times \llbracket \mathbf{Y} \rrbracket_0 + \llbracket \mathbf{X} \rrbracket_0 \times \llbracket \mathbf{Y} \rrbracket_1 + \llbracket \mathbf{X} \rrbracket_1 \times \llbracket \mathbf{Y} \rrbracket_0)_{S_0} + \\
& (\llbracket \mathbf{X} \rrbracket_1 \times \llbracket \mathbf{Y} \rrbracket_1 + \llbracket \mathbf{X} \rrbracket_1 \times \llbracket \mathbf{Y} \rrbracket_2 + \llbracket \mathbf{X} \rrbracket_2 \times \llbracket \mathbf{Y} \rrbracket_1)_{S_1} + \qquad (4.9) \\
& (\llbracket \mathbf{X} \rrbracket_2 \times \llbracket \mathbf{Y} \rrbracket_2 + \llbracket \mathbf{X} \rrbracket_0 \times \llbracket \mathbf{Y} \rrbracket_2 + \llbracket \mathbf{X} \rrbracket_2 \times \llbracket \mathbf{Y} \rrbracket_0)_{S_2} \\
& (\mathbf{R}_0^{(0)} + \mathbf{R}_1^{(0)} + \mathbf{R}_2^{(0)})_{S_0} + (\mathbf{R}_0^{(1)} + \mathbf{R}_1^{(1)} + \mathbf{R}_2^{(1)})_{S_1} + (\mathbf{R}_0^{(2)} + \mathbf{R}_1^{(2)} + \mathbf{R}_2^{(2)})_{S_2} \qquad (4.10) \\
=& \left[\begin{array}{l}
(\mathbf{R}_0^{(0)} + \mathbf{R}_0^{(1)} + \mathbf{R}_0^{(2)})_{S_0} + (\mathbf{R}_1^{(0)} + \mathbf{R}_1^{(1)} + \mathbf{R}_1^{(2)})_{S_1} + (\mathbf{R}_2^{(0)} + \mathbf{R}_2^{(1)} + \mathbf{R}_2^{(2)})_{S_2} \\
(\mathbf{R}_1^{(0)} + \mathbf{R}_1^{(1)} + \mathbf{R}_1^{(2)})_{S_0} + (\mathbf{R}_2^{(0)} + \mathbf{R}_2^{(1)} + \mathbf{R}_2^{(2)})_{S_1} + (\mathbf{R}_0^{(0)} + \mathbf{R}_0^{(1)} + \mathbf{R}_0^{(2)})_{S_2}
\end{array} \right. \qquad (4.11)
\end{aligned}
$$

S_0 can cheat at three stages: (*i*) before the re-sharing phase where S_0 modifies their own shares $(\llbracket \mathbf{X} \rrbracket_0, \llbracket \mathbf{Y} \rrbracket_0, \llbracket \mathbf{X} \rrbracket_1, \llbracket \mathbf{Y} \rrbracket_1)$ (Equation 4.9); (*ii*) during the re-sharing phase where S_0 sends inconsistent shares to other servers (Equation 4.10); (*iii*) after the re-sharing phase where S_0 deviates the linear combination (Equation 4.11). It can be seen that (*ii*) and (*iii*) may result in the servers storing inconsistent copies with each other, which can be detected at the output phase of the protocol. Specifically, every server S_i performs the random linear combination of all components $\llbracket z_j \rrbracket_i$ in the resulting matrix $[\![\mathbf{Z}]\!]_i$ as $[\![x]\!]_i \leftarrow \sum_j r_j [\![z_j]\!]_i$. Due to RSS, $[\![x]\!]_i$ will be computed by two servers on their own shares. This means if $[\![\mathbf{Z}]\!]_i$ is inconsistent from two servers, the client will receive two different x_i and, therefore, can tell whether one of the servers has cheated.

Finally, we show that if the adversary adds any error to his local computation before the re-sharing phase (*i.e.*, stage (*i*)), they will also get caught. Let $\mathbf{T}$ be an error introduced by S_0 during its local computation. By Equation 4.9, the computation will now become $(\mathbf{X} \times \mathbf{Y} + \mathbf{T})$. Hence, to make the client not abort, S_0 should modify its shares of the MACs in such a way that all servers will compute the valid share of the MAC of the form $\alpha(\mathbf{X} \times \mathbf{Y} + \mathbf{T})$ at the end. Remark that the MAC of the multiplication $\mathbf{X} \times \mathbf{Y}$ is $\alpha(\mathbf{X} \times \mathbf{Y})$, which can be computed by multiplying $\alpha\mathbf{X}$ with $\mathbf{Y}$ via replicated secret sharing as

$$\begin{aligned}\alpha\mathbf{X} \times \mathbf{Y} =& ([\![\alpha\mathbf{X}]\!]_0 + [\![\alpha\mathbf{X}]\!]_1 + [\![\alpha\mathbf{X}]\!]_2) \times ([\![\mathbf{Y}]\!]_0 + [\![\mathbf{Y}]\!]_1 + [\![\mathbf{Y}]\!]_2)\\ =& ([\![\alpha\mathbf{X}]\!]_0 \times [\![\mathbf{Y}]\!]_0 + [\![\alpha\mathbf{X}]\!]_0 \times [\![\mathbf{Y}]\!]_1 + [\![\alpha\mathbf{X}]\!]_1 \times [\![\mathbf{Y}]\!]_0)_{S_0} +\\ & ([\![\alpha\mathbf{X}]\!]_1 \times [\![\mathbf{Y}]\!]_1 + [\![\alpha\mathbf{X}]\!]_1 \times [\![\mathbf{Y}]\!]_2 + [\![\alpha\mathbf{X}]\!]_2 \times [\![\mathbf{Y}]\!]_1)_{S_1} +\\ & ([\![\alpha\mathbf{X}]\!]_2 \times [\![\mathbf{Y}]\!]_2 + [\![\alpha\mathbf{X}]\!]_2 \times [\![\mathbf{Y}]\!]_0 + [\![\alpha\mathbf{X}]\!]_0 \times [\![\mathbf{Y}]\!]_2)_{S_2}.\end{aligned} \tag{4.12}$$

Let $\mathbf{T}'$ be an error introduced by S_0 during the local computation in Equation 4.12. As shown above, the resulting MAC computation will be of the form $(\alpha\mathbf{X} \times \mathbf{Y} + \mathbf{T}')$. Thus,

$$\alpha(\mathbf{X} \times \mathbf{Y}) + \mathbf{T}' = \alpha(\mathbf{X} \times \mathbf{Y} + \mathbf{T}) \iff \mathbf{T}' = \alpha\mathbf{T}. \tag{4.13}$$

Since α is the global MAC key known only be the client, the probability that S_0 can generate a valid $(\mathbf{T}, \mathbf{T}')$ pair is $\frac{1}{|\mathbb{F}_p|}$. That means the adversary cannot deviate from the protocol, otherwise, they will cause the client to abort the protocol with high probability.

We define Game 1' as follows. In this game, the client executes $\Pi_{\mathcal{F}}$ with three servers using dummy matrices, instead of the one chosen by the environment $\mathcal{Z}$. We introduce the ideal functionality $\mathcal{F}_{\mathsf{mult}}$, which the client queries to answer the environment requests. During executing $\Pi_{\mathcal{F}}$, if the client does not abort, the output of $\mathcal{F}$ is forwarded to $\mathcal{Z}$. We claim that Game 1 and Game 2 are statistically indistinguishable, in which the view of the adversary can be simulated given the view of the honest servers. At the beginning of the $\Pi_{\mathcal{F}}$ protocol, the client distributes the authenticated share of the multiplication matrices to each server. Due to the perfect secrecy of additive secret sharing, all these shares are uniformly distributed. After the local computation, each server re-shares the computed result with additive secret sharing and distributes the shares to other servers (*i.e.*, step 1 in Figure 4.25). Such shares are also uniformly distributed due to the security of additive secret sharing. All these properties permit to simulate the view of the adversary given the view of the honest servers.

We define Game 0' similar to Game 0 except that the client uses dummy matrices to interact with the servers, instead of the ones provided by the environment $\mathcal{Z}$. The client queries the ideal functionality $\mathcal{F}_{\mathsf{pir}}$ on the actual input provided by $\mathcal{Z}$ and forwards the output to $\mathcal{Z}$. We claim that Game 1' and Game 0' are indistinguishable using the same argument as between Game 1 and Game 0. We can see that Game 0' is the ideal game $\textsc{Ideal}_{\mathcal{F}_{\mathsf{mult}},\mathcal{S}_{\mathsf{mult}},\mathcal{Z}}$ with simulator $\mathcal{S}_{\mathsf{mult}}$ and the environment $\mathcal{Z}$.

Putting all the games together, we have that Game 0 $\equiv$ Game 1 $\equiv$ Game 1' = Game 0' and this completes the proof. □

Corollary 4. In the ℓ-server setting, the matrix multiplication protocol by SPDZ in Figure 4.26 is secure against a malicious adversary corrupting (ℓ-1) servers.

Proof. This proof can be derived from the proof of Lemma 7 and the one in [51] so that we will not present it in detail due to repetition. Intuitively, the proof works by defining the simulator and the hybrid games similar to the ones in the proof of Lemma 7 with a slight tweak as follows. It was proven in [51] that the probability that the adversary can cheat during the SPDZ multiplication over two shared values is $1/|\mathbb{F}_p|$. Thus, we can follow the proof in [51] and present it for the vectorized values (*i.e.*, matrix) to construct Game 1. Similarly, it has been also proven in [51] that the view of the ideal process and the real process is statistically indistinguishable by the SPDZ multiplication over single values. We can follow [51] to construct Game 1' for vectorized values. □

Lemma 8. In the 3-server setting, the XOR-PIR protocol on an authenticated shared database in Figure 4.27 is secure against a malicious adversary corrupting an arbitrary server.

Proof. We prove this by constructing a simulator such that the environment $\mathcal{Z}$ cannot distinguish between the real protocol and the ideal functionality. We define the simulator $\mathcal{S}_{\mathsf{pir}}$ in the ideal world and a sequence of hybrid games as follows.

To simulate the setup protocol, the simulator follows the honest setup procedure with a dummy database $\mathbf{B}'$ containing N dummy data items, on behalf of the client. For the retrieval simulation, the simulator follows the honest retrieval procedure, on behalf of the client, to read a block with dummy ID. During the access operation, if the client (executed by $\mathcal{S}_{\mathsf{pir}}$) aborts then the simulator sends abort to $\mathcal{F}_{\mathsf{pir}}$ and stops. Otherwise, the simulator returns ok to $\mathcal{F}_{\mathsf{pir}}$.

We define a sequence of hybrid games to show that the following real world and the simulation in the ideal world are statistically indistinguishable:

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F}_{\mathsf{pir}},\mathcal{S}_{\mathsf{pir}},\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda).$$

We define Game 0 as the real game $\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda)$ with an environment $\mathcal{Z}$ and three servers in the presence of an adversary $\mathcal{A}$. presented in Definition 10. In this case, the real PIR protocol $\Pi_{\mathcal{F}}$ is the one presented in Figure 4.27. Without loss of generality, we assume server S_0 is corrupted.

We define Game 1 as follows. In this game, the client maintains a copy of the database locally. Whenever the client privately retrieves a data item block from the servers, if abort does not occur, the client uses its version stored locally for further processing. The difference between Game 0 and Game 1 happens if at some point, the client retrieves an incorrect item from the servers, but unable to detect since the adversarial server generates a valid MAC for it (thus the abort does not occur). We claim that Game 0 and Game 1 are statistically indistinguishable. Similar to other proofs, the intuition is to show that if the adversarial server ever cheats during the PIR computation, it will be caught with high probability as follows.

Every server $S_i \in \{S_0, S_1, S_2\}$ stores two authenticated shares of the database $\mathbf{B}$ as $\langle \mathbf{B} \rangle_i = ([\![\mathbf{B}]\!]_i, [\![\alpha \mathbf{B}]\!]_i) = (([\![\mathbf{b}_j]\!]_i), ([\![\alpha \mathbf{b}_j]\!]_i))$ and $\langle \mathbf{B} \rangle_{i+1} = ([\![\mathbf{B}]\!]_{i+1}, [\![\alpha \mathbf{B}]\!]_{i+1}) = (([\![\mathbf{b}_j]\!]_{i+1}), ([\![\alpha \mathbf{b}_j]\!]_{i+1}))$. According to the XOR-PIR protocol, every S_i aggregates (*i.e.*, XOR) all authenticated shares of the blocks that correspond with the value '1' in the client queries. Let $\mathbf{e}_0$ and $\mathbf{e}_1$ be the queries that the client sends to S_0, which are the binary strings of length ℓ, where ℓ is the number of database blocks. Let $\hat{e}_0$ (*resp.* $\hat{e}_1$) be the error introduced by S_0 during the bitwise operations between $[\![\mathbf{B}]\!]_0$ (*resp.* $[\![\mathbf{B}]\!]_1$) and $\mathbf{e}_0$ (*resp.* $\mathbf{e}_1$). So, the reply that the client obtains from S_0 is of the form

$$\begin{aligned} [\![b]\!]_0^{(0)} \oplus e_0 &= \bigoplus_{\forall j \,:\, \mathbf{e}_0[j]=1} [\![b_j]\!]_0 \\ [\![b]\!]_1^{(0)} \oplus e_1 &= \bigoplus_{\forall j \,:\, \mathbf{e}_1[j]=1} [\![b_j]\!]_1 \end{aligned} \tag{4.14}$$

Since S_1, S_2 are honest, the client obtains honest answers $[\![b]\!]_0^{(1)}$, $[\![b]\!]_1^{(1)}$, $[\![b]\!]_2^{(0)}$, $[\![b]\!]_2^{(1)}$ from them. Due to XOR-PIR, the client reconstructs the following shares from the server:

$$\begin{aligned} [\![b]\!]_0 \oplus e_0 &= [\![b]\!]_0^{(0)} \oplus [\![b]\!]_0^{(1)} = [\![b]\!]_0 + t_0 \\ [\![b]\!]_1 \oplus e_1 &= [\![b]\!]_1^{(0)} \oplus [\![b]\!]_1^{(1)} = [\![b]\!]_1 + t_1 \\ [\![b]\!]_2 &= [\![b]\!]_2^{(0)} \oplus [\![b]\!]_2^{(1)} = [\![b]\!]_2 \end{aligned} \tag{4.15}$$

By additive secret sharing, the client recovers a block of the form $b' = [\![b]\!]_0 + [\![b]\!]_1 + [\![b]\!]_2 + t_0 + t_1 = b + t$ where $t = t_0 + t_1 \in \mathbb{F}_p$. Equation 4.15 implies that if the adversary introduces any bit of error during the XOR computation, the client will recover incorrect shares of the original block b thereby, obtaining an arbitrary block b' different from b. In order to make the client not abort, the adversary must inject the errors during the bitwise computation between $\langle \alpha \mathbf{B} \rangle_0$ with $\mathbf{e}_0$ and $\langle \alpha \mathbf{B} \rangle_1$ with $\mathbf{e}_1$, in such a way that allows the client to reconstruct a valid MAC for b', *i.e.*, $\alpha b' = \alpha b + \alpha t$. Since α is unknown, the probability that the adversary can cheat to make the client reconstruct a valid MAC for b' is $\frac{1}{|\mathbb{F}_p|}$. That means the adversary must follow the protocol faithfully, otherwise they will cause the client to abort with high probability.

We define Game 1' as follows. The client executes the Setup with a dummy database $\mathbf{B}'$ (as similar to $\mathcal{S}_{\mathsf{pir}}$) instead of the one provided by $\mathcal{Z}$. For each request, the client executes a dummy retrieval with three servers instead of the one chosen by $\mathcal{Z}$. In this game, we use the ideal functionality $\mathcal{F}_{\mathsf{pir}}$, which store the database provided by $\mathcal{Z}$ in the setup phase, to answer the environment requests. For each retrieval, if the client does not abort, it forwards the output of $\mathcal{F}_{\mathsf{pir}}$ to $\mathcal{Z}$.

We say that Game 1 and Game 1' are indistinguishable. Notice that if the client does not abort in these games, then the data item is retrieved correctly and the corrupted server follows the protocol faithfully as the honest servers. In our PIR protocol, the database is stored in the form of authenticated shares at the server, which are uniformly distributed due to the perfect privacy of additive secret sharing. The PIR queries are shared by XOR secret sharing, which are also uniformly distributed due to the perfect secrecy of XOR. Due to PIR, the computation is performed over the entire database at each server. All these security properties indicate that the view of the adversary can be simulated given the view of the honest servers that execute the XOR-PIR protocol with the client.

We define Game 0' similar to Game 0 except that the client uses a dummy database and a dummy retrieval index to interact with the servers, instead of the ones provided by the environment $\mathcal{Z}$. The client queries the ideal functionality $\mathcal{F}_{\mathsf{pir}}$ on the actual input provided by $\mathcal{Z}$ and forwards the output to $\mathcal{Z}$. We claim that Game 1' and Game 0' are indistinguishable using the same argument as between Game 1 and Game 0. Game 0' is the ideal game $\textsc{Ideal}_{\mathcal{F}_{\mathsf{pir}},\mathcal{S}_{\mathsf{pir}},\mathcal{Z}}$ with simulator $\mathcal{S}_{\mathsf{pir}}$ and the environment $\mathcal{Z}$. Putting all the games together, we have that Game 0 $\equiv$ Game 1 $\equiv$ Game 1' $\equiv$ Game 0' and this completes the proof. □

We now prove the security of multi-server PIR protocols. We first present the security model of multi-server PIR based on simulation as follows.

Definition 11 (Multi-Server PIR with Verifiability). We first define the ideal world and real world as follows. Let $\mathcal{F}_{\mathsf{pir}}$ be an ideal functionality, which maintains a version of the database on behalf of the client and answers the client's requests as follows.

An environment $\mathcal{Z}$ provides a database $\mathbf{B}$ to the client. The client sends $\mathbf{B}$ to $\mathcal{F}_{\mathsf{pir}}$. $\mathcal{F}_{\mathsf{pir}}$ stores $\mathbf{B}$ and notifies the simulator $\mathcal{S}_{\mathsf{pir}}$ the completion of the setup, but not the content of $\mathbf{B}$. The simulator $\mathcal{S}_{\mathsf{pir}}$ returns ok or abort to $\mathcal{F}_{\mathsf{pir}}$. $\mathcal{F}_{\mathsf{pir}}$ then returns ok or $\perp$ to the client accordingly. In each retrieval step, the environment $\mathcal{Z}$ specifies an index idx as the client's input. The client sends idx to $\mathcal{F}_{\mathsf{pir}}$. $\mathcal{F}_{\mathsf{pir}}$ notifies the simulator $\mathcal{S}_{\mathsf{pir}}$ (without revealing idx to $\mathcal{S}_{\mathsf{pir}}$). If $\mathcal{S}_{\mathsf{pir}}$ returns ok to $\mathcal{F}_{\mathsf{pir}}$, $\mathcal{F}_{\mathsf{pir}}$ sends $\mathbf{b} \leftarrow \mathbf{B}[\mathsf{idx}]$ to the client. The client then returns $\mathbf{b}$ to the environment $\mathcal{Z}$. If $\mathcal{S}_{\mathsf{pir}}$ returns abort to $\mathcal{F}_{\mathsf{pir}}$, $\mathcal{F}_{\mathsf{pir}}$ returns $\perp$ to the client.

In the real world, $\mathcal{Z}$ gives the client a database $\mathbf{B}$. The client executes the Setup with ℓ servers $(S_0, \dots, S_{\ell-1})$. At each time step, $\mathcal{Z}$ specifies an input idx to the client. The client executes the PIR protocol Π with $(S_0, \dots, S_{\ell-1})$. The environment $\mathcal{Z}$ gets the view of the adversary $\mathcal{A}$ after every operation. The client outputs to the environment $\mathcal{Z}$ the output of Π or abort.

We say that a protocol $\Pi_{\mathcal{F}}$ securely realizes the ideal functionality $\mathcal{F}_{\mathsf{pir}}$ in the presence of a malicious adversary corrupting t servers iff for any PPT real-world adversary that corrupts up to t servers, there exists a simulator $\mathcal{S}_{\mathsf{pir}}$ such that for all non-uniform, polynomial-time environment $\mathcal{Z}$, there exists a negligible function negl such that

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F}_{\mathsf{pir}},\mathcal{S}_{\mathsf{pir}},\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda).$$

Corollary 5. In the 3-server setting, the RSS-PIR protocol on an authenticated shared database in Figure 4.28 is secure against a malicious adversary corrupting an arbitrary server.

Proof. This can be derived from the proof of Lemma 8 so we will not repeat due to repetition. Intuitively, we construct the Simulator $\mathcal{S}_{\mathsf{pir}}$ and the hybrid games similar to the ones in the proof of Lemma 8, with a small change as follows. This protocol interprets the PIR database as a matrix to execute the matrix multiplication protocol via replicated secret sharing. We have shown in the proof of Lemma 7 that the probability that the adversary can inject a malicious input to the stage (i) of the matrix multiplication without being caught is $\frac{1}{|\mathbb{F}_p|}$. We can use this argument to construct Game

1. In Game 1', we can argue that the view of the ideal process and the real process is statistically indistinguishable because the retrieval queries in this PIR are in the form of additive secret sharing, and therefore, are uniformly distributed. □

Corollary 6. In the ℓ-server setting, the SPDZ-PIR protocol on an authenticated shared database setting in Figure 4.29 is secure against a malicious server corrupting up to $(\ell - 1)$ servers.

Proof. Similar to the proof of Corollary 5, this proof can be derived from the proof of Lemma 8 by replacing the XOR operations with the matrix multiplication protocol by SPDZ secret sharing. □

Now we state the security of our MACAO schemes as follows.

Theorem 5 (MACAO Security). MACAO framework is statistically (information-theoretically) secure by Definition 9.

Proof. We define a simulator in the ideal world and a sequence of hybrid games as follows.

To simulate the setup protocol, the simulator follows the honest setup procedure with a database DB' containing N blocks b_i for each $b_i \overset{\$}{\leftarrow} \{0,1\}^{|b|}$, on behalf of the client. For the access simulation, the simulator follows the honest access protocol, on behalf of the client, to read a dummy block with dummy ID. During the access operation, if the client protocol (executed by the simulator) aborts then the simulator sends abort to $\mathcal{F}_{\mathsf{oram}}$ and stops. Otherwise, the simulator returns ok to $\mathcal{F}_{\mathsf{oram}}$.

We define a sequence of hybrid games to show that the following real world and the simulation in the ideal world are statistically indistinguishable:

$$|\Pr[\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F}_{\mathsf{oram}},\mathcal{S}_{\mathsf{oram}},\mathcal{Z}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda).$$

We define Game 0 as the real game $\textsc{Real}_{\Pi_{\mathcal{F}},\mathcal{A},\mathcal{Z}}(\lambda)$ with an environment $\mathcal{Z}$ and ℓ servers in the presence of an adversary $\mathcal{A}$ as presented in Definition 8. In this game, the real ORAM access protocol $\Pi_{\mathcal{F}}$ is the one presented in Figure 4.31, where the retrieval and evictions subroutines are presented in Figure 4.32, Figure 4.33, Figure 4.34, Figure 4.35.

We define Game 1 as follows. In this game, the client maintains a copy of the data blocks b_i in plaintext locally. Whenever the client accesses a block b_i from the servers, If abort does not occur, the client uses its plaintext stored locally for further processing. The difference between Game 0 and Game 1 happens if, at some point, the client retrieves a modified block from the servers, but unable to detect since the adversarial server generates a valid MAC for it (thus the abort does not occur).

We claim that Game 0 and Game 1 are statistically indistinguishable as follows. First, MACAO schemes harness multi-server PIR protocols based on XOR/RSS/SPDZ scheme to perform the retrieval phase. We have shown that all these protocols are secure against the malicious adversary by Lemma 8, Corollary 5 and Corollary 6, where the probability that the adversary can tamper with the block and forge a valid MAC is $\frac{1}{|\mathbb{F}_p|}$. Second, MACAO schemes use the authenticated matrix multiplication protocols by RSS or SPDZ to perform the eviction. By Lemma 7 and Corollary 4, all these protocols are secure against the malicious adversary, where the probability that the adversary can deviate from the protocol without being caught is $\frac{1}{|\mathbb{F}_p|}$.

We define Game 1' as follows. In this game, the client executes the Setup with a dummy database DB' similar to the simulation, instead of the one provided by the environment $\mathcal{Z}$. For each access operation, the client executes a dummy access with ℓ servers instead of the one chosen by $\mathcal{Z}$. We also introduce the ideal functionality $\mathcal{F}_{\mathsf{oram}}$ storing the database provided by the environment $\mathcal{Z}$ in the setup phase, which the client queries to answer the environment requests. At each time of access, if the client does not abort, it forwards the output of $\mathcal{F}_{\mathsf{oram}}$ to $\mathcal{Z}$.

We claim that Game 1 and Game 1' are statistically indistinguishable as follows. In both games, if the client does not abort then the data block is retrieved correctly. That means the corrupted server follows the protocol faithfully as the honest servers. We show that the view of the adversary in these games can be simulated given the view of the honest servers that execute Access protocol of MACAO with the client as follows. In MACAO, the client database is stored in the ORAM tree at the server in the form of authenticated shares, which achieves a perfect security due to the perfect privacy of additive secret sharing. MACAO schemes harness the tree-ORAM paradigm, where the client privately assigns each block to a path selected uniformly at random. Once a block is retrieved, the client locally assigns it to a new random path, and therefore, it is unknown to the server. Thus, given any data request sequence, the server observes a sequence of random retrieval paths. The eviction path in MACAO is deterministic following the reverse lexicographical order and therefore, it is publicly known by anyone. MACAO schemes employ the push-down strategy in [179], which achieves a negligible failure probability with fixed system parameters (*e.g.*, bucket size, stash size) by Lemma 6. Therefore, given any two data request sequences of the same length, the servers observe two access patterns that are statistically indistinguishable from each other (the statistical bit comes from the negligible failure probability of push-down strategy). Moreover, any computation performed by the servers is secure due to the security multi-server PIR protocols by Corollary 5, Lemma 8, Corollary 6 and multiplicative homomorphic properties of SPDZ and RSS schemes. These security properties show that the view of the adversarial server can be simulated given the view of the honest servers in MACAO.

We define Game 0' similar to Game 0 except that the client uses a dummy database and dummy access operations to interact with the servers, instead of the ones provided by the environment $\mathcal{Z}$. The client queries the ideal functionality $\mathcal{F}$ on the input of $\mathcal{Z}$ and forwards its output to the environment $\mathcal{Z}$. We claim that Game 1' and Game 0' are indistinguishable using the same argument as between Game 1 and Game 0. It is easy to see that Game 0' is the ideal game $\textsc{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ with simulator $\mathcal{S}$ and the environment $\mathcal{Z}$.

Putting all the games together, we have that Game 0 $\equiv$ Game 1 $\equiv$ Game 1' $\equiv$ Game 0' and this completes the proof. □

4.5.2.5 Cost Analysis

We analyze the asymptotic cost of our proposed MACAO schemes. We treat some parameters as constant including the finite field (p), the bucket size Z and the number of servers ℓ (*i.e.*, $\ell = 3$ in MACAO$_{\mathsf{rss}}$ and $\ell \geq 2$ in MACAO$_{\mathsf{spdz}}$). Following the tree ORAM literature (*e.g.*, [161, 169, 179]), we consider the statistical security parameter as a function of database size, *i.e.*, $\lambda = \mathcal{O}(\log N)$. Let $L = Z(H+1) = \mathcal{O}(\log N)$ be the length of the path in the MACAO structure, and $C = |b|/|\mathbb{F}_p|$ be the number of chunks per data block.

In the MACAO$_{\mathsf{rss}}$ retrieval, the client sends six L-bit binary strings and receives six $2|b|$-bit replies if using XOR-PIR. If using RSS-PIR, the client sends six $(L|\mathbb{F}_p|)$-bit queries, and receives three $2|b|$-bit replies. In the MACAO$_{\mathsf{rss}}$ eviction, the client sends to each server two authenticated shares of a data block and $(H+1)$ permutation matrices of size $(Z+1)\times(Z+1)$, where each element is $\log_2 p$ bits. Since $L = \mathcal{O}(\log N)$ and p is fixed, the client communication cost per MACAO$_{\mathsf{rss}}$ access is $\mathcal{O}(|b| + \log N)$. MACAO$_{\mathsf{spdz}}$ has a similar asymptotic bandwidth cost as MACAO$_{\mathsf{rss}}$ because they only differ in the fixed number of authenticated shares per server (2 *vs.* 1), and the number of servers ℓ (yet ℓ is fixed).

In MACAO$_{\mathsf{rss}}$, servers communicate with each other only in the eviction phase, where two authenticated shares of the entire eviction path is transmitted from one server to the others.

Hence, the server-server communication is $4L(\ell-1)|b|= \mathcal{O}(|b|\log N)$. In $\mathsf{MACAO_{spdz}}$, servers need to communicate with each other not only in the eviction but also in the retrieval phase. For each retrieval/eviction operation, every server sends the authenticated shares of the entire path and the client queries/matrices to all the others. Thus, its total server-server communication is $2L(|\mathbb{F}_p|+|b|+(Z+1)^2+|b|) = \mathcal{O}(|b|\log N)$.

In $\mathsf{MACAO_{rss}}$ retrieval, the client generates $4L$ random bits and performs XOR on L-bit data and $|b|$-bit data four times if using XOR-PIR. If using RSS-PIR, the client generates $(\ell-1)L|\mathbb{F}_p|$ random bits. In both cases, the client additionally performs $2C$ additions (for block and MAC recovery) and C multiplications (for MAC comparison) over $\mathbb{F}_p$ field. For each $\mathsf{MACAO_{rss}}$ eviction, the client generates $L(\ell-1)(Z+1)^2\log_2 p$ random bits, performs $2L(Z+1)^2(\ell-1)$ additions and $L(Z+1)^2$ multiplications over $\mathbb{F}_p$. The cost of $\mathsf{MACAO_{spdz}}$ is similar to that of $\mathsf{MACAO_{rss}}$ using RSS-PIR, but with $\ell \geq 2$.

In $\mathsf{MACAO_{rss}}$ retrieval, each server performs XOR operations on $2|b|$-bit strings approximately L times if using XOR-PIR. If using RSS-PIR, each server performs $6LC$ modular multiplications, $4LC$ additions over $\mathbb{F}_p$. Each $\mathsf{MACAO_{rss}}$ eviction incurs $6LC$ multiplications and $16LC$ additions over $\mathbb{F}_p$ with $4LC\log_2 p$ random bits being generated.

In both $\mathsf{MACAO_{rss}}$ and $\mathsf{MACAO_{spdz}}$, the position map is of size $N(\log_2 N + \log_2\log_2 N)$. We follow the eviction in [179], which requires the client to maintain a stash of size $\mathcal{O}(|b|\lambda)$ for negligible overflow probability. In total, the asymptotic client storage overhead is $\mathcal{O}(N(\log N + \log\log N) + |b|\log N)$.

An ORAM tree with N leaves can store $2N$ data blocks. Each node in the tree can store Z blocks. In $\mathsf{MACAO_{rss}}$ and $\mathsf{MACAO_{spdz}}$ schemes, each server stores one and two authenticated shares of the ORAM tree, respectively. Therefore, the storage overhead per server in $\mathsf{MACAO}_{\mathsf{rss}}$ and $\mathsf{MACAO}_{\mathsf{spdz}}$ is $4Z|b|N$ and $2Z|b|N$ bits $= \mathcal{O}(N)$, respectively.

4.5.2.6 Extensions

In this section, we describe some tricks that can be applied to our MACAO schemes.

We first propose a trick to reduce the bandwidth overhead as follows. Since our ORAM framework relies on XOR secret sharing and additive secret sharing as the main building blocks, the retrieval queries and eviction data can be created and distributed in a more communication-efficient manner. The client can generate the authenticated shares of retrieval queries, data blocks and permutation matrices using a Pseudo-Random Function (PRF) instead of a truly random function. To reduce the communication overhead, the client can create random seeds for such pseudo-random generator using a truly random function, and securely send the seeds to $(\ell - 1)$ servers so that they can generate their own shares themselves. Since the client only needs to send the shares to one server, this strategy can significantly reduce the client bandwidth overhead. The price to pay for this is the reduction of the security level from information-theoretic to computational due to the pseudo-random generation function. In $\mathsf{MACAO}_{\mathsf{rss}}$ scheme, we can further apply this trick to reduce the server-server communication overhead in the eviction phase. After performing the local computation (*e.g.*, line 1 in Figure 4.25), every server can generate and send the seeds to other servers to let them calculate their re-shared values. In this case, every server only needs to send a shared matrix (instead of four) to one other server.

We propose another trick to reduce the client storage overhead as follows. In our framework, the client maintains two major components including a position map of size $\mathcal{O}(N(\log N + \log\log N)$ and a stash $\mathbf{S}$ of size $\mathcal{O}(|b|\log N)$. While the position map can be stored remotely on the server-side using the recursion and meta-data techniques [54, 161], we present two solutions to remove $\mathbf{S}$ at the client as follows. The first solution is to store $\mathbf{S}$ remotely at the server-side in the form of authenticated shares $(\llbracket \mathbf{S} \rrbracket, \llbracket \alpha\mathbf{S} \rrbracket)$, and leverage homomorphic matrix multiplication protocols to obliviously pick and drop the block from/into $\mathbf{S}$. We treat $\mathbf{S}$ as an additional level of the ORAM tree appended to the root as suggested in [179]. Thus, when executing the PIR protocol in the retrieval phase, we need to include the stash level, and therefore, the retrieval query length (and the number of data blocks in the path) will be $(Z(H+1)+\lambda) = \mathcal{O}(\log N)$. In the eviction phase, to obliviously put a block b into $\mathbf{S}[x]$, the client creates a unit vector $\mathbf{v} = (v_0, \ldots, v_{\lambda-1})$ where $v_x = 1$ and $v_y = 0$ for all $0 \leq y \neq x < \lambda$. The client creates and sends authenticated shares $\langle b \rangle$ and $\langle \mathbf{v} \rangle$, and every server performs $\langle \mathbf{S} \rangle \leftarrow \langle b \rangle \times \langle \mathbf{v} \rangle^\top + \langle \mathbf{S} \rangle$. To obliviously pick a block at $\mathbf{S}[x']$, the client creates a unit vector $\mathbf{v}' = (v_0, \ldots, v_{\lambda-1})$, where $e_{x'} = 1$ and $e_y = 0$ for all $0 \leq y \neq x < \lambda$. The client creates and sends authenticated shares of $\mathbf{v}'$ and the servers perform $\langle b \rangle \leftarrow \langle \mathbf{v}' \rangle \times \langle \mathbf{S} \rangle^\top$ and $\langle \mathbf{S} \rangle \leftarrow \langle \mathbf{S} \rangle - \langle \mathbf{v}' \rangle \times \langle \mathbf{S} \rangle^\top$.

The other method is to implement the triplet eviction principle proposed in [54] using homomorphic properties of additive shares as similar to [89]. Since this approach requires the bucket size to be of size $\mathcal{O}(\log N)$ for negligible overflow, the computation and communication in the retrieval phase will be increased by a factor of $\mathcal{O}(\log N)$.

4.5.3 Experimental Evaluation

4.5.3.1 Implementation

We fully implemented both MACAO schemes in §4.5.2.3 all the extensions in §4.5.2.6. In MACAO$_{\text{spdz}}$, we implemented only the online phase of the SPDZ-based matrix multiplication protocol since we assume that authenticated shares of Beaver-like matrix triples were pre-computed sufficiently in the offline phase. The implementation was written in C++ consisting of more than 25K lines of code for the MACAO controllers at the client- and server-side. We used three main external libraries: (*i*) Shoup's NTL library v9.10.0 [162] for the server computation; (*ii*) ZeroMQ library [99] for the network communication; (*iii*) pthread for server computation parallelization. Our implementation made use of SIMD instructions to optimize the performance of bit-wise operations and vectorized computations on Intel x64 architecture. For the reduced bandwidth trick, we used tomcrypt library [53] to implement seeded pseudo-random number generators using sober128 stream cipher. Each server stored a 128-bit secret seed shared with the client. In $\mathsf{MACAO}_{\mathsf{rss}}$ scheme, each server stored two extra 128-bit secret seeds shared with the other two servers, which are used to re-share the local computation during the RSS-based matrix multiplication in the eviction phase.

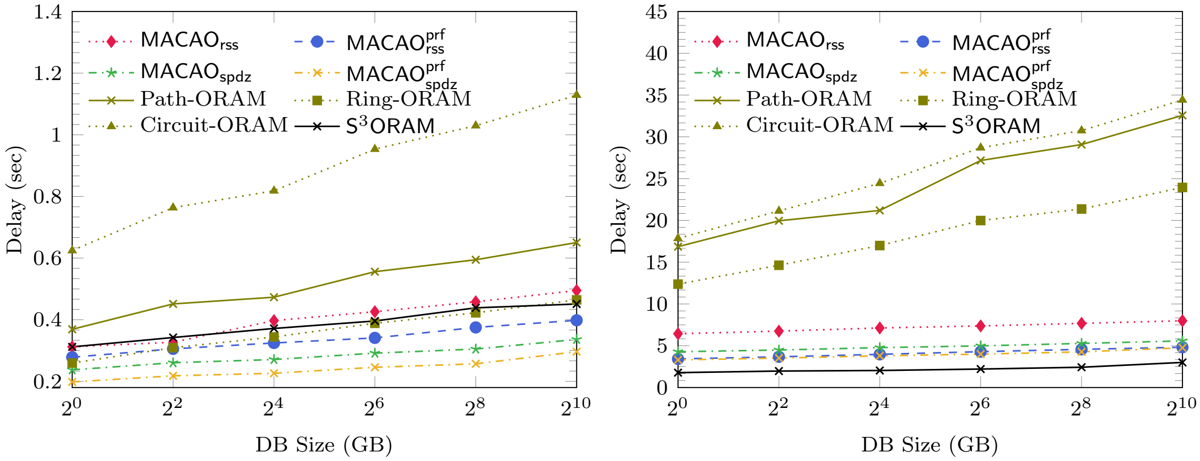
Our implementation can be found at https://github.com/thanghoang/MACAO.

4.5.3.2 Configuration and Methodology

For the server-side, we employed three c5.4xlarge Amazon EC2 instances each equipped with 3.00 GHz 16-core Intel Xeon 8124M CPU, 16 GB RAM and 8 TB networked Elastic Block Storage (EBS)-based SSD. For the client, we used a Macbook Pro with 2.6 GHz 6-core Intel Core i7 8850H CPU and 32 GB RAM.

We used a random database of size ranging from 1 GB to 1 TB. We selected two standard data block sizes including 4 KB and 256 KB as these are commonly used in small-scale and large-scale file systems, respectively.

We used a standard home Internet setting for client-server communication. Specifically, the laptop client was connected to the Internet via WiFi with 54.5 Mbps download, 5.72 Mbps upload throughput and 20ms round-trip latency to Amazon EC2 servers. The server instances were set up



(a) Block size $|b|= 4$ KB

(b) Block size $|b|= 256$ KB

Figure 4.36: End-to-end delay of MACAO schemes and their counterparts.

geographically close to each other, which resulted in the inter-server throughput of 1 Gbps with 3ms round-trip latency.

We selected S^{3}ORAM and Path-ORAM as the main counterparts for MACAO since S^{3}ORAM is the most efficient multi-server ORAM scheme (but with no malicious security), while the others are state-of-the-art single-server ORAM schemes. Although Onion-ORAM offers similar properties to MACAO (*e.g.*, constant bandwidth, malicious security), we did not explicitly compare our framework with it because the delay of Onion-ORAM was shown significantly higher than S^{3}ORAM and Path-ORAM. The main performance evaluation metric is *the end-to-end delay*, which captures the processing time at client- and server-side (*e.g.*, I/O, computation) as well as the network communication among parties. We configured the system parameters for all schemes as follows so that they achieve the same failure probability of 2^{-80}.

- MACAO: We selected the bucket size $Z = 2$, stash size $|\mathbf{S}|= 80$ and performed two deterministic evictions per access as suggested in [179]. We selected a 59-bit prime field for the computational advantage of 64-bit architecture and the optimization of NTL library. In this experiment, we demonstrated the performance of MACAO$_{\mathsf{rss}}$ with the RSS-PIR protocol only. It is because this protocol allows to further enable secure computation on the accessed block, and its delay is higher than XOR-PIR (so the comparison with counterparts will be more conservative).

- S^{3}ORAM: We used the open-sourced implementation in [90]. We selected $Z = 74$ and eviction frequency $A = 37$. Similar to MACAO, we selected a 59-bit prime field.
- Path-ORAM: We implemented a prototype of Path-ORAM. We selected $Z = 4$ and $|\mathbf{S}| = 80$ [169]. We used Intel AES-NI library to accelerate cryptographic operations. We used AES with counter mode (CTR) for encryption and decryption. We created the MAC tag for each node in the Path-ORAM tree using AES-128 CMAC.
- Ring-ORAM: We selected standard parameters as suggested in [150] (*i.e.*, $Z = 16$ and $A = 20$).
- Circuit-ORAM: We selected $Z = 2$, $|\mathbf{S}| = 80$ and performed two evictions per access as in [179]. Similar to Path-ORAM, we used AES-128 CMAC for authentication and AES-CTR for encryption with Intel AES-NI.

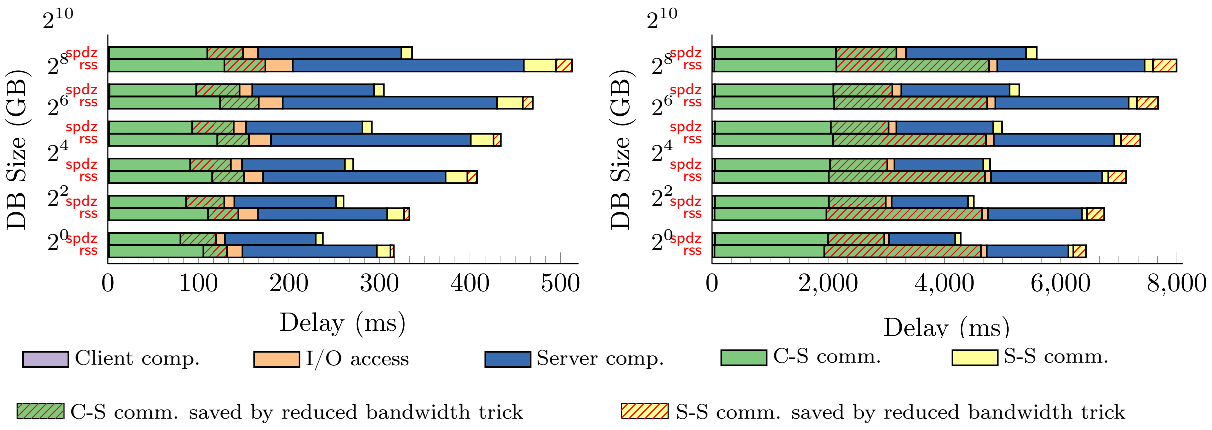
4.5.3.3 Setup Delay

We first discuss the time to set up necessary MACAO components (*e.g.*, authenticated shares of the ORAM trees, position maps) on the client machine. The delay grew linearly to the database size. Specifically, it took around 370 s to 357,601 s to construct MACAO$_{\mathsf{rss}}$ components for 1 GB to 1 TB database with 4 KB block size. For MACAO$_{\mathsf{spdz}}$ components, it took 244 s to 241,553 s, which was round 1.5 times faster than MACAO$_{\mathsf{rss}}$ since MACAO$_{\mathsf{spdz}}$ only needs 2 servers (*vs.* 3 in MACAO$_{\mathsf{rss}}$). With 255 KB block size, the setup delay was 142 s to 135,927 s for MACAO$_{\mathsf{spdz}}$ components, and 209 s to 215,482 s for MACAO$_{\mathsf{rss}}$ components. We note that we did not measure the preprocessing cost to generate multiplication triples for MACAO$_{\text{spdz}}$ scheme as it is out-of-scope of this dissertation. We refer curious readers to [112] for its detailed benchmarks.

4.5.3.4 Overall Result

We present in Figure 4.36 the end-to-end delay of MACAO schemes compared with selected counterparts with 4 KB and 255 KB block sizes and database sizes from 1 GB to 1 TB. In the home network setting, all MACAO schemes outperformed Path-ORAM and Circuit-ORAM in all testing cases, especially when the block size was large (*i.e.*, 256 KB). Specifically, Path-ORAM and Circuit-ORAM took 369 ms to 650 ms and 625 ms to 1.2 s to access a 4 KB block, respectively, whereas MACAO schemes took 198 ms to 336 ms. All MACAO schemes (except MACAO$_{\text{rss}}$) were also faster than Ring-ORAM for 4 KB block access. For 256 KB block access, the performance gap between MACAO and single-server ORAM schemes significantly increased since MACAO featured the constant client-bandwidth blowup. In particular, Path-ORAM, Circuit-ORAM and Ring-ORAM took 16 s to 32 s, 17 s to 34 s and 12 s to 24 s, respectively, for each 256 KB-block access, whereas MACAO schemes only took 3.3 s to 5.5 s. This resulted in MACAO being up to 7× faster than single-server ORAM schemes.

On the other hand, the performance of MACAO schemes was comparable to S^{3}ORAM, where S^{3}ORAM took 312 ms to 451 ms per 4 KB-block access, and 1.78 s to 3.11 s per 256 KB-block access, respectively. MACAO$_{\text{spdz}}$ scheme was faster than S^{3}ORAM for 4 KB-block access since it operated on two servers (*vs.* 3 in S^{3}ORAM) with small amount of data, and the retrieval phase of MACAO incurred less number of blocks to be computed than S^{3}ORAM($\mathcal{O}(\log N)$ *vs.* $\mathcal{O}(\log^2 N)$). We notice that MACAO$_{\text{spdz}}$ operates on the preprocessing model, where their online access operation performance depends on the availability of authenticated matrix multiplication shares computed in the offline



(a) $|b|= 4$ KB

(b) $|b|= 256$ KB

Figure 4.37: Cost breakdown of MACAO schemes.

phase. For 256 KB-block access, S^{3}ORAM was approximately 1.5 times faster than MACAO schemes. This is mainly because MACAO schemes perform the computation on the information-theoretic MAC components, whose size is equal to the block size. Notice that S^{3}ORAM does not have the MAC and it does not offer integrity and security against the malicious adversary.

One might also observe from Figure 4.36 that the bandwidth reduction trick significantly lowered the end-to-end delays of MACAO schemes (denoted as $\mathsf{MACAO}_{\mathsf{rss}}^{\mathsf{prf}}$ and $\mathsf{MACAO}_{\mathsf{spdz}}^{\mathsf{prf}}$ schemes). This trick allowed us to reduce the performance gap between the MACAO schemes using RSS and SPDZ when the amount of data to be transmitted was large as in the 256 KB-block access. The price to pay for such efficiency is the reduction from information-theoretic to computational security. To aid more understanding, we provide the detailed cost of MACAO schemes in the following section.

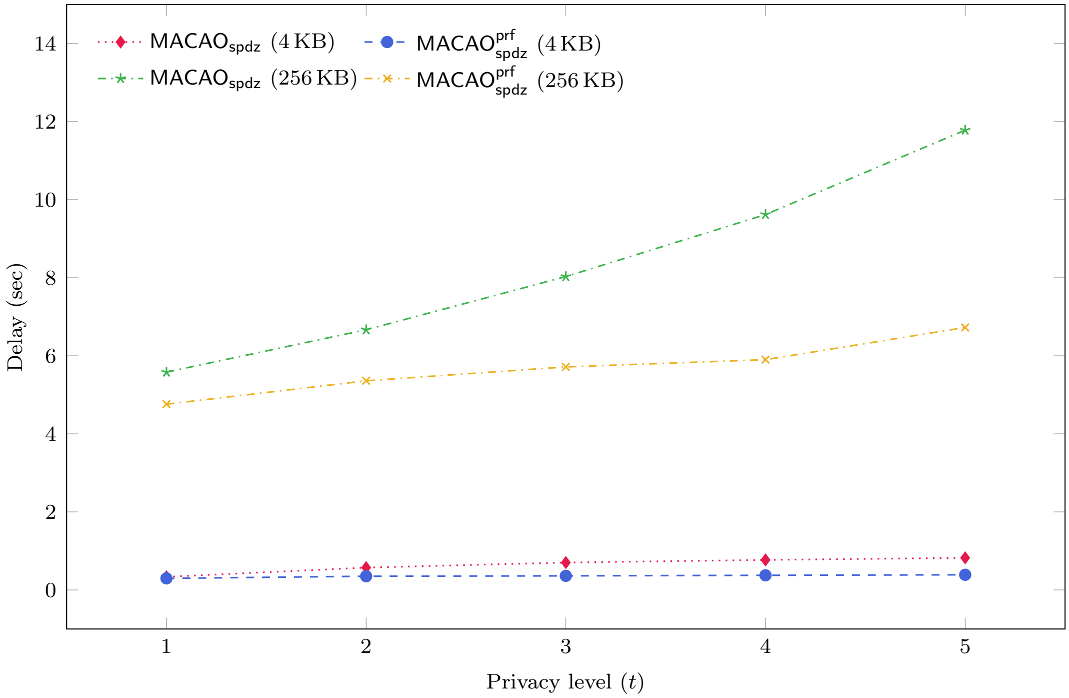


Figure 4.38: End-to-end delay with varied privacy levels.

4.5.3.5 Cost Breakdown

We decomposed the delay of MACAO schemes to investigate cost factors that impact the performance. As shown in Figure 4.37, there were four main sources of delay including client processing, server processing, client-server communication and server-server communication.

MACAO schemes incurred very low computation at the client-side thereby, making them the ideal choice for resource-limited clients such as mobile devices. The client main task was to generate shares of the retrieval query and permutation matrices for eviction by invoking pseudo/true-random number generator. The client recovered the accessed block and verified its integrity by performing some modular additions and multiplications. All these operations are very lightweight, all of which cost less than 4 ms and 40 ms for 4 KB and 256 KB block size on 1 TB database, respectively.

We disabled default caching mechanisms [146] to minimize the impact of random access sequence on the I/O latency. The disk access contributed a small amount to the delay of MACAO schemes due to the following reasons. The MACAO structure was stored on a network-based storage unit called EBS with 2.1 Gbps throughput. Meanwhile, the amount of data to be loaded per retrieval was small, which was only $4|b|(H+1)$ KB, where $|b|\in\{4,256\}$ and $H\in\{11,\ldots,27\}$ for up to 1 TB of outsourced data. In MACAO schemes, the disk I/O access only impacted the retrieval, but not eviction. This is because MACAO schemes follow the deterministic eviction, where the data along the eviction path can be pre-loaded into the memory before the push-down operation. Hence, the data can be read directly from the cache if needed, given that they were processed in the previous operations but have not been written to the disk yet.

This contributed a large portion to the total delay, mostly due to the matrix multiplication in the eviction phase. The server computation in MACAO$_{\text{rss}}$ was higher than in MACAO$_{\text{spdz}}$ since it incurred more number of additions than MACAO$_{\text{spdz}}$ for each homomorphic multiplication.

MACAO schemes feature a constant client-bandwidth blowup similar to S^{3}ORAM. Therefore, only the query size and the eviction matrix size increased when the database size increased while the number of data blocks to be transmitted remained the same. Therefore, although it was one of the most significant factors contributing to the total delay, the client-server communication cost of MACAO schemes was likely to remain the same when increasing the database size as shown in Figure 4.37, where most of the time was spent to download/upload a constant number authenticated shares of the data blocks. MACAO$_{\text{spdz}}$ incurred less client-server communication delay than MACAO$_{\text{rss}}$ because it only needs two servers, instead of three. Figure 4.37 also shows that the bandwidth reduction trick significantly reduced the client communication delay (the green bar with red filled pattern). This trick allows the client to send the authenticated share to only one server, thereby making the client-communication overhead of $\mathsf{MACAO}_{\mathsf{rss}}$ and $\mathsf{MACAO}_{\mathsf{spdz}}$ schemes almost the same.

Server-Server communication is the second smallest portion of the total delay. We can also see that the bandwidth reduction trick also helped to reduce the server-server communication in $\mathsf{MACAO}_{\mathsf{rss}}$ scheme (the yellow bar with red pattern in Figure 4.37). The $\mathsf{MACAO}_{\mathsf{rss}}$ scheme had higher inter-server communication delay than $\mathsf{MACAO}_{\mathsf{spdz}}$ since three servers communicated with each other, compared with only two in $\mathsf{MACAO}_{\mathsf{spdz}}$.

MACAO schemes harness the eviction strategy in [179] so that they incur a constant server storage blowup. In $\mathsf{MACAO}_{\mathsf{rss}}$, every server stores two authenticated shares of the ORAM tree so that storage overhead per server is $8|\mathsf{DB}|$ (*i.e.*, 4 times blowup from [179]). On the other hand, every server in $\mathsf{MACAO}_{\mathsf{spdz}}$ stores one authenticated shared ORAM tree, and therefore, the server storage overhead is $4|\mathsf{DB}|$ (2 times blowup from [179]). Regarding the client storage, $\mathsf{MACAO}_{\mathsf{rss}}$ schemes add an extra of $\mathcal{O}(N \log\log N)$ bits to the storage overhead of [179], which is analytically $|N|\log_2 N + \log_2(\log_2 N) + 80|b|$ in total. Empirically, with 1 TB DB and 256 KB block size, the client storage overhead of $\mathsf{MACAO}_{\mathsf{rss}}$, $\mathsf{MACAO}_{\mathsf{spdz}}$ is 33.23 MB. With 1 TB DB and 4 KB block size, it is 1.33 GB.

4.5.3.6 Performance with Varying Privacy Levels

We conducted an experiment to evaluate the performance of $\mathsf{MACAO}_{\mathsf{spdz}}$ and $\mathsf{MACAO}^{\mathsf{prf}}_{\mathsf{spdz}}$ schemes under higher privacy levels by increasing the number of servers. Due to RSS, we did not evaluate $\mathsf{MACAO}_{\mathsf{rss}}$ and $\mathsf{MACAO}^{\mathsf{prf}}_{\mathsf{rss}}$ since they incur significant server storage for high privacy levels. Figure 4.38 outlines the delay of $\mathsf{MACAO}_{\mathsf{spdz}}$ and $\mathsf{MACAO}^{\mathsf{prf}}_{\mathsf{spdz}}$ scheme under 1 TB database with 4 KB and 256 KB block sizes. When increasing the number of servers, $\mathsf{MACAO}^{\mathsf{prf}}_{\mathsf{spdz}}$ incurred much less delay than $\mathsf{MACAO}_{\mathsf{spdz}}$, especially in the 256 KB block size setting, since the client only sent data to one server while the other servers generated authenticated shares on their own.

4.6 Oblivious Data Structures

Although ORAM can seal the access pattern leakage in any application, applying it straightforwardly may incur high communication, computation and monetary cost. On the other hand, there are unique characteristics in certain applications that we can exploit to design more efficient oblivious access alternatives. In this section, we propose several oblivious data structures, which permit oblivious access protocols (*e.g.*, ORAM, PIR) to be deployed more efficiently in the context of searchable encryption and database services.

4.6.1 Distributed Data Structure for Oblivious Searchable Encryption

We propose a new oblivious access scheme over the encrypted index in DSSE that we call Distributed Oblivious Data structure (DOD-DSSE). Our intuition is to leverage two non-colluding servers and exploit the properties of an incidence matrix to seal information leakages in DSSE, while incurring only a small-constant overhead. DOD-DSSE achieves the following desirable properties:

- *High security*: DOD-DSSE seals leakages from search and update operations on the encrypted index and, therefore, it offers much higher security than traditional DSSE schemes. Specifically, DOD-DSSE breaks the linkability between access operations on the encrypted index **I**, hides search (i) and update (ii) patterns. This allows DOD-DSSE to prevent a server from learning whether or not the same query is continuously repeated as well as the relationship between keywords and files. Therefore, DOD-DSSE is secure against statistical attacks to which all traditional DSSE schemes are vulnerable due to leakages from the access patterns on the encrypted index. Table 4.5 summaries the security of DOD-DSSE, compared with traditional DSSEs.

- *High efficiency*: DOD-DSSE offers much lower bandwidth overhead and delay than applying ORAM-based techniques (*e.g.*, ODS [182]) to the encrypted data structure $\mathbf{I}$. Our experimental results indicate that DOD-DSSE is much faster than using ODS with Path ORAM protocol on dictionary[12] and incidence matrix[13] data structures, respectively, in terms of end-to-end delay (Table 4.5). DOD-DSSE only takes around one second to perform an access operation on a very large data structure (see §4.6.1.4 for a detailed comparison).

- *Formal security analysis and full-fledged implementation*: We fully analyze the security and information leakages of DOD-DSSE (§4.6.1.3). We provide a detailed implementation of DOD-DSSE on two virtual Amazon EC2 servers and strictly evaluated the performance of DOD-DSSE on real network settings (§4.6.1.4). *We also released the implementation of* DOD-DSSE *for public use*[14].

These properties make DOD-DSSE an ideal alternative for privacy-critical cloud applications. We briefly describe the main idea of *DOD-DSSE* as follows:

Existing DSSE schemes rely on a deterministic association between the (address) token of a query and its corresponding encrypted result in the DSSE data structure. In other words, each query x is represented by a deterministic address token-data tuple $(u_x, \mathbf{I}_{u_x})$ in the encrypted data structure $\mathbf{I}$. Despite permitting consistent and fast search/update operations, these deterministic relations leak

[12] Dictionary is a $\langle \mathsf{key}, \mathsf{value} \rangle$ structure such that given a keyword key, its corresponding value is a list of file IDs in which key appears. This data structure offers sublinear search time, but leaks information due to its size depending on the file IDs associated with key.

[13] Incidence matrix is a data structure which represents the relationship between keywords (indexing rows) and files (indexing columns) via its cell value. For example, if matrix entry $\mathbf{I}[i, j]$ is set to 1, it means the keyword indexing the ith row appears in the file indexing the jth column. Similarly, the $\mathbf{I}[i, j]$ entry is set to 0 if the keyword indexing the ith row does not appear in the file indexing column j.

[14] Available at https://github.com/thanghoang/DOD-DSSE/

the access pattern on the encrypted index defined in Definition 2. The research challenge is to devise cryptographic methods that can create a random uniform address token-data tuple $(u_x, \mathbf{I}_{u_x})$ for each query x in an oblivious way with just a small number of communication rounds and processing time. DOD-DSSE achieves this by using a "*fetch–reencrypt–swap*" strategy between two servers as follows:

First, the client creates two encrypted data structures, each including address-data tuples $(u_x, \mathbf{I}_{u_x})$ of all possible search and update queries, and then sends them to two non-colluding servers (S_0, S_1), respectively. To perform a search or update operation, the client sends a search query and an update query to each server. One query is for the real operation while the other three are randomly selected (fake) queries (Figure 4.39, step (1)). Each server sends back to the client the corresponding address-data tuples that have been queried. After that, the client decrypts the received data to obtain the result (step (2)), and then re-encrypts them (step (3)). The client creates new address-data tuple for each performed query by assigning re-encrypted data to a random address (step (4)). Finally, the client swaps such address-data tuples and writes them back to the other server (step (5)). That means the new address-data tuple of the query being read from server S_0 will be written to server S_1 and vice versa. This strategy makes each server observe a randomized data structure-access pattern with only one-time repetition of a *unlinkable query* that has been performed on the other server, provided that the two servers do not collude. §4.6.1.2 presents detailed constructions of DOD-DSSE.

There are two limitations in our design. First, we assume that the two servers storing the encrypted data structures are non-colluding; Second, DOD-DSSE leaks to each server S_b $(b = 0, 1)$ a one-time repetition of a query that was previously performed on the other server. This query cannot be linked to any other queries performed on S_b and it *never repeats* on S_b again.

Table 4.5: Security and performance of DOD-DSSE *vs.* its counterparts.

Scheme	Security					Performance	Setting
	Data structure-access pattern		Update leakage	Query result size	Statistical attacks	End-to-end crypto delay†	# Server
	1-time repetition of an unlinkable query	Full query linkability§					
Traditional DSSE [109],[138],[35],[189]	✗	✗	✗	✗	✗	< 0.2 s	1
ODICT	✓	✓	✗*	✗*	✓	192.2 s*	1
OMAT	✓	✓	✓	✓	✓	767.5 s	1
DOD-DSSE	✗‡	✓	✓	✓	✓	**1.1 s**	**2**

We simulated the cost of ODS [182] with Path-ORAM protocol [169] on dictionary (ODICT) and incidence matrix (OMAT) data structures.

† The delays of schemes were measured in our experiment with an average network latency of 31 ms and throughput of 30 Mbps.

‡ This leakage does not lead to any statistical attacks.

* Due to the sublinear operation time of dictionary data structure, ODS cannot fully hide the length of search/update result without fully padding which is very costly. To evaluate the performance of ODICT over the real network, we only simulated ODICT with average padding.

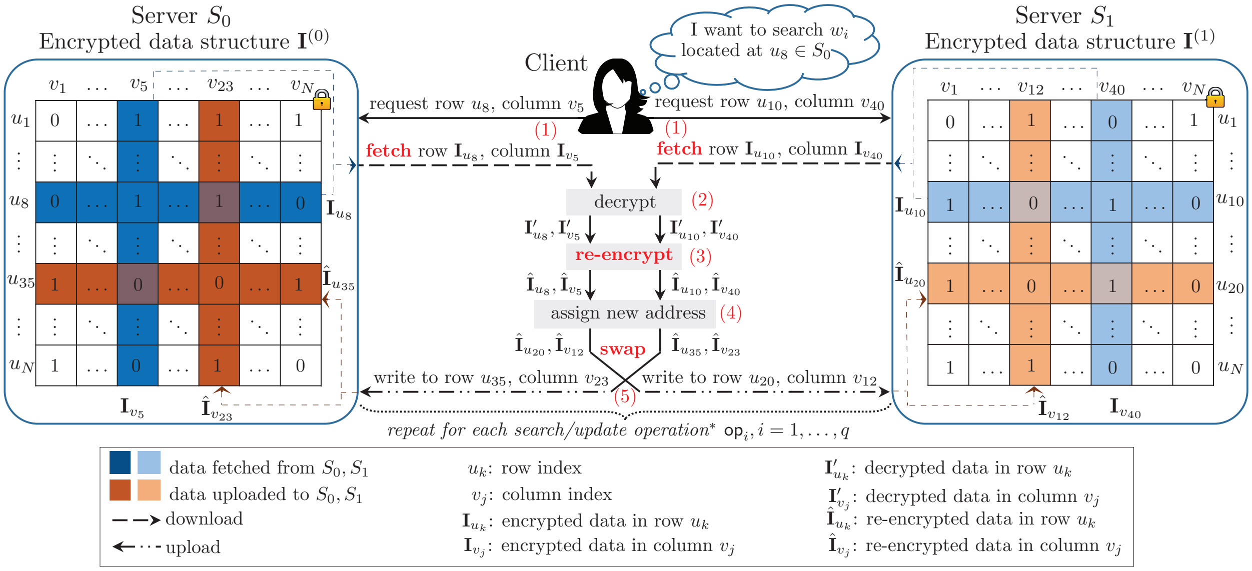
§ Full query linkability allows the adversary to perform, for example, frequency analysis [124], resulting in statistical attacks.

We note that the performance and security benefits of DOD-DSSE well-justifies these limitations. Furthermore, (i) two practical non-colluding servers can be found in real world as competitive cloud providers such as Amazon, Microsoft and Google are very unlikely to collude against their client. (ii) Indeed, we show that with minimal information leakage (*i.e.*, one-time repetition of an unknown and unlinkable query on the other server), DOD-DSSE seals all search/update patterns, prevents statistical attacks which are main objectives of a secure DSSE. At the same time, it achieves extremely efficient performance compared to using ORAM-based techniques. Therefore, DOD-DSSE offers an ideal security-performance trade-off for DSSE.

4.6.1.1 System and Security Models

Our system model comprises a client and two servers $\mathcal{S} = (S_0, S_1)$, each storing an instance of an encrypted data structure created from the same file collection.

Assumption 1. Servers communicate with the client via private channels. (i) (S_0, S_1) are honest-but-curious, meaning that they show interest in learning information but follow the protocol faithfully; they do not inject malicious inputs to the system. (ii) *S_0 and S_1 do not collude.*


Server S_0
Encrypted data structure $\mathbf{I}^{(0)}$
Client
I want to search w_i located at $u_8 \in S_0$
Server S_1
Encrypted data structure $\mathbf{I}^{(1)}$
v_1 … v_5 … v_{23} … v_N
u_1 ⋮ u_8 ⋮ u_{35} ⋮ u_N
$\mathbf{I}_{u_8}$
$\hat{\mathbf{I}}_{u_{35}}$
$\mathbf{I}_{v_5}$
$\hat{\mathbf{I}}_{v_{23}}$
v_1 … v_{12} … v_{40} … v_N
u_1 ⋮ u_{10} ⋮ u_{20} ⋮ u_N
$\mathbf{I}_{u_{10}}$
$\hat{\mathbf{I}}_{u_{20}}$
$\hat{\mathbf{I}}_{v_{12}}$
$\mathbf{I}_{v_{40}}$
request row u_8, column v_5
request row u_{10}, column v_{40}
(1)
(1)
fetch row $\mathbf{I}_{u_8}$, column $\mathbf{I}_{v_5}$
fetch row $\mathbf{I}_{u_{10}}$, column $\mathbf{I}_{v_{40}}$
decrypt (2)
$\mathbf{I}'_{u_8}, \mathbf{I}'_{v_5}$
$\mathbf{I}'_{u_{10}}, \mathbf{I}'_{v_{40}}$
re-encrypt (3)
$\hat{\mathbf{I}}_{u_8}, \hat{\mathbf{I}}_{v_5}$
$\hat{\mathbf{I}}_{u_{10}}, \hat{\mathbf{I}}_{v_{40}}$
assign new address (4)
$\hat{\mathbf{I}}_{u_{20}}, \hat{\mathbf{I}}_{v_{12}}$
swap
$\hat{\mathbf{I}}_{u_{35}}, \hat{\mathbf{I}}_{v_{23}}$
write to row u_{35}, column v_{23}
write to row u_{20}, column v_{12}
(5)
repeat for each search/update operation* $\mathsf{op}_i, i = 1, \ldots, q$
data fetched from S_0, S_1
data uploaded to S_0, S_1
download
upload
u_k: row index
v_j: column index
$\mathbf{I}_{u_k}$: encrypted data in row u_k
$\mathbf{I}_{v_j}$: encrypted data in column v_j
$\mathbf{I}'_{u_k}$: decrypted data in row u_k
$\mathbf{I}'_{v_j}$: decrypted data in column v_j
$\hat{\mathbf{I}}_{u_k}$: re-encrypted data in row u_k
$\hat{\mathbf{I}}_{v_j}$: re-encrypted data in column v_j

*This strategy provides the *unlinkability between queries* as the same keyword/file generates different queries to each server. Multiple repetitions of the same operation cannot be observed if S_0 and S_1 are non-colluding. Hence, it prevents leakage from search & update patterns and invalidates statistical attacks.

Figure 4.39: Search/update operation on the encrypted index in DOD-DSSE.

In DOD-DSSE, data request sequences $(\vec{\sigma_0}, \vec{\sigma_1})$ of length q by Definition 2 are independently observed by servers (S_0, S_1), respectively. We assume that the encrypted data structure can store up to N distinct data items, each corresponding with either a search or an update query. Each item is represented by a unique address-data tuple (u, data) in the data structure. The security of DOD-DSSE scheme relies on the fact that any access operations $\mathsf{op}_i^{(b)} \in \vec{\sigma_b}$ observed by server S_b, for all $1 \leq i \leq q$ are guaranteed to be unlinkable. This achievement enables us to protect the data structure-access pattern in each server as defined in Definition 2. We define the unlinkability property of access operations on the encrypted data structure in DSSE as follows:

Definition 12. Let $(u_i^{(b)}, \mathsf{data}_i^{(b)}), (u_j^{(b)}, \mathsf{data}_j^{(b)})$ be address-data tuples requested by access operations $(\mathsf{op}_i^{(b)}, \mathsf{op}_j^{(b)}) \in \vec{\sigma_b}$ observed by server S_b, respectively. $\mathsf{op}_i^{(b)}$ is called unlinkable to $\mathsf{op}_j^{(b)}$ if the probability that $\langle (u_i^{(b)}, \mathsf{data}_i^{(b)}), (u_j^{(b)}, \mathsf{data}_j^{(b)}) \rangle$ represent the same item being accessed is $\frac{1}{N}$, where N is the number of distinct items stored in S_b.

Note that this is the upper bound of linkability probability that one can infer from two arbitrary tuples. The unlinkability in Definition 12 implies the DOD-DSSE security definition, which is comparable to that of ORAM as follows:

Definition 13. DOD-DSSE on the server S_b leaks no information beyond ORAM (Definition 5) with the exception of one-time repetition of an unknown and unlinkable query on the other server $S_{\neg b}$, to which S_b does not have access.

We will give a detailed security analysis in §4.6.1.3 after presenting the construction of DOD-DSSE in the following section.

4.6.1.2 The Proposed DOD-DSSE *Scheme*

We first describe the encrypted data structure used in DOD-DSSE, followed by several newly proposed algorithms.

We first assume that f and w denote a file and a keyword, respectively. m and n denote the maximum number of files and keywords in the dataset, respectively. $\mathbf{f} = (f_{id_1}, \ldots, f_{id_m})$ denotes the collection of files. An encrypted data structure enables encrypted search and update operations for a keyword w or a file f_{id}. We adopt a keyword-file *incidence matrix* to be the DSSE data structure $\mathbf{I}$ due to its security and performance advantages, compared with other typical types such as multi-linked list [109], dictionary [138], and tree [108]. For the sake of simplicity, we assume that keywords are assigned to row indices while file IDs are assigned to column indices. We assume our file collection $\mathbf{f}$ consists of $m' \leq N$ unique keywords and $n' \leq N$ file IDs, where N is the maximum number of unique keywords and files that our $\mathbf{I}$ can support. We construct the encrypted index $\mathbf{I}^{(0)}$ for server S_0 and $\mathbf{I}^{(1)}$ for S_1 as follows:

First, we assign each item x, where x is a keyword or file ID, to a unique random address in $\mathbf{I}^{(b)}$ as $u_x^{(b)} \xleftarrow{\$} \mathcal{L}_b$ for each $b \in \{0,1\}$, where $\mathcal{L}_b$ is the set of unassigned row or column indices in $\mathbf{I}^{(b)}$. For security reasons which will be analyzed in §4.6.1.3, $|\mathcal{L}|= 2N$. In other words, $\mathbf{I}$ *is a square matrix of size* $2N \times 2N$ to cover N keywords and files.

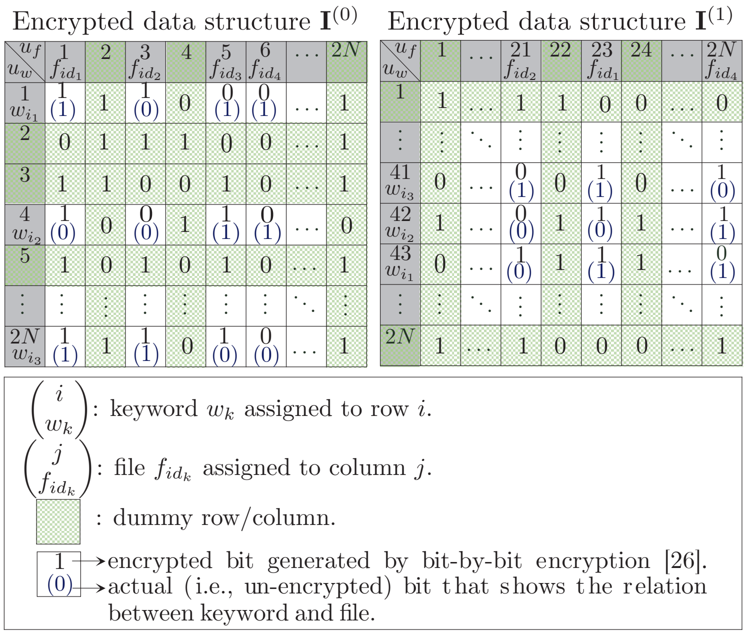
We represent the relationship between a keyword and a file by a cell value in $\mathbf{I}^{(b)}$. $\mathbf{I}^{(b)}[i,j] = 1$ means the keyword w assigned to row i appears in the file assigned to column j in server S_b and $\mathbf{I}^{(k)}[i,j] = 0$ otherwise. We can consider the data $\mathbf{I}_{u_x}^{(b)}$ of item x as a row or column data which is a binary string of length $2N$ representing the relationship between x and its object in server S_b. A search or update query of x will correspond with retrieving a whole row or column respectively.

Finally, we encrypt every cell in $\mathbf{I}^{(b)}$ using bit-by-bit IND-CPA encryption scheme with a counter and a key as $\mathbf{I}^{(b)}[i,j] \leftarrow \mathsf{Enc}\tau_i^{(b)}\mathbf{I}^{(b)}[i,j], c_j^{(b)}$, where $c_j^{(b)}$ is a counter derived from column index j and a value $a_j^{(b)}$ in server S_b and $\tau_i^{(b)}$ is a row key derived from the row index i and a secret key generated for server S_b. We store the information of $a_j^{(b)}$ in global counter arrays $c^{(b)}$ of length $2N$ which can be retrieved as $a_j^{(b)} \leftarrow c^{(b)}[j]$, for each $b \in \{0,1\}$. We describe detailed constructions of $\mathbf{I}^{(0)}, \mathbf{I}^{(1)}$ in Figure 4.41. Figure 4.40 depicts the structure and content of $\mathbf{I}^{(0)}$ and $\mathbf{I}^{(1)}$.

We create data structures T_w, T_f stored on the client-side for keywords and files, respectively which are defined as:

$$T : (H(x), \langle u_x^{(0)}, u_x^{(1)}, b_x \rangle).$$

(T_w, T_f) are used to keep track of the assigned addresses $(u_x^{(0)}, u_x^{(1)})$ of each item x on servers (S_0, S_1), respectively, as well as the server ID (*i.e.*, $b_x \in \{0,1\}$) where it was last accessed. We define functions for T as follows:



Encrypted data structure $\mathbf{I}^{(0)}$

u_f u_w	1 f_{id_1}	2	3 f_{id_2}	4	5 f_{id_3}	6 f_{id_4}	$\ldots$	$2N$
1 w_{i_1}	1 (1)	1	1 (0)	0	0 (1)	0 (1)	$\ldots$	1
2	0	1	1	1	0	0	$\ldots$	1
3	1	1	0	0	1	0	$\ldots$	1
4 w_{i_2}	1 (0)	0	0 (0)	1	1 (1)	0 (1)	$\ldots$	0
5	1	0	1	0	1	0	$\ldots$	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$2N$ w_{i_3}	1 (1)	1	1 (1)	0	1 (0)	0 (0)	$\ldots$	1

Encrypted data structure $\mathbf{I}^{(1)}$

u_f u_w	1	$\ldots$	21 f_{id_2}	22	23 f_{id_1}	24	$\ldots$	$2N$ f_{id_4}
1	1	$\ldots$	1	1	0	0	$\ldots$	0
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
41 w_{i_3}	0	$\ldots$	0 (1)	0	1 (1)	0	$\ldots$	1 (0)
42 w_{i_2}	1	$\ldots$	0 (0)	1	1 (0)	1	$\ldots$	1 (1)
43 w_{i_1}	0	$\ldots$	1 (0)	1	1 (1)	1	$\ldots$	0 (1)
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$2N$	1	$\ldots$	1	0	0	0	$\ldots$	1

$\begin{pmatrix} i \\ w_k \end{pmatrix}$: keyword w_k assigned to row i.

$\begin{pmatrix} j \\ f_{id_k} \end{pmatrix}$: file f_{id_k} assigned to column j.

▨ : dummy row/column.

1 → encrypted bit generated by bit-by-bit encryption [26].
(0) → actual (i.e., un-encrypted) bit that shows the relation between keyword and file.

Figure 4.40: Distributed encrypted index generated from the file collection.

- T.insert(key, value): insert hash of x (*i.e.*, $H(x)$) as key and x's information $\langle u_x^{(0)}, u_x^{(1)}, b_x \rangle$ as value into T. It can accept null as key, in which the value (i.e, $\langle u_{\mathsf{null}}^{(0)}, u_{\mathsf{null}}^{(1)}, b_{\mathsf{null}} \rangle$) will be inserted into empty slots in T.

- $j_x \leftarrow T.\mathsf{get}(H(x))$: find the index j_x of x in T using its hash value $H(x)$.

- $j_x \leftarrow T.\mathsf{lookup}(u_x^{(b)}, b)$: find the index j_x of x in T using its address u_x on the server S_b.

Information about x can be retrieved via its index j_x in T as $(H(x), \langle u_x^{(0)}, u_x^{(1)}, b_x \rangle) \leftarrow T[j_x]$.

We can see that $\mathbf{I}$ is a $2N \times 2N$ matrix storing the relationship between N unique keywords and N files. There are at least N empty rows and N empty columns in $\mathbf{I}$. We also include in T_w, T_f sets of such "dummy" addresses in $\mathbf{I}^{(0)}, \mathbf{I}^{(1)}$, denoted as $T_w.\mathcal{L}_0, T_w.\mathcal{L}_1$, for keywords and $T_f.\mathcal{L}_0, T_f.\mathcal{L}_1$ for files respectively. This is to achieve the consistency of DOD-DSSE data structure and security (see §4.6.1.3).

```
$(K, T_w, T_f, \mathbf{I}^{(0)}, \mathbf{I}^{(1)}) \leftarrow$ DOD-DSSE.Init$(\kappa, \mathbf{f})$:
Initialization: Set $T_w, T_f$ to be empty
$\mathbf{I}'^{(b)}[i,j] \leftarrow 0$ and $c^{(b)}[j] \leftarrow 1$, for all $1 \leq i \leq 2N$, $1 \leq j \leq 2N$ and for each $b \in \{0,1\}$
$T_x.\mathcal{L}_b \leftarrow \{1, 2, \ldots, 2N\}$ for each $x \in \{w, f\}$ and $b \in \{0,1\}$
$K \leftarrow$ DOD-DSSE.Gen$(1^\kappa)$
Extract unique keywords $\mathbf{w} = (w_1, \ldots, w_{m'})$ from files $\mathbf{f} = (f_{id_1}, \ldots, f_{id_{n'}})$
# Create unecnrypted data structure $\mathbf{I}'^{(0)}$ for server $S_0$, $\mathbf{I}'^{(1)}$ for server $S_1$
for each $w_i \in \{w_1, \ldots, w_{m'}\}$ do
    $(u_i^{(0)}, u_i^{(1)}, T_w) \leftarrow$ DOD-DSSE.Assign$(H(w_i), T_w)$
    for each $id_j \in \{id_1, \ldots, id_{n'}\}$ do
        $(v_j^{(0)}, v_j^{(1)}, T_f) \leftarrow$ DOD-DSSE.Assign$(H(id_j), T_f)$
        if $w_i$ appears in $f_{id_j}$ then
            $\mathbf{I}'^{(0)}[u_i^{(0)}, v_j^{(0)}] \leftarrow 1$, $\mathbf{I}'^{(1)}[u_i^{(1)}, v_j^{(1)}] \leftarrow 1$
# Reserve row and column indices for new keywords and files being added in the future
Call DOD-DSSE.Assign$(\mathsf{null}, T_x)$ multiple times until all values in all $N$ slots in $T_x$ are filled, for each $x \in \{w, f\}$
# Encrypt every row of data structures $\mathbf{I}'^{(0)}$ and $\mathbf{I}'^{(1)}$
for each server $S_b \in \{S_0, S_1\}$ do
    $\mathbf{I}^{(b)}[i, *] \leftarrow$ DOD-DSSE.Enc$(\mathbf{I}'^{(b)}[i, *], i, b, K)$ for each row $i \in \{1, \ldots, 2N\}$
return $(K, T_w, T_f, \mathbf{I}^{(0)}, \mathbf{I}^{(1)})$
```

Figure 4.41: DOD-DSSE setup algorithm.

We present detailed implementations of DOD-DSSE in three main algorithms with four subroutines. We provide in Figure 4.46 the decryption procedure for a row/column data of **I**. The encryption procedure DOD-DSSE.Enc() is not explicitly defined and it works similarly by substituting Dec for Enc in lines 4 and 8 of Figure 4.46.

The main operation of DOD-DSSE is presented in Figure 4.42. First, the client generates for each server one search and one update queries including two row indices (one is dummy) and two column indices (one is dummy) as in Figure 4.45 (step 1). The client reads data in such addresses from the servers and decrypts only data in non-dummy addresses (steps 2 – 7). After that, the client can perform an actual search or update operation over the decrypted data (steps 8–11). Finally, the decrypted data are re-encrypted with new counters and written back to addresses in the server from where they were read as well as the dummy addresses in the other server (steps 17–21). Notice that such data need to be updated before re-encryption to preserve keyword-file relations (steps 15–16),

```
$\mathbf{id} \leftarrow \mathsf{DOD\text{-}DSSE.Access}(\mathsf{op}, x, \mathbf{S}, K, T_w, T_f)$:
    # Generate search and update queries
1:  $(\{u_j^{(b)}, ut_j^{(b)}\}_{j \in \{w,f\}, b \in \{0,1\}}, \beta) \leftarrow \mathsf{DOD\text{-}DSSE.CreateQueries}(H(x))$

2:  for each $j \in \{w, f\}$ do
        # Retrieve from each server 2 columns or 2 rows depending on index $j$
3:      for each server $S_b \in \{S_0, S_1\}$ do
4:          $\mathbf{I}_{u_j}^{(b)} \leftarrow \mathsf{Read}(u_j^{(b)})$ from $S_b$
5:          $\mathbf{I}_{ut_j}^{(b)} \leftarrow \mathsf{Read}(ut_j^{(b)})$ from $S_b$

        # Decrypt retrieved row and column
6:      $\mathbf{I'}_{u_j}^{(b)} \leftarrow \mathsf{DOD\text{-}DSSE.Dec}(\mathbf{I}_{u_j}^{(\beta)}, u_j^{(\beta)}, \beta, K)$
7:      $\mathbf{I'}_{ut_j}^{(\neg b)} \leftarrow \mathsf{DOD\text{-}DSSE.Dec}(\mathbf{I}_{ut_j}^{(\neg\beta)}, ut_j^{(\neg\beta)}, \neg\beta, K)$

8:  if $\mathsf{op} = \mathsf{search}$ then
9:      Extract column index from $\mathbf{I'}_{u_w}^{(\beta)}$ as $\mathbf{id} \leftarrow (y_1, \ldots, y_l)$, where $\mathbf{I'}_{u_w}^{(\beta)}[y_k] = 1$ for each $y_k \in \{1, \ldots, 2N\} \setminus T_f.\mathcal{L}_\beta$,
    $1 \leq k \leq l$
10: else
11:     Update list of keywords in $\mathbf{I'}_{u_f}^{(\beta)}$ and key in $T_f$, $T_w$ corresponding with the file being updated

12: for each $j \in \{w, f\}$ do
        # Update new address of non-dummy column and row data in the other server
13:     $T_j \leftarrow \mathsf{DOD\text{-}DSSE.UpdateT}(T_j, u_j^{(\beta)}, \beta, u_j^{(\neg)})$
14:     $T_j \leftarrow \mathsf{DOD\text{-}DSSE.UpdateT}(T_j, ut_j^{(\neg\beta)}, \neg\beta, ut_j^{(\beta)})$
15:     Update cell values in $\mathbf{I'}_{u_j}^{(\beta)}$ and $\mathbf{I'}_{ut_j}^{(\neg\beta)}$ to preserve keyword-file relations with changes at steps 13, 14
16:     Create $(\mathbf{I'}_{u_j}^{(\neg\beta)}, \mathbf{I'}_{u_j}^{(\neg\beta)})$ based on $\mathbf{I'}_{u_j}^{(\beta)}$, $(\mathbf{I'}_{ut_j}^{(\neg\beta)})$ respectively to preserve keyword-file relation consistency
    in both servers

    # Increase all counter values in global counter arrays
17: $c^{(b)}[i] \leftarrow c^{(b)}[i] + 1$ for each $b \in \{0, 1\}$ and $i \in \{1, \ldots, 2N\}$
18: for each $j \in \{w, f\}$ do
19:     for each server $S_b \in \{S_0, S_1\}$ do
            # Re-encrypt retrieved data with newly updated counters
20:         $\hat{\mathbf{I}}_{u_j}^{(b)} \leftarrow \mathsf{DOD\text{-}DSSE.Enc}(\mathbf{I'}_{u_j}^{(b)}, u_j^{(b)}, b, K)$, $\hat{\mathbf{I}}_{ut_j}^{(b)} \leftarrow \mathsf{DOD\text{-}DSSE.Enc}(\mathbf{I'}_{ut_j}^{(b)}, ut_j^{(b)}, b, K)$
            # Write re-encrypted data back to corresponding server $S_b$
21:         $\mathsf{Write}(u_j^{(b)}, \hat{\mathbf{I}}_{u_j}^{(b)})$, $upData(ut_j^{(b)}, \hat{\mathbf{I}}_{ut_j}^{(b)})$ to $S_b$
22: return id
```

Figure 4.42: DOD-DSSE access protocol.

and their new addresses in the other server are updated in hash tables (steps 12–16) so that they can be retrieved correctly in subsequent operations.

4.6.1.3 Security Analysis

Let $(\overrightarrow{\sigma_0}, \overrightarrow{\sigma_1})$ be a query sequence of length q sent to servers (S_0, S_1) respectively. By Definition 2, the access patterns $\langle \mathbf{AP}_0(\overrightarrow{\sigma_0}), \mathbf{AP}_1(\overrightarrow{\sigma_1}) \rangle$ observed by (S_0, S_1), respectively, are:

```
$K \leftarrow \mathsf{DOD\text{-}DSSE.Gen}(1^\kappa)$:
  # Generate keys to encrypt two data structures
$k_0 \leftarrow \mathcal{E}.\mathsf{Gen}(1^\kappa)$, $k_1 \leftarrow \mathcal{E}.\mathsf{Gen}(1^\kappa)$
return $K \leftarrow (k_0, k_1)$
```

Figure 4.43: DOD-DSSE key generation algorithm.

```
$(u_x^{(0)}, u_x^{(1)}, T) \leftarrow \mathsf{DOD\text{-}DSSE.Assign}(x, T)$
  # Pick a random address from dummy set in each server
for each $b \in \{0,1\}$ do
    $u_x^{(b)} \xleftarrow{\$} T.\mathcal{L}_b$
    $T.\mathcal{L}_b \leftarrow T.\mathcal{L}_b \setminus \{u_x^{(b)}\}$

  # Randomly assign server ID for $x$
$b_x \xleftarrow{\$} \{0,1\}$
  # Store assigned info of $x$ in hash table
$T.\mathsf{insert}\left(x, \langle u_x^{(0)}, u_x^{(1)}, b_x\rangle\right)$
return $(u_x^{(0)}, u_x^{(1)}, T)$
```

Figure 4.44: DOD-DSSE assign subroutine.

$$\begin{aligned}\mathbf{AP}_0 &= \{\mathsf{access}(x_1^{(0)}), \ldots, \mathsf{access}(x_i^{(0)}), \ldots, \mathsf{access}(x_q^{0})\}\\ \mathbf{AP}_1 &= \{\mathsf{access}(x_1^{(1)}), \ldots, \mathsf{access}(x_i^{(1)}), \ldots, \mathsf{access}(x_q^{(1)})\},\end{aligned} \tag{4.16}$$

$\mathsf{access}(x_i^{(b)}) = (\{\mathsf{read}(u_{j,ti}^{(b)}, \mathbf{I}_{u_{j,ti}}^{(b)})\}, \{\mathsf{write}(u_{j,ti}^{(b)}, \hat{\mathbf{I}}_{u_{j,ti}}^{(b)})\})$, for $j \in \{w, f\}$, $1 \leq t \leq 2$, $b \in \{0,1\}$ and $1 \leq i \leq q$ performing read-then-write operations on the server S_b, given a DOD-DSSE.Access operation op_i (Figure 4.42) at step i. Each address-data tuple $(u_{j,ti}^{(b)}, \mathbf{I}_{u_{j,ti}}^{(b)})$ comprises a random row or column address and an IND-CPA encryption output, respectively.

Remark 3. Due to the properties of the square incidence matrix data structure, rows and columns intersect each other and have the same length. For each actual operation, DOD-DSSE *performs a search query and an update query to each server S_b. This prevents S_b from determining (i) if the actual intention of the client is to search or to update, and (ii) which data-address tuple corresponds with search or update query. These properties prevent S_b from separately forming search and update patterns defined in Definition 3.*

```
$(\mathcal{U}, \underline{)} \leftarrow \mathsf{CreateQueries}(x)$:
  # Get hash table entry for $x$ and its info
$j_x \leftarrow T_x.\mathsf{get}(x)$
$\beta \leftarrow T_x[j_x].b_x$
$u_x^{(\underline{\ })} \leftarrow T_x[j_x].u_x^{(\beta)}$
  # Select a random non-dummy row/column index $u_{\bar{x}}^{(\beta)}$.
  # If $x$ is $w$ then $\bar{x}$ is $f$ and vice-versa
$u_{\bar{x}}^{(\underline{\ })} \overset{\$}{\leftarrow} \{1, \ldots, 2N\} \setminus T_{\bar{x}}.\mathcal{L}_\beta$
for each $j \in \{w, f\}$ do
    # Select a random non-dummy index from server $S_{\neg\beta}$
    $u_j^{(\neg\beta)} \overset{\$}{\leftarrow} \{1, \ldots, 2N\} \setminus T_j.\mathcal{L}_{\neg\beta}$
    # Randomly select dummy row & column indices in $S_0$, $S_1$
    for each $b \in \{0, 1\}$ do
        $ut_j^{(b)} \overset{\$}{\leftarrow} T_j.\mathcal{L}_b$
return $(\mathcal{U}, \beta)$, where $\mathcal{U} = \{u_j^{(b)}, ut_j^{(b)}\}_{j \in \{w,f\}, b \in \{0,1\}}$
```

Figure 4.45: DOD-DSSE query creation subroutine.

```
$\mathbf{I}'_u \leftarrow \mathsf{DOD\text{-}DSSE.Dec}(\mathbf{I}_u, u, b, K)$:
if $u$ is a row index then
    $\tau_u^{(b)} \leftarrow \mathsf{KDF}(k_b || u)$
    for $j = 1 \ldots, 2N$ do
        $\mathbf{I}'_u[j] \leftarrow \mathsf{Dec}\tau_u^{(b)} \mathbf{I}_u[j], j || c^{(b)}[j]$
else        # if $u$ is a column index
    for $i = 1 \ldots, 2N$ do
        $\tau_i^{(b)} \leftarrow \mathsf{KDF}(k_b || i)$
        $\mathbf{I}'_u[i] \leftarrow \mathsf{Dec}\tau_i^{(b)} \mathbf{I}_u[i], u || c^{(b)}[u]$
return $\mathbf{I}'$
```

Figure 4.46: DOD-DSSE decryption subroutine.

Remark 4. Data items associated with search and update queries are located in two independent address spaces (i.e., row index vs. *column index) and, therefore, their access operations are independent from each other. For the sake of brevity, we only analyze the security of search queries. The same analysis can be applied to update queries. From now on, whenever we say data* $\mathbf{I}_{u_x}$ *of the query* x *at address* u_x*, we mean the row data corresponding with the search query along with its row index.*

According to the unlinkability definition (Definition 12) and DOD-DSSE access scheme in Figure 4.42, we define in Definition 14 the unlinkability property of a data item which is read from

```
$\underline{T \leftarrow \textsf{DOD-DSSE.UpdateT}(T, qIdx, b, nIdx)}$:
    # Get hash table entry for $qIdx$ in $S_b$
    $j_x \leftarrow T.\textsf{lookup}(qIdx, b)$

    # Update hash table with new entry $nIdx$ and server $b$
    $oIdx \leftarrow T[j].u_x^{(\neg b)}$
    $T[j_x].u_x^{(\neg b)} \leftarrow nIdx$
    $T[j_x].b_x \leftarrow \neg b$

    # Remove $nIdx$ from dummy set $T.\mathcal{L}_{\neg b}$ and add $oIdx$ to it
    $T.\mathcal{L}_{\neg b} \leftarrow T.\mathcal{L}_{\neg b} \cup \{oIdx\} \setminus \{nIdx\}$
    return $T$
```

Figure 4.47: DOD-DSSE client state update subroutine.

server $S_{\neg b}$, and its new representation is then written to S_b under S_b's view. We then show in Lemma 9 that any access patterns observed by servers (S_0, S_1) in our scheme are unlinkable to each other by Definition 14 under Lemma 9. Finally, we prove that DOD-DSSE achieves our main security notion (Definition 13) in Theorem 6.

Definition 14. Let $(u_x^{(b)}, \mathbf{I}_{u_x}^{(b)})$ represent an item x in a set $\mathcal{I}^{(b)}$ of N distinct data items on server S_b such that $u_x^{(b)} \neq u_{x'}^{(b)}$ and $\mathbf{I}_{u_x}^{(b)} \neq \mathbf{I}_{u_{x'}}^{(b)}$ for each $x', x \in \mathcal{I}^{(b)}$ and $x \neq x'$. $(ut_{x''}^{(b)}, \hat{\mathbf{I}}_{ut_{x''}}^{(b)}) \in \mathbf{AP}_b$ (as in (Equation 4.16)) is a new representation of an arbitrary data item $x'' \in \mathcal{D}$, which has just been accessed on server $S_{\neg b}$. In DOD-DSSE, $(ut_{x''}^{(b)}, \hat{\mathbf{I}}_{ut_{x''}}^{(b)})$ is unlinkable to $\mathcal{I}^{(b)}$ if and only if the probability that $(ut_{x''}^{(b)}, \hat{\mathbf{I}}_{ut_{x''}}^{(b)})$ represents the same item with any tuples $(u_x^{(b)}, \mathbf{I}_{u_x}^{(b)})$ for each $x \in \mathcal{I}^{(b)}$ is $\frac{1}{N}$.

Lemma 9. Under 1, any access patterns observed by S_b and $S_{\neg b}$ as in (Equation 4.16) are unlinkable with each other by Definition 12.

Proof. For each DOD-DSSE operation x_i (Figure 4.42), server S_b observes that two address-data tuples are accessed per search query simultaneously. One of them is to read while the other is to write data being read from $S_{\neg b}$. The data from all accessed addresses are IND-CPA re-encrypted with new counters before being written back (Figure 4.42, steps 17, 20) so that it is computationally indistinguishable for S_b to determine which address is being read or written. To begin with, we show that $\mathsf{access}(x_i^{(b)})$ is unlinkable to $\mathsf{access}(x_i^{(\neg b)})$ as follows:

We first analyze the address-data tuple denoted as $(u^{(b)}, \mathbf{I}_u^{(b)})$, which is read and observed by S_b. $\mathbf{I}_u^{(b)}$ is decrypted into $\mathbf{I'}_u^{(b)}$ and then is IND-CPA re-encrypted with a new counter before being written to $S_{\neg b}$ (steps 17, 20). $\mathbf{I'}$ is assigned by the client to a new random index selected from a set of dummy addresses in $S_{\neg b}$ as $u^{(\neg b)} \overset{\$}{\leftarrow} T_w.\mathcal{L}_{\neg b}$, which is independent from $u^{(b)}$. Under 1, S_b does not have a view on $S_{\neg b}$ and vice versa. So, S_b does not know if $\mathbf{I'}$ is assigned to which $u^{(\neg b)}$ in $S_{\neg b}$ and under which new encryption form $\hat{\mathbf{I}}$. Therefore, $(u^{(b)}, \mathbf{I}_u^{(b)})$ can represent the same item with any address-data tuples $(u^{(\neg b)}, \mathbf{I}_u^{(\neg b)})$ in $S_{\neg b}$ with the same probability of $\frac{1}{N}$, where N is the number of items in $S_{\neg b}$. By Definition 14, $(u^{(b)}, \mathbf{I}_u^{(b)})$ is unlinkable to any items in $S_{\neg b}$ from S_b's view. Considering the $S_{\neg b}$'s view, $S_{\neg b}$ also does not know which address-data tuple $(u^{(b)}, \mathbf{I}_u^{(b)})$ was read from S_b under 1. Meanwhile, $\hat{\mathbf{I}}$ is a IND-CPA encryption so that it looks random-uniform to all other data in $S_{\neg b}$. Moreover, the address associating with $\hat{\mathbf{I}}$ is selected randomly from the set of dummy addresses $\mathcal{L}_{\neg b}$ with $|\mathcal{L}_{\neg b}|= N$. It is oblivious for $S_{\neg b}$ to link $\hat{\mathbf{I}}$ to any item which will be queried subsequently. Notice that to achieve this obliviousness, it is mandatory to always keep $|\mathcal{L}_{\neg b}| = N$. Once the new IND-CPA encryption form $\hat{\mathbf{I}}$ of an item is written to new address $u^{(\neg b)}$ in $S_{\neg b}$, its old address in $S_{\neg b}$ will be set to dummy and included to $\mathcal{L}_{\neg b}$ by the client (Figure 4.47, steps 2, 5).

We next consider search address-data tuple denoted as $(ut^{(b)}, \hat{\mathbf{I}}_{ut}^{(b)})$ which is written to S_b under S_b's view. This tuple is the new representation of an arbitrary item which has just been queried from $S_{\neg b}$. As DOD-DSSE access operations on servers S_b and $S_{\neg b}$ are symmetric, meaning that S_b can act as $S_{\neg b}$ in the aforementioned analysis and vice versa. Therefore, the same analysis is applied to this case.

Finally, we show that if each pair $(\mathsf{access}(x_i^{(b)}), \mathsf{access}(x_i^{(\neg b)}))$ is pairwise unlinkable to each other, for all $1 \leq i \leq q$, then $\mathsf{access}(x_i^{(b)})$ is also unlinkable to others $\mathsf{access}(x_j^{(\neg b)})$ for all $1 \leq j \neq i \leq q$. Without loss of generality, we assume that $j < i$. As $(\mathsf{access}(x_j^{(b)}), \mathsf{access}(x_j^{(\neg b)}))$ is pairwise unlinkable, meaning that given $\mathsf{access}(x_j^{(b)})$ observed by S_b, the corresponding $\mathsf{access}(x_j^{(\neg b)})$ generated in $S_{\neg b}$ is oblivious from S_b's view. It is computationally infeasible for S_b to link $\mathsf{access}(x_i^{(b)})$ with any access patterns $\mathsf{access}(x_j^{(\neg b)})$ generated in $S_{\neg b}$ by just observing $\mathsf{access}(x_j^{(b)})$. The same principle applies to $S_{\neg b}$ as operations on two servers are symmetric. Hence, the Lemma 9 holds. □

Corollary 7. For any access pattern observed by S_b (or $S_{\neg b}$), the same query will result in the same address being accessed on S_b (or $S_{\neg b}$), at most twice.

Proof. Assume that at step i the real query x is performed and its corresponding data item is read from address $u_x^{(b)}$ in server S_b. According to DOD-DSSE scheme, data of x will be written to an arbitrary address $u_x^{(\neg b)}$ in $S_{\neg b}$ (Figure 4.42, step 21). Given that the same query x is performed again at step $j > i$, its data will be read from $S_{\neg b}$ so that $S_{\neg b}$ can observe the address $u_x^{(\neg b)}$ accessed at step i is now accessed again. By this access pattern, $S_{\neg b}$ can infer the same query is performed at step i and j on it. However from S_b's view, it is oblivious for S_b to determine if the data being written to an arbitrary address $ut_x^{((b))}$ at step j is associated with the query x at step i due to Lemma 9. Now assume that at step k, where $k > j > i$, the query x is performed again. It will be read from S_b and then written to $S_{\neg b}$. Similar to that of S_b at step j, $S_{\neg b}$ observes an access operation which is unlinkable to access operations generated at step j and i by Lemma 9. It can be easily seen that the same query only generates the same address access at most two times. Therefore, the corollary holds. □

Theorem 6. Given a server S_b, DOD-DSSE *achieves security by Definition 13, meaning that* DOD-DSSE *leaks no information beyond ORAM security definition with the exception of one-time repetition of an unlinkable query on the other $S_{\neg b}$, to which S_b does not have access.*

Proof. Given an access pattern $\mathbf{AP}_b$ of length q as in (Equation 4.16) observed by server S_b, denote $\mathcal{Y}_q$ as the set of all possible combinations y of N data items, where $|y|= q$. We have N^q possible strings as $|\mathcal{Y}_q|= N^q$. Let $u_{i,1}^{(b)}, u_{i,2}^{(b)}$ be read and write addresses for search query observed by S_b, respectively, given a data access request $\mathsf{access}(x_i^{(b)})$. From S_b's view, data item of the real query x_i (denoted as $\mathbf{I}'_{x_i}$) can be accessed from $u_{i,1}^{(b)}$, or $u_{i,2}^{(b)}$ (*i.e.*, $\mathbf{I}'_{x_i}$ is actually accessed from $S_{\neg b}$ and then written to S_b). By Lemma 9, we have:

$$\Pr(\mathsf{access}(x_i^{(b)})) = \sum_{j=1}^{2}\frac{1}{2}\Big[\Pr(\mathsf{pos}(\mathbf{I}'_{x_i}) = u_{i,j}^{(b)})\Pr(u_{i,j}^{(b)} \in T_w.\mathcal{L}_b)$$
$$+\Pr(\mathsf{pos}(\mathbf{I}'_{x_i}) = u_{i,j}^{(b)})\Pr(u_{i,j}^{(b)} \notin T_w.\mathcal{L}_b)\Big] = \sum_{j=1}^{2}\frac{1}{2}\left[\frac{1}{N}\frac{1}{2}+\frac{1}{N}\frac{1}{2}\right] = \frac{1}{N}.$$

Notice that given a real query x_i, Figure 4.42 generates two random row indices on each server S_b: one is from the set of dummy addresses and the other is from from the set of non-empty addresses. Such addresses are removed from their current set and included in the other set. This is to maintain the size of each set so that given another real query $x_j \neq x_i$, its generated addresses are randomly chosen from size-consistent sets, making it independent of each other. Hence, from S_b's observation, access patterns generated by $\mathsf{access}(x_i^{(b)})$ are computational indistinguishable from those generated by $\mathsf{access}(x_j^{(b)})$, given that $x_j \neq x_i$.

For $x_j = x_i$, with $q \geq j > i \geq 1$, we have two cases:

1. If $\mathsf{pos}(\mathbf{I}'_{x_i}) = u^{(b)}_{i,1}$ then $\mathsf{pos}(\mathbf{I}'_{x_j}) = u^{(b)}_{j,2}$, meaning $\mathbf{I}'_{x_i}$ is read from S_b, while $\mathbf{I}'_{x_j}$ is read from $S_{\neg b}$. By Lemma 9, data in $u_{j,2}$ is unlinkable to any data items in S_b. Therefore, $\mathsf{access}(x^{(b)}_j)$ generates an access pattern which is statistically independent from $\mathsf{access}(x^{(b)}_i)$ in server S_b.

2. If $\mathsf{pos}(\mathbf{I}'_{x_i}) = u^{(b)}_{i,2}$ then $\mathsf{pos}(\mathbf{I}'_{x_j}) = u^{(b)}_{j,1}$, meaning $\mathbf{I}_{x_i}$ is read from $S_{\neg b}$ and $\mathbf{I}'_{x_j}$ is read from S_b. S_b observes that the same address is accessed again (*i.e.*, $u^{(b)}_{j,1} = u^{(b)}_{i,2}$). By Lemma 9, data from $u^{(b)}_{i,2}$ is written to $S_{\neg b}$ which is unlinkable to any data items in $S_{\neg b}$; the probability that S_b can determine if $\mathbf{I}'_{x_j}$ is re-written back to it in subsequent access operations is $\frac{1}{N}$. That means given another query x_k such that $x_k = x_j = x_i$, with $k > j > i$, the access pattern generated by $\mathsf{access}(x^{(b)}_k)$ is statistically independent from $\mathsf{access}(x^{(b)}_j)$ as in case (i). Therefore, the information DOD-DSSE leaks in this case is that the same query can generate the same address access at most twice, as shown in Corollary 7.

To sum up, we have $\Pr(\mathbf{AP}_b) = \prod_{j=1}^{q} \Pr(\mathsf{access}(x^{(b)}_j)) = \left(\frac{1}{N}\right)^{(q-r)}$, where r is the number of one-time repetitions of data access requests as in case (ii). There are $N^{(t-r)}$ possible strings $y \in \mathcal{Y}_q$ that can generate the same $\mathbf{AP}_b$ so that it is computationally indistinguishable for S_b to determine which string in $N^{(t-r)}$ candidates generates the $\mathbf{AP}_b$. In the worst case, where there are $r = q/2$ repetitions in the data request sequence of length q, then $\Pr(\mathbf{AP}_b) = \left(\frac{1}{N}\right)^{q/2}$. □

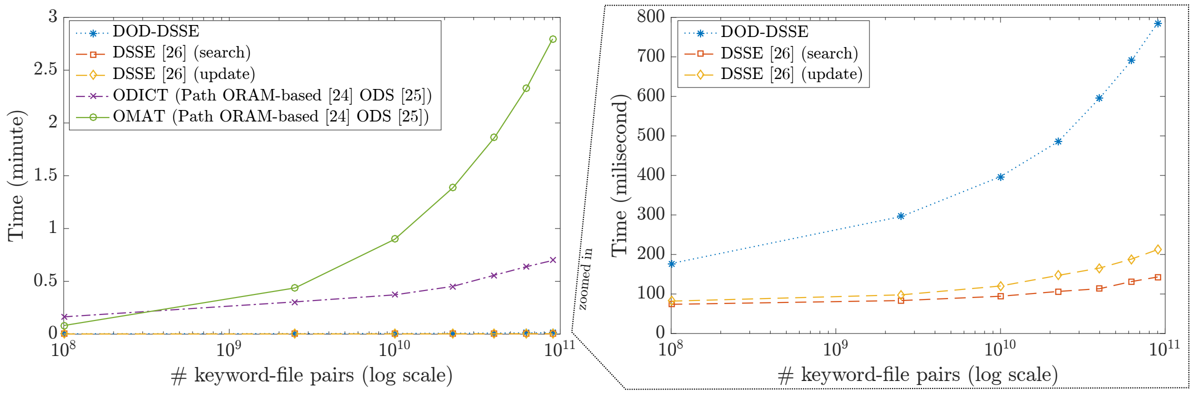
In traditional DSSE, each search or update query on a keyword or a file produces the same address being accessed for consistency purposes. This deterministic relation between queries and address tokens permits an adversary to perform statistical attacks, such as query frequency analysis, to uncover the relations among keyword/file being accessed [34, 103, 124, 190].

In DOD-DSSE, queries observed in each server are unlinkable by Definition 12 to each other, meaning that they can be independently generated by any possible keywords/files, from the server's view. DOD-DSSE achieves the security by Definition 13 in that, the only information that server S_b can infer from its observed access pattern is one-time repetition of an arbitrary query, which is previously performed on the other server $S_{\neg b}$. Note that servers do not have a view of each other's accesses or queries (*i.e.*, non-colluding servers). This leakage is negligible for any practical setting, and does not permit to establish any statistical relationship, since one-time repetitions are unlinkable. These security guarantees imply that DOD-DSSE can not only prevent statistical analysis (*e.g.*, [124]) but also any other potential threats that may exploit the linkability among arbitrary queries.

4.6.1.4 Experimental Evaluation

We evaluated the performance of our scheme on real network settings with different network latencies. By latency, we mean the round-trip time taken by a packet to go from the host (*i.e.*, client) to the destination (*i.e.*, server) and back. In addition, we made several comparisons. First, we compared our scheme's cryptographic end-to-end delay (*i.e.*, the time to completely process a search or update operation) with a traditional DSSE scheme which does not hide the access pattern (*e.g.*, [189]). We then compared to a *simulated* scheme which applies ORAM on a DSSE dictionary data structure and matrix data structure[15]. We notice that in order to be comparable with ORAM, which can achieve oblivious operations (*i.e.*, whether the operation is search or update), our DOD-DSSE scheme is designed to always perform *both* search and update queries regardless of the type of actual operation which is required. Therefore, search and update operations require the same amount of time. This is in contrast with traditional DSSE schemes in which search and update operations incur different delays. This will be shown in the following experiments.

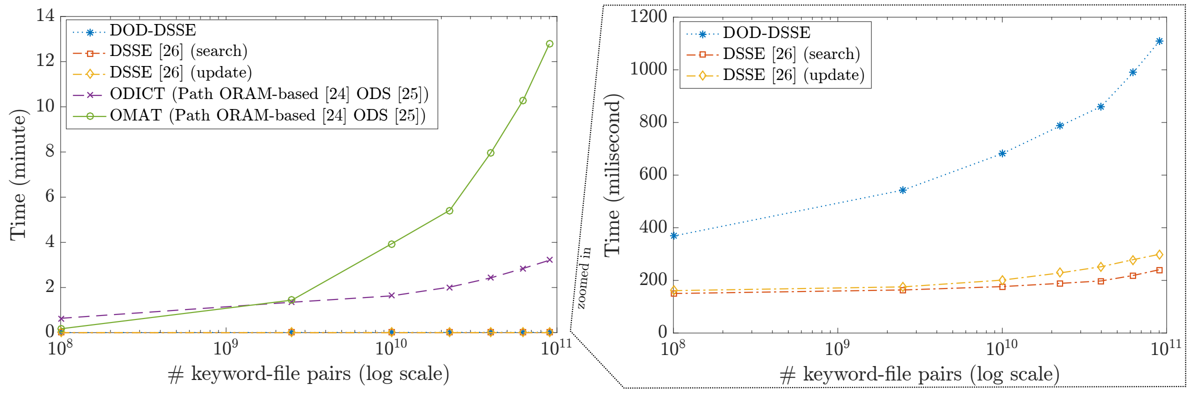
[15] We did not implement the full scheme, we estimated the performance by simulation in a real network setting and assuming a 4 KB block-size ORAM as presented in [169, 182].



(a) *DOD − DSSE* vs. ODS and traditional DSSE

(b) *DOD − DSSE* vs. traditional DSSE

Figure 4.48: End-to-end cryptographic delay with in-state network latency.



(a) *DOD − DSSE* vs. ODS and traditional DSSE

(b) *DOD − DSSE* vs. traditional DSSE

Figure 4.49: End-to-end cryptographic delay with out-state network setting.

We used a HP Z230 Desktop as the client and two virtual servers provided by Amazon EC2. The client machine was installed with CentOS 7.2 and equipped with Intel Xeon CPU E3-1231v3 @ 3.40GHz, 16 GB RAM and 256 GB SSD. We deployed servers running Ubuntu 14.04 with `m4.4xlarge` instance type which offers 16 vCPUs @ 2.4 GHz, Intel Xeon E5-2676v3, 64 GB RAM and 200 GB SSD for each server.

We adopted Google sparse hash table[16] to implement the data structures T_f, T_w stored at the client side. We implemented IND-CPA encryption and decryption schemes using AES-CTR mode as it supports parallelism and key pre-computation. We used AES-128 CMAC to implement

[16] https://github.com/sparsehash/sparsehash

the hash function H. For cryptographic primitives, we utilized libtomcrypt[17] with Intel AES-NI hardware accelerated library[18] to optimize the performance of cryptographic operations. We used ZeroMQ library[19] to implement network communication between client and server(s).

We performed our experiments on the Enron email dataset[20]. We selected subsets of the Enron corpus to construct the DSSE data structure with various combinations of keyword-file pairs, ranging from 10^8 to 9×10^{10}. This is to evaluate the performance of DOD-DSSE and its counterparts with different dataset sizes starting from small to very large similar to [35].

We first measured the pre-processing time to build the encrypted data structures in DOD-DSSE with different sizes. With the largest data structure being experimented, which consists of 9×10^{11} keyword-file pairs (*i.e.*, 300,000 files and 300,000 keywords), it takes the client roughly 20 hours to construct two encrypted incidence matrices. For 10^8 keyword-file pairs, the time is 30 seconds. Notice that this initialization phase is only run one time in the offline phase so that its cost is not an important factor. Our focus is to evaluate the performance of DOD-DSSE and its counterparts in the online phase, where we perform search and update operations on the constructed data structure(s).

Next, we showed the performance of DOD-DSSE scheme in the online phase. We created two Amazon EC2 servers in the same geographical region (*i.e.*, in-state), resulting in an average network latency of 11 ms and throughput of 100 Mbps. It takes approximately 800 ms to perform a search (or update) operation on the distributed data structure consisting of 9×10^{11} keyword-file pairs, as demonstrated in Figure 4.48.

[17] http://www.libtom.org/LibTomCrypt/
[18] https://software.intel.com/articles/download-the-intel-aesni-sample-library
[19] http://zeromq.org/
[20] https://www.cs.cmu.edu/∼./enron/

Table 4.6: Total size of encrypted data structure(s) in GB.

# keyword-file pairs	DOD-DSSE[†]	DSSE [189]	ODICT	OMAT
10^8	0.1	0.02	1.37	0.05
2.5×10^9	2.4	0.6	37.38	1.16
10^{10}	10	2.4	149.54	4.66
2.25×10^{10}	22.4	5.6	336.46	10.48
4×10^{10}	40	10	598.15	18.63
6.25×10^{10}	62.4	15.6	934.60	29.10
9×10^{10}	90	22.4	1345.83	41.91

[†] DOD-DSSE stores two encrypted data structures in two non-colluding servers so that the storage cost for each server will be a half of presented numbers.

We compared the actual cost of DOD-DSSE with that of our counterparts. We selected the scheme in [189] to be our main traditional DSSE counterpart as, to the best of our knowledge, it is the most secure DSSE scheme in the literature. We used Path ORAM [169] protocol for ODS as it offers optimal bandwidth overhead. We simulated the cost of using ODS on a dictionary (denoted as ODICT) and a square incidence matrix (denoted as OMAT) data structures because the former provides sublinear operating time while the latter achieves the best security. We applied an average padding strategy to mitigate the information leakage from ODICT due to the optimal search/update time property.

We analyzed the asymptotic communication complexity of the aforementioned schemes. Traditional DSSE achieves optimal bandwidth overhead of $O(r)$, where r is the number of data corresponding with the search/update query [189]. In ODS approaches, keyword-files pairs are packaged into 4KB blocks, and the total number of blocks is denoted as B. Given a search/update operation, the number of blocks being transmitted by OMAT and ODICT is $s \cdot c \cdot O(\log B)$ and $s' \cdot c \cdot O(\log B)$, respectively, where $c = 4$ is the bucket size in Path ORAM [169] and s, s' are the numbers of communication rounds to retrieve sufficient results for the query [182]. In DOD-DSSE, the bandwidth complexity is $4 \cdot O(N)$, where N is the maximum number of unique keywords/files that DOD-DSSE can support.

After that, we benchmarked the actual performance of DOD-DSSE and its counterparts in practice based on the previous asymptotic communication analysis. Figure 4.48 demonstrates the actual end-to-end cryptographic delay (*i.e.*, encryption, transmission delays) of schemes using in-state Amazon EC2 server(s) with various data structure sizes. We can see that DOD-DSSE incurs a small-constant communication overhead due to extra queries (*i.e.*, 4x times slower than traditional DSSE). However, it is *approximately 50x and 210x times faster than ODICT and OMAT* which specifically take 42 and 167 seconds to perform an operation on the large data structure, respectively. This indicated that even though the asymptotic complexity of ODS approaches looks very efficient, hidden constants such as c, s, s' actually contribute a lot to the communication overhead, as shown in Figure 4.48.

We inspected the cost of DOD-DSSE to investigate the impacts of network communication and cryptographic operations on the end-to-end delay. We observed that the majority of the delay is due to network transmission, in which the ratio between it and cryptographic operations is roughly 8:1. To investigate more the impact of network latency and throughput on DOD-DSSE and its counterparts, we setup two EC2 servers geographically located outside of our state, resulting in a network latency and throughput of 31 ms and 30 Mbps, respectively. As it can be seen in Figure 4.49(b), this geographically distributed out-state environment makes DOD-DSSE and traditional DSSE perform approximately 200 ms and 100 ms slower than in-state setting, respectively. Due to the characteristics of ODS requiring a number of communication rounds to perform a search or update operation, slower network latency and throughput significantly impact the performance of ODICT and OMAT. We can see that DOD-DSSE is now 170x and 690x times faster than ODICT and OMAT, respectively by this setting. Comparing with the in-state configuration, ODICT and

OMAT are both 4.5x times slower than their in-state version. This implies 12.78 minutes and 3.2 minutes to accomplish an operation, respectively (Figure 4.49(a)).

Finally, we analyzed the storage cost of DOD-DSSE. With the largest dataset being experimented in this study (*i.e.*, 9×10^{11} keyword-file pairs), DOD-DSSE requires approximately 35 MB to store at the client side all necessary information for its operation such as symmetric keys, T_f, T_w and global counter arrays. This can be easily fulfilled even by resource-limited devices such as a smartphone or a tablet. Table 4.6 shows the total size of the encrypted data structure(s) stored at the server side required by DOD-DSSE and its counterparts with different dataset sizes. DOD-DSSE requires 8x and 2x times as much storage space as that of traditional DSSE and OMAT, respectively, and yet, is much more compact than ODICT using dictionary[21]. Considering the advantages of DOD-DSSE in terms of efficiency, storage cost and achieved security aspects over traditional DSSE and ORAM-based methods, our scheme is likely to be an ideal security-performance trade-off DSSE scheme for privacy-critical cloud computing.

4.6.2 ODSE: Oblivious Dynamic Searchable Encryption

In DSSE, it is highly desirable to seal access pattern leakages when accessing the encrypted index ($\mathbf{I}$) and encrypted files ($\mathbf{f}$). 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 $\mathbf{f}$. In this work, we focus more on oblivious access techniques on the index ($\mathbf{I}$) that are more bandwidth-efficient than using generic ORAM (Figure 4.50). Specifically, we propose ODSE, a comprehensive oblivious encrypted index framework in the multi-server setting with the application to DSSE. The framework contains three

[21] We would like to notice an advantage of dictionary over matrix structure. That is, it is not limited by the number of unique keywords and files, but only the maximum number of keyword-file pairs which might be useful for applications requiring diverse keyword-file relations.

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 §4.6.2.3).

- *Low end-to-end delay:* All ODSE schemes offer low end-to-end-delay, which are 3×-57× faster than using generic ORAM atop the DSSE encrypted index (with optimization [67]) under real network settings.

- *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

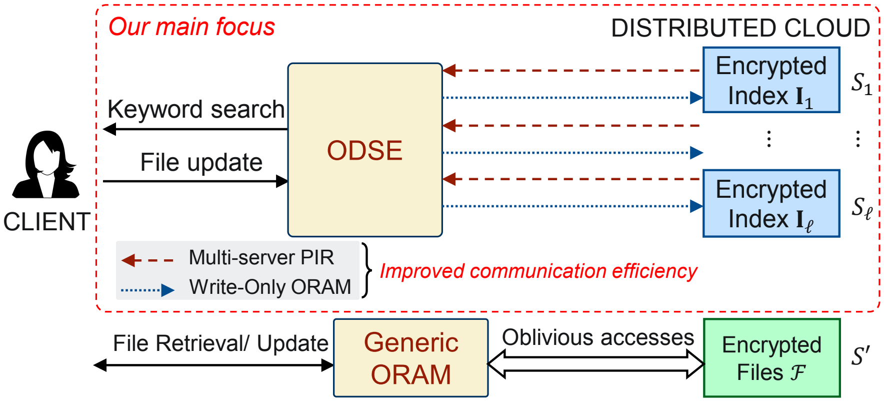


Figure 4.50: ODSE research objective and high-level approach.

knowledge, we are among the first to open-source an oblivious access framework for the encrypted index in DSSE. The code is available at https://github.com/thanghoang/ODSE.

4.6.2.1 System and Threat Models

Our system model comprises a client and ℓ servers $\boldsymbol{S} = (S_1, \ldots, S_\ell)$, each storing a version of the encrypted index. The encrypted files are stored on a separate server different from $\boldsymbol{S}$ (as in [92]), which can be obliviously accessed via a generic ORAM scheme [150, 169]. We focus only on oblivious access on distributed encrypted index $\mathcal{I}$ on $\boldsymbol{S}$. We present the definition of ODSE as follows.

Definition 15. An Oblivious Distributed Dynamic Searchable Symmetric Encryption (ODSE) scheme is a tuple of one algorithm and two protocols $\mathsf{ODSE} = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$, where the input and the output for the client and the servers are separated with semicolon such that:

1. $(\sigma, \mathcal{I}) \leftarrow \mathsf{Setup}(\mathbf{f})$: Given a set of files $\mathbf{f}$ 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.

In our system, the client is trusted and the set of servers $\boldsymbol{S}$ 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 (§4.6.2.4). We allow up to $t < \ell$ (privacy parameter) servers among $\boldsymbol{S}$ 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 16 (ODSE security w.r.t. semi-honest adversary). Let $\overrightarrow{o} = (\mathsf{op}_1, \ldots, \mathsf{op}_q)$ 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 $\mathbf{ODSE}_j(\overrightarrow{o})$ represent the ODSE client's sequence of interactions with server S_j, given an operation sequence $\overrightarrow{o}$.

An ODSE is *correct* if for any operation sequence $\overrightarrow{o}$, $\{\mathbf{ODSE}_1, \ldots, \mathbf{ODSE}_\ell\}$ returns data consistent with $\overrightarrow{o}$, except with negligible probability.

An ODSE is *t-secure* if $\forall \mathcal{L} \subseteq \{1, \ldots, \ell\}$ such that $|\mathcal{L}| \leq t$, for any two operation sequences $\overrightarrow{o}_1$ and $\overrightarrow{o}_2$ where $|\overrightarrow{o}_1| = |\overrightarrow{o}_2|$, the views $\{\mathbf{ODSE}_{l\in\mathcal{L}}(\overrightarrow{o}_1)\}$ and $\{\mathbf{ODSE}_{l\in\mathcal{L}}(\overrightarrow{o}_2)\}$ observed by a coalition of up to t servers are computationally indistinguishable.

By Definition 15, 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 16. 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.

4.6.2.2 The Proposed Semi-Honest ODSE Schemes

In DSSE, keyword search and file update on **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 **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 [82]. 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 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

Table 4.7: ODSE symbols and notation.

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
$\mathbf{S}$	Stash to (temporarily) store column data
$\mathbf{c}$	Column counter vector

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 §4.6.2.3 and §4.6.2.4, respectively.

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 $\mathbf{S}$ to temporarily store columns that might not be written back during the update due to the overflow.

```
$(\sigma,\mathcal{I}) \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Setup}(\mathbf{f})$:
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 $\Pi$ and $\Pi'$ be a random permutation on $\{1,\ldots,2n\}$ and $\{1,\ldots,m\}$ respectively
3: $k_3 \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)$
4: Extract keywords $(w_1,\ldots,w_m)$ from files $\mathbf{f} = \{f_{id_1},\ldots,f_{id_n}\}$
5: for $i = 1,\ldots,m$ do
6:     $T_w[w_i] \leftarrow \Pi'(i)$
7:     for $j = 1,\ldots,n$ do
8:         $T_f[id_j] \leftarrow \Pi(j)$
9:         if $w_i$ appears in $f_{id_j}$ then
10:            $\mathbf{I}'[x_i,y_j] \leftarrow 1$, where $x_i \leftarrow T_w[w_i]$, $y_j \leftarrow T_f[id_j]$
11: for  $i = 1,\ldots,m$ do
12:     $\tau_i \leftarrow \mathsf{KDF}(k_3||i)$                              # Compute key for each row
13:     for $j = 1,\ldots,2n$ do
14:         $\mathbf{I}[i,j] \leftarrow \mathcal{E}.\mathsf{Enc}_{\tau_i}(\mathbf{I}'[i,j], j||c_j)$        # Ciphertext $\mathbf{I}[i,j]$ is one-bit long
15: Let $\mathcal{I}$ contain $\ell$ copies of $\mathbf{I}$ and $\sigma \leftarrow (k_3, T_w, T_f, \mathbf{c})$
16: return $(\sigma,\mathcal{I})$
```

Figure 4.51: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ setup algorithm.

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.

Figure 4.51 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), 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

```
$(\mathcal{R}; \bot) \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Search}(w, \sigma; \mathcal{I})$:
Client:
 1: $i \leftarrow T_w[w]$
 2: $(\rho_1, \ldots, \rho_\ell) \leftarrow \mathsf{PIR}^{\mathsf{xor}}.\mathsf{CreateQuery}(i)$
 3: Send $\rho_l$ to $S_l$ for $l \in \{1, \ldots, \ell\}$
Server: each $S_l \in \{S_1, \ldots, S_\ell\}$ receiving $\rho_l$ 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 $\ell$ servers
 6: $\mathbf{I}[i, *] \leftarrow \mathsf{PIR}^{\mathsf{xor}}.\mathsf{Reconstruct}(\hat{I}_1, \ldots, \hat{I}_\ell)$
 7: $\tau_i \leftarrow \mathsf{KDF}(k_3 || i)$
 8: for $j = 1, \ldots, 2n$ do
 9:     $\mathbf{I}'[i, j] \leftarrow \mathcal{E}.\mathsf{Dec}_{\tau_i}(\mathbf{I}[i, j], j || c_j)$
10: Let $\mathcal{J} := \{j : (\mathbf{I}'[i, j] = 1)$ and $((j$ is not dummy) or $(\mathbf{I}'[i, j] \in \mathbf{S}))\}$
11: return $(\mathcal{R}; \bot)$, where $\mathcal{R}$ contains file IDs at column indexes in $\mathcal{J}$
```

Figure 4.52: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ search protocol.

stream key. To this end, the client sends a replica of $\mathbf{I}$ to ℓ servers and keeps some information (*i.e.*, $k_3, T_w, T_f, \mathbf{c}$) private.

$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ harnesses XOR-based PIR on the row dimension of $\mathbf{I}$ to conduct the oblivious keyword search as shown in Figure 4.52. 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).

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 4.53. 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

```
$(\sigma';\mathcal{I}') \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}.\mathsf{Update}(f_{id},\sigma;\mathcal{I})$:
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:     $\hat{I}[x_i] \leftarrow 1$, where $x_i \leftarrow T_w[w_i]$
 4: $\mathbf{S} \leftarrow \mathbf{S} \cup \{(id,\hat{I})\}$ and $T_f[id] \leftarrow 0$            # Put updated column to stash
 5: Let $\mathcal{J}$ contain $\lambda$ random-selected column indexes, send $\mathcal{J}$ to an arbitrary server $S_l$
Server $S_l$: 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:     $\tau_i \leftarrow \mathsf{KDF}(k_3||i)$
 9:     for each index $j \in \mathcal{J}$ do                          # Decrypt columns
10:         $\mathbf{I}'[i,j] \leftarrow \mathcal{E}.\mathsf{Dec}_{\tau_i}(\mathbf{I}_l[i,j], j||c_j)$
11: for each dummy index $\hat{j} \in \mathcal{J}$ do                   # Put columns from stash
12:     $\mathbf{I}'[*,\hat{j}] \leftarrow \hat{I}$ and $T_f[id] \leftarrow \hat{j}$, where $(id,\hat{I})$ is picked from $\mathbf{S}$
13: for each index $j \in \mathcal{J}$ do                              # Re-encrypt columns
14:     $c_j \leftarrow c_j + 1$
15:     for $i = 1,\ldots,m$ do
16:         $\hat{\mathbf{I}}[i,j] \leftarrow \mathcal{E}.\mathsf{Enc}_{\tau_i}(\mathbf{I}'[i,j], j||c_j)$
17: Send $\{\hat{\mathbf{I}}[*,j]\}_{j\in\mathcal{J}}$ to $\ell$ servers
Server: each $S_l \in \{S_1,\ldots,S_\ell\}$ receiving $\{\hat{\mathbf{I}}[*,j]\}_{j\in\mathcal{J}}$ do
18: for each $j \in \mathcal{J}$ do
19:     $\mathbf{I}_l[*,j] \leftarrow \hat{\mathbf{I}}[*,j]$
20: return $(\sigma';\mathcal{I}')$ where $\mathcal{I}'$ is $\mathbf{I}_l$ updated at $\ell$ servers, and $\sigma'$ is updated client state
```

Figure 4.53: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ update protocol.

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

Figure 4.54: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ setup algorithm.

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).

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

```
$(\mathcal{R};\perp) \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}.\mathsf{Search}(w,\sigma;\mathcal{I})$:
Client:
1: $i \leftarrow T_w[w]$
2: $([\![\mathbf{e}]\!]_1,\dots,[\![\mathbf{e}]\!]_\ell) \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{CreateQuery}(i)$
3: Send $[\![\mathbf{e}]\!]_l$, to $S_l$ for $l \in \{1,\dots,\ell\}$
Server: each $S_l \in \{S_1,\dots,S_\ell\}$ receiving $[\![\mathbf{e}]\!]_l$ do:
4: for $j = 1\dots,2n'$ do
5:     for $k = 1,\dots,m$ do
6:         $\langle c_{jk}^{(l)}\rangle_{\mathsf{bin}} \leftarrow \mathbf{I}_l[k,(j-1)\cdot\lfloor\log_2 p\rfloor + 1 \dots j\cdot\lfloor\log_2 p\rfloor]$    # $j^{th}$ batch of $k^{th}$ row
7:     $\mathbf{c}_j^{(l)} \leftarrow (c_{j1}^{(l)},\dots,c_{jm}^{(l)})$
8:     $[\![b_j]\!]_l \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Retrieve}([\![\mathbf{e}]\!]_l,\mathbf{c}_j^{(l)})$
9: Send $([\![b_1]\!]_l,\dots,[\![b_{2n'}]\!]_l)$ to the client
Client: On receive $\{\mathcal{B}_j = \{[\![b_j]\!]_1,\dots,[\![b_j]\!]_\ell\}\}_{j=1}^{2n'}$ from $\ell$ servers
10: for $j = 1\dots,2n'$ do
11:     $b_j \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\mathcal{B}_j,t)$
12: $\mathbf{I}[i,*] \leftarrow \langle b_1\rangle_{\mathsf{bin}}||\dots||\langle b_{2n'}\rangle_{\mathsf{bin}}$
13: $\tau_i \leftarrow \mathsf{KDF}(k_3||i)$
14: for $j = 1,\dots,2n$ 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 $(\mathbf{I}'[i,j] \in \mathbf{S}))\}$
17: return $(\mathcal{R};\perp)$, where $\mathcal{R}$ contains file IDs at column indices in $\mathcal{J}$
```

Figure 4.55: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ search protocol.

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.

$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ 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 4.54.

$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ harnesses SSS-based PIR protocol on the row dimension of $\mathbf{I}$ to conduct keyword search as shown in Figure 4.55. 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

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

Figure 4.56: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ update protocol.

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).

$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ 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 4.56).

$\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ scheme relies on IND-CPA encryption for the encrypted index so that it only offers (at most) computational security. We now 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.

Figure 4.57 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

```
$(\sigma, \mathcal{I}) \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}.\mathsf{Setup}(\mathbf{f})$:
$(\mathbf{I}', T_w, T_f) \leftarrow$ Execute lines 2–10 in Figure 4.51
for $i = 1, \ldots, m$ do
    for $j = 1, \ldots, 2n'$ do
        $\langle b_{ij} \rangle_{\mathsf{bin}} \leftarrow \mathbf{I}[i, (j-1) \cdot \lfloor \log_2 p \rfloor + 1 \ldots j \cdot \lfloor \log_2 p \rfloor]$ # Pack each row into batches of size $\lfloor \log_2 p \rfloor$
        $(\mathbf{I}_1[i,j], \ldots, \mathbf{I}_\ell[i,j]) \leftarrow \mathsf{SSS.Create}(\langle b_{ij} \rangle_{\mathsf{bin}}, t)$
return $(\sigma, \mathcal{I})$, where $\mathcal{I} \leftarrow \{\mathbf{I}_1, \ldots, \mathbf{I}_\ell\}$ and $\sigma \leftarrow (T_w, T_f)$
```

Figure 4.57: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ setup algorithm.

```
$(\mathcal{R}; \perp) \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}.\mathsf{Search}(w, \sigma; \mathcal{I})$:
Client:
 1: $i \leftarrow T_w[w]$
 2: $([\![\mathbf{e}]\!]_1, \ldots, [\![\mathbf{e}]\!]_\ell) \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{CreateQuery}(i)$
 3: Send $[\![\mathbf{e}]\!]_l$, to $S_l$ for $l \in \{1, \ldots, \ell\}$
Server: each $S_l \in \{S_1, \ldots, S_\ell\}$ receiving $[\![\mathbf{e}]\!]_l$ do
 4: for $j = 1 \ldots, 2n'$ do
 5:     $[\![b_j]\!]_l \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Retrieve}([\![\mathbf{e}]\!]_l, \mathbf{I}_l[*, j])$
 6: Send $([\![b_1]\!]_l, \ldots, [\![b_{2N'}]\!]_l)$ to the client
Client: On receive $\{\mathcal{B}_j = \{[\![b_j]\!]_1, \ldots, [\![b_j]\!]_\ell\}\}_{j=1}^{2n'}$ from $\ell$ servers
 7: for $j = 1 \ldots, 2n'$ do
 8:     $b_j \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\mathcal{B}_j, 2t)$
 9: $\mathbf{I}'[i, *] \leftarrow \langle b_1 \rangle_{\mathsf{bin}} || \ldots || \langle b_{2n'} \rangle_{\mathsf{bin}}$
10: Let $\mathcal{J} := \{j : (\mathbf{I}'[i,j] = 1)$ and $((j$ is not dummy) or $(\mathbf{I}'[i,j] \in \mathbf{S}))\}$
11: return $(\mathcal{R}; \perp)$, where $\mathcal{R}$ contains file IDs at column indices in $\mathcal{J}$
```

Figure 4.58: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ search protocol.

$n' = n / \lfloor \log_2 p \rfloor$. To this end, the client sends $\mathbf{I}_l$ to server S_l and keep position maps (*i.e.*, T_w, T_f) private.

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 4.58. 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 S_l is a share matrix, each dot product computation in the SSS-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).

```
$(\sigma';\mathcal{I}') \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}.\mathsf{Update}(f_{id},\sigma;\mathcal{I})$:
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:     $\hat{I}[x_i] \leftarrow 1$, where $x_i \leftarrow T_w[w_i]$
 4: $\mathbf{S} \leftarrow \mathbf{S} \cup \{(id,\hat{I})\}$ and $T_f[id] \leftarrow 0$
 5: Let $\mathcal{J}$ contain $\lambda$ random-selected column indexes, send $\mathcal{J}$ to $(t+1)$ arbitrary servers $S_{l_1},\ldots,S_{l_{t+1}}$
Server: each $S_l \in \{S_{l_1},\ldots,S_{l_{t+1}}\}$ 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:     for each index $j \in \mathcal{J}$ do
 9:         $b_{ij} \leftarrow \mathsf{SSS.Recover}(\mathcal{B}_{ij},t)$
10:         $\mathbf{I}'[i, j\cdot\lfloor\log_2 p\rfloor+1,...(j+1)\cdot\lfloor\log_2 p\rfloor] \leftarrow \langle b_{ij}\rangle_{\mathsf{bin}}$
11: for each dummy column $\mathbf{I}'[*,\hat{j}]$ do
12:     $\mathbf{I}'[*,\hat{j}] \leftarrow \hat{I}$ and $T_f[id] \leftarrow \hat{j}$, where $(id,\hat{I})$ is picked from $\mathbf{S}$
13: for each index $j \in \mathcal{J}$ do
14:     for $i = 1\ldots,m$ do
15:         $\langle b'_{ij}\rangle_{\mathsf{bin}} \leftarrow \mathbf{I}'[i, j\cdot\lfloor\log_2 p\rfloor+1,\ldots,(j+1)\cdot\lfloor\log_2 p\rfloor]$
16:         $(\hat{\mathbf{I}}_1[i,j],\ldots,\hat{\mathbf{I}}_\ell[i,j]) \leftarrow \mathsf{SSS.Create}(b'_{ij},t)$
17: Send $\{\hat{\mathbf{I}}_l[*,j]\}_{j\in\mathcal{J}}$ to $S_l$ for $l \in \{1,\ldots,\ell\}$
Server: each $S_l \in \{S_1,\ldots,S_\ell\}$ receiving $\{\hat{\mathbf{I}}_l[*,j]\}_{j\in\mathcal{J}}$ do
18: for each $j \in \mathcal{J}$ do
19:     $\mathbf{I}_l[*,j] \leftarrow \hat{\mathbf{I}}_l[*,j]$
20: return $(\sigma';\mathcal{I}')$ where $\mathcal{I}'$ is $\mathbf{I}_l$ updated at each server $S_l$ and $\sigma'$ is updated client state
```

Figure 4.59: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ update protocol.

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 4.59. 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 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).

4.6.2.3 Security Analysis

Remark 5. 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 16, 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 7. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ *scheme is computationally* $(\ell - 1)$*-secure by Definition 16.*

Proof. (Sketch) *(i) Oblivious Search*: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ leverages XOR-based PIR and therefore, achieves $(\ell - 1)$-privacy for keyword search as proven in [45]. *(ii) Oblivious Update*: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ employs Write-Only ORAM which achieves negligible write failure probability and therefore, it offers the statistical security without counting the encryption. The index in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ is IND-CPA encrypted, which offers computational security. Therefore in general, the update access pattern of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ scheme is computationally indistinguishable. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ 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 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$. *(iii) ODSE Security:* By Remark 5, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ performs both search and update regardless of the actual operation. As analyzed, search is $(\ell - 1)$-private and update pattern is computationally secure. Therefore, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ achieves computational $(\ell - 1)$-security by Definition 16. □

Theorem 8. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ *scheme is computationally* t*-secure by Definition 16.*

Proof. (Sketch) *(i) Oblivious Search*: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ leverages a SSS-based PIR protocol and therefore, achieves t-privacy for keyword search due to the t-privacy property of SSS as proven in [17, 73]. *(ii) Oblivious Update*: Similar to $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ leverages Write-Only ORAM over IND-CPA encrypted database, which offers computational security as shown in [23]. *(iii) ODSE Security:* By Remark 5, 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 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ is a computationally indistinguishable in the presence of t colluding servers. □

Theorem 9. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ scheme is information-theoretically (statistically) t-secure by Definition 16.

Proof. (Sketch) *(i) Oblivious Search*: $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ leverages an SSS-based PIR protocol and therefore, achieves t-privacy for keyword search due to the t-privacy property of SSS [73]. *(ii) Oblivious Update*: The index in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is SSS-shared, which is information-theoretically secure in the presence of t colluding servers. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ also employs Write-Only ORAM, which offers statistical security due to negligible write failure probability. Therefore in general, the update access pattern of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ scheme is information-theoretically (statistically) indistinguishable in the coalition of up to t servers. *(iii) ODSE Security*: By Remark 5, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ 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, $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is information-theoretically (statistically) t-secure by Definition 16. □

4.6.2.4 ODSE with Malicious Security

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.

Let $\Sigma = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vrfy})$ be a secure keyed MAC scheme [110]: $\theta \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$ 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).

Let $\theta \leftarrow \mathbb{F}_p$ 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.

```
$(\sigma, \mathcal{I}) \leftarrow \textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}.\textsf{Setup}(\mathbf{f})$:
1: $(\mathbf{I}, T_f, T_w, \mathbf{c}, \kappa) \leftarrow$ Execute lines 1-14 in Figure 4.51
2: $\theta \leftarrow \Sigma.\textsf{Gen}(1^\lambda)$
3: for  $i = 1, \ldots, m$ do
4:     for $j = 1, \ldots, 2n/|\mu|$ do                # $|\mu|$:(pre-defined) length of the MAC tag.
5:         $\mathbf{T}[i, j] \leftarrow \Sigma.\textsf{Mac}_\theta(\mathbf{I}[i, (j-1) \cdot |\mu|+1 \ldots j \cdot |\mu|])$
6: Let $\mathcal{I}$ contain $\ell$ copies of $(\mathbf{I}, \mathbf{T})$ and $\sigma \leftarrow (\theta, k_3, T_w, T_f, \mathbf{c})$
7: return $(\sigma, \mathcal{I})$
```

Figure 4.60: $\textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}$ setup algorithm.

```
$(\mathcal{R}; \perp) \leftarrow \textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}.\textsf{Search}(w, \sigma; \mathcal{I})$:
Client:
 1: $(\rho_1, \ldots, \rho_\ell) \leftarrow$ Execute lines 1-2 in Figure 4.52  (to learn that the keyword $w$ corresponds to
    $i^{th}$ row and request retrieval of $i^{th}$ row privately)
 2: Send $\rho_l$ to $S_l$ for $l \in \{1, \ldots, \ell\}$
Server: each $S_l \in \{S_1, \ldots, S_\ell\}$ receiving $\rho_l$ do
 3: $\hat{I}_l \leftarrow$ Execute line 4 in Figure 4.52
 4: $\hat{T}_l \leftarrow \textsf{PIR}^{\textsf{xor}}.\textsf{Retrieve}(\rho_l, \mathbf{T}_l)$
 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 $\ell$ servers
 6: $\mathbf{I}[i, *] \leftarrow$ Execute line 6 in Figure 4.52
 7: $\mathbf{T}[i, *] \leftarrow \textsf{PIR}^{\textsf{xor}}.\textsf{Reconstruct}(\hat{T}_1, \ldots, \hat{T}_\ell)$
 8: for $j = 1 \ldots 2n/|\mu|$ do
 9:     if $\Sigma.\textsf{Vrfy}_\theta(\mathbf{I}[i, (j-1) \cdot |\mu|+1 \ldots j \cdot |\mu|], \mathbf{T}[i, j]) = 0$ then
10:         return abort
11: $\mathcal{J} \leftarrow$ Execute lines 7-10 in Figure 4.52
12: return  $(\mathcal{R}; \perp)$, where $\mathcal{R}$ contains file IDs at column indexes in $\mathcal{J}$
```

Figure 4.61: $\textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}$ search protocol.

We present $\textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}$, the extended version of $\textsf{ODSE}^{\textsf{wo}}_{\textsf{xor}}$, 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 $\textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}$ protocols are defined as follows.

Figure 4.60 presents the setup of $\textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}$ scheme with the MAC tag generation for the encrypted index. Generally speaking, it first generates the encrypted index **I** similar to semi-honest $\textsf{ODSE}^{\textsf{wo}}_{\textsf{xor}}$ (line 1), and then generates a MAC key (line 2), followed by computing a matrix **T** containing the MAC tag for each $|\mu|$-bit blocks of each row of **I** (lines 3-5). In this context, each server in the system stores two matrices including the encrypted index **I** and the MAC matrix **T**.

```
$(\sigma';\mathcal{I}') \leftarrow \textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}.\textsf{Update}(f_{id}, \sigma; \mathcal{I})$:
Client:
 1: $(\mathbf{S}, T_f) \leftarrow$ Execute lines 2-4 in Figure 4.53
 2: $\mathcal{J}' \leftarrow$ Select $\lambda$ random indexes of $|\mu|$-bit columns in $\mathbf{I}$
 3: Send $\mathcal{J}'$ to an arbitrary server $S_l$
Server $S_l$: 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 $\Sigma.\textsf{Vrfy}_\theta(\mathbf{I}[i, (j'-1)\cdot|\mu|+1 \ldots j'\cdot|\mu|], \mathbf{T}_l[i, j]) = 0$ then
 8:             return abort
 9: $(\hat{\mathbf{I}}, \mathbf{S}, T_f, \mathbf{c}) \leftarrow$ Execute lines 7-16 in Figure 4.53 with $\mathcal{J} = \{j', \ldots, j'\cdot|\mu|-1\}_{j'\in\mathcal{J}'}$
10: $\hat{\mathbf{T}}[i, j'] \leftarrow \Sigma.\textsf{Mac}_\theta(\hat{\mathbf{I}}[i, (j'-1)\cdot|\mu|+1 \ldots j'\cdot|\mu|])$ 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 $\ell$ servers
Server: each $S_l \in \{S_1, \ldots, S_\ell\}$ 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 4.53
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 $\ell$ servers, and $\sigma'$ is the updated client state
```

Figure 4.62: $\textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}$ update protocol.

Figure 4.61 presents the search protocol of $\textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}$, which is extended from the search protocol of semi-honest $\textsf{ODSE}^{\textsf{wo}}_{\textsf{xor}}$ to be secure against malicious adversary. Specifically, the client generates XOR-PIR queries for ℓ servers similar to the semi-honest $\textsf{ODSE}^{\textsf{wo}}_{\textsf{xor}}$ scheme (line 1). Each server 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 $\textsf{ODSE}^{\textsf{wo}}_{\textsf{xor}}$ scheme (line 11). Otherwise, the client aborts and notifies that at least one of the servers is malicious (line 10).

Figure 4.62 presents the update protocol of $\textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}$ extended from the semi-honest $\textsf{ODSE}^{\textsf{wo}}_{\textsf{xor}}$ for malicious security. Instead of downloading λ random 1-bit columns as in the semi-honest $\textsf{ODSE}^{\textsf{wo}}_{\textsf{xor}}$, 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).

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

Figure 4.63: $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ 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 4.55
2: Send $[\![\mathbf{e}]\!]_l$ to $S_l$ for each $l \in \{1, \ldots, \ell\}$
Server: each $S_l \in \{S_1, \ldots, S_\ell\}$ receiving $[\![\mathbf{e}]\!]_l$ do:
3: $([\![b_1]\!]_l, \ldots, [\![b_{2n'}]\!]_l) \leftarrow$ Execute lines 4-8 in Figure 4.55
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 $\ell$ servers
6: $\mathcal{X} \leftarrow$ Select $t+1$ servers among $\ell$ servers
7: $\mathcal{B}_j \leftarrow \{[\![b_j]\!]_x\}_{x \in \mathcal{X}, j \in [2n']}$, $\mathcal{T}_j \leftarrow \{[\![\mu_j]\!]_x\}_{x \in \mathcal{X}, j \in [\frac{2n}{|\mu|}]}$
8: $\mathbf{I}[i, *] \leftarrow$ Execute lines 10-12 in Figure 4.55
9: for $j = 1 \ldots, 2n/|\mu|$ do
10:     $\mathbf{T}[i, j] \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\mathcal{T}_j, t)$
11:     if $\Sigma.\mathsf{Vrfy}_\theta(\mathbf{I}[i, (j-1) \cdot |\mu|+1 \ldots j \cdot |\mu|], \mathbf{T}[i, j]) = 0$ then
12:         if all distinct subset $\mathcal{X}$ have been processed then
13:             return abort
14:         $\mathcal{X} \leftarrow$ Select another set of $t+1$ servers and goto line 7
15: $\mathcal{J} \leftarrow$ Execute lines 13 -17 in Figure 4.55
16: return $(\mathcal{R}; \perp)$, where $\mathcal{R}$ contains file IDs at column indexes in $\mathcal{J}$
```

Figure 4.64: $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ search protocol.

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).

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 [81, 185] 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 [185]), which might increase the deployment cost. Another approach is to harness the t-out-of-ℓ threshold property

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

Figure 4.65: $\textsf{MR-ODSE}^{\textsf{wo}}_{\textsf{ro}}$ update protocol.

```
$(\sigma,\mathcal{I}) \leftarrow \textsf{MR-ODSE}^{\textsf{wo}}_{\textsf{it}}.\textsf{Setup}(\mathbf{f})$:
$(\langle \mathbf{I}_1,\ldots,\mathbf{I}_\ell \rangle, T_w, T_f, \langle b_{11},\ldots,b_{m2n'} \rangle) \leftarrow$ Execute lines 1-5 in Figure 4.58
$\alpha \xleftarrow{\$} \mathbb{F}_p$
$(\mathbf{T}_1[i,j],\ldots,\mathbf{T}_\ell[i,j]) \leftarrow \textsf{SSS.Create}(\alpha \cdot b_{ij}, t)$ for $i = 1,\ldots,m$ and for $j = 1,\ldots,2n'$
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 $\sigma \leftarrow (\alpha, T_w, T_f)$
```

Figure 4.66: $\textsf{MR-ODSE}^{\textsf{wo}}_{\textsf{it}}$ setup algorithm.

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

The index structure of $\textsf{MR-ODSE}^{\textsf{wo}}_{\textsf{ro}}$ is identical to that of $\textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}$. Thus, its setup algorithm is identical to that of $\textsf{MD-ODSE}^{\textsf{wo}}_{\textsf{xor}}$, where the MAC tag is created for each $|t|$-bit blocks in each row of the encrypted index (Figure 4.63).

Figure 4.64 outlines the search protocol of $\textsf{MR-ODSE}^{\textsf{wo}}_{\textsf{ro}}$ extended from that of $\textsf{ODSE}^{\textsf{wo}}_{\textsf{ro}}$ for malicious security. For each time of oblivious keyword search, the client creates SSS-based PIR query as in the semi-honest $\textsf{ODSE}^{\textsf{wo}}_{\textsf{ro}}$ (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) .

```
$(\mathcal{R};\perp) \leftarrow \mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}.\mathsf{Search}(w,\sigma;\mathcal{I})$:
Client:
 1: $(i, \langle [\![\mathbf{e}]\!]_1, \ldots, [\![\mathbf{e}]\!]_\ell \rangle) \leftarrow$ Execute lines 1-2 in Figure 4.58
 2: Send $[\![\mathbf{e}]\!]_l$, to $S_l$ for each $l \in \{1, \ldots, \ell\}$
Server: each $S_l \in \{S_1, \ldots, S_\ell\}$ receiving $[\![\mathbf{e}]\!]_l$ do
 3: $([\![b_1]\!]_i, \ldots, [\![b_{2N'}]\!]_i) \leftarrow$ Execute lines 4 -5 in Figure 4.58
 4: $[\![\mu_j]\!]_l \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Retrieve}([\![\mathbf{e}]\!]_l, \mathbf{T}_l[*, j])$ 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 $\ell$ servers
 6: $\mathcal{X} \leftarrow$ Select $2t+1$ servers among $\ell$ 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 4.58
 9: for $j = 1 \ldots, 2n'$ do
10:     $\mu_j \leftarrow \mathsf{PIR}^{\mathsf{sss}}.\mathsf{Reconstruct}(\mathcal{T}_j, 2t)$
11:     if $(\alpha \cdot \beta_j \neq \mu_j)$ then
12:         if (all distinct subset $\mathcal{X}$ have been processed) then
13:             return abort
14:         $\mathcal{X} \leftarrow$ Select another set of $2t+1$ servers and goto line 7
15: $\mathcal{J} \leftarrow$ Execute lines 9-10 in Figure 4.58
16: return $(\mathcal{R};\perp)$, where $\mathcal{R}$ contains file IDs at column indexes in $\mathcal{J}$
```

Figure 4.67: $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ search protocol.

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).

The update protocol in $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ is similar to that of $\mathsf{MD\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ (Figure 4.65). 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.

We present $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$, the extended version of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ that inherits all properties of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ (*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.

```
$(\sigma';\mathcal{I}') \leftarrow \mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}.\mathsf{Update}(f_{id},\sigma;\mathcal{I})$:
Client:
 1: $(\mathcal{J},\mathbf{S},T_f) \leftarrow$ Execute lines 1-5 in Figure 4.59
 2: Send $\mathcal{J}$ to $\ell$ servers $(S_1,\ldots,S_\ell)$
Server: each $S_l \in \{S_1,\ldots,S_\ell\}$ 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 $\ell$ 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 4.59
 7: for $i = 1\ldots m$ do
 8:     for each $j \in \mathcal{J}$ do
 9:         if $(\alpha\cdot\mathbf{I}'[i,j] \neq \mathbf{T}[i,j])$ then
10:             if (all distinct subset $\mathcal{X}$ have been processed) then
11:                 return abort
12:             $\mathcal{X} \leftarrow$ 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 4.59
14: $(\hat{\mathbf{T}}_1[i,j],\ldots,\hat{\mathbf{T}}_\ell[i,j]) \leftarrow \mathsf{SSS.Create}(\alpha\cdot b'_{ij},t)$ 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 $S_l$ for $l = 1,\ldots,\ell$
Server: each $S_l \in \{S_1,\ldots,S_\ell\}$ 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 4.59
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 $\ell$ servers and $\sigma'$ is updated client state
```

Figure 4.68: $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ update protocol.

$\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ follows the principles in the semi-honest $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ scheme to create the share index (Figure 4.66, 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.

Figure 4.67 presents the search protocol of $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ extended from that of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ for malicious security. The extension follows the line of the $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ scheme. Specifically, the servers 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 $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ scheme to obtain the final search result (line 15).

Figure 4.68 presents the update protocol of $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$. 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 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ 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.

4.6.2.5 Experimental Evaluation

We fully implemented all ODSE schemes in C++. We used `Google Sparsehash` library [4] to implement position maps T_f and T_w. We utilized `Intel AES-NI` library [80] to implement AES-CTR encryption/decryption in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ schemes. We leveraged `Shoup NTL` library [162] for pseudo-random number generator and arithmetic operations over finite field. We used `ZeroMQ` library [3] 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.

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.

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 [136] so that our keyword distribution and query pattern are similar to [136].

We compared ODSE with a standard DSSE scheme [35], 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 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ schemes, and three servers for $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ scheme. We selected $\lambda = 4$ for $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$, and $\lambda' = 4$ with $\mathbb{F}_p$ where p is a 16-bit prime for $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ schemes $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$. 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 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ schemes to be two, three and four, respectively to handle one adversary. We then increased the number of servers to allow more malicious servers.
- Standard DSSE: We selected one of the most efficient DSSE schemes by Cash *et al.* in [35] (*i.e.*, $\Pi^{\mathsf{dyn}}_{\mathsf{2lev}}$ variant) to demonstrate the performance gap between ODSE and the standard DSSE. We estimated the performance of $\Pi^{\mathsf{dyn}}_{\mathsf{2lev}}$ 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 [1] 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.

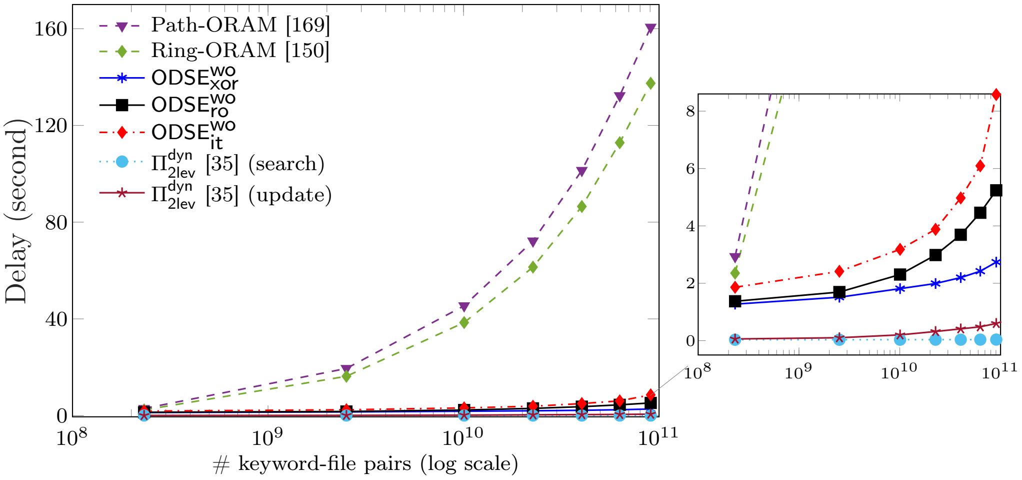


Figure 4.69: Latency of semi-honest ODSE schemes and their counterparts.

- Using generic ORAM atop DSSE encrypted index: We selected *non-recursive* Path-ORAM [169] and Ring-ORAM [150], 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 [67] 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 [136] along with the round-trip optimization as in [67]. Note that these estimates are also conservative, where memory access delays were excluded, and cryptographic operations were optimized and parallelized for an objective comparison.

Figure 4.69 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 5). ODSE offers a higher security than standard DSSE at the cost of a longer delay. Nevertheless, ODSE schemes are 3×-57× 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, $\Pi^{\mathsf{dyn}}_{\mathsf{2lev}}$ cost 36 milliseconds and 600 milliseconds to finish a search and update operation, respectively. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$, 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 [67].

We present the separate delay for the search and update operations in ODSE schemes in Table 4.8. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ is the most efficient in terms of search, whose delay was less than 1 second. This is due to the fact that $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ only triggers XOR operations and the size of the search query is minimal (*i.e.*, a binary string). $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ are more robust (*e.g.*, malicious tolerant) and 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 4.8. In the following section, we dissect the end-to-end delay of ODSE schemes to understand which factors contributing the most to their performance.

Table 4.8: Comparison of ODSE and its counterparts.

Scheme	Security				Delay (second)		Distributed Setting[†]	
	Forward privacy	**Backward privacy**	**Hidden access pattern[‡]**	**Encrypted index[*]**	**Search**	**Update**	**Privacy level**	**Improved Robustness**
Standard DSSE [35]	✗	✗	✗	Computational	0.036	0.62	-	-
Path-ORAM[169]	✓	✓	Computational	Computational	160.6		-	-
Ring-ORAM [150]	✓	✓	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 §4.6.2.5.

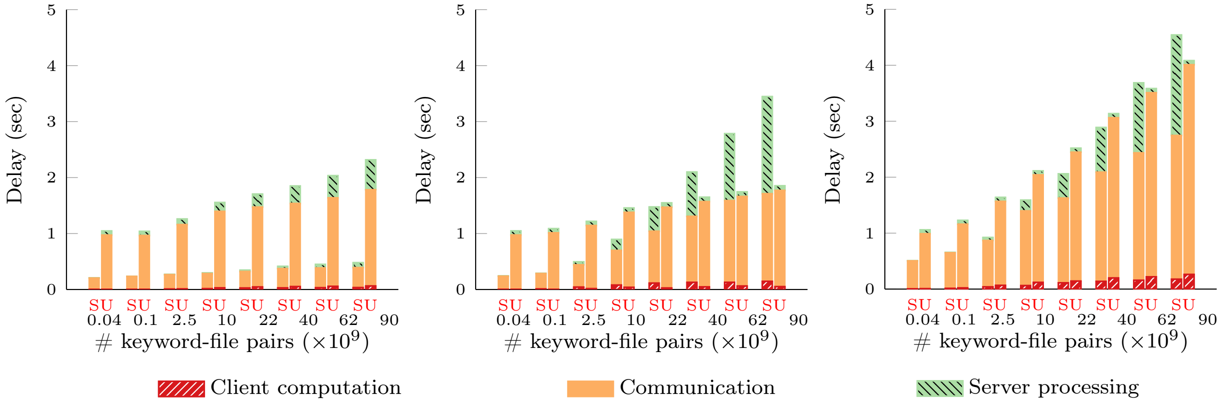
[*] The encrypted index in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ is information-theoretically secure because it is SSS. Other schemes employ IND-CPA encryption so that their index is computationally secure (see §4.6.2.3).

[‡] 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.

Figure 4.70 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.

As shown in Figure 4.70, 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 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.



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

Figure 4.70: Detailed search (S) and update (U) costs of semi-honest ODSE schemes.

Data transmission is the most dominating factor in the delay of ODSE schemes. The communication cost of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ is the smallest among all ODSE schemes since the size of search query and the data transmitted from servers are only binary strings. In $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ 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.

The cost of PIR operations in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ is negligible as it uses XOR tricks. The PIR computation overhead in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ 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. $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ 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 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, we stored the matrix row-wise for row-friendly access to permit efficient XOR

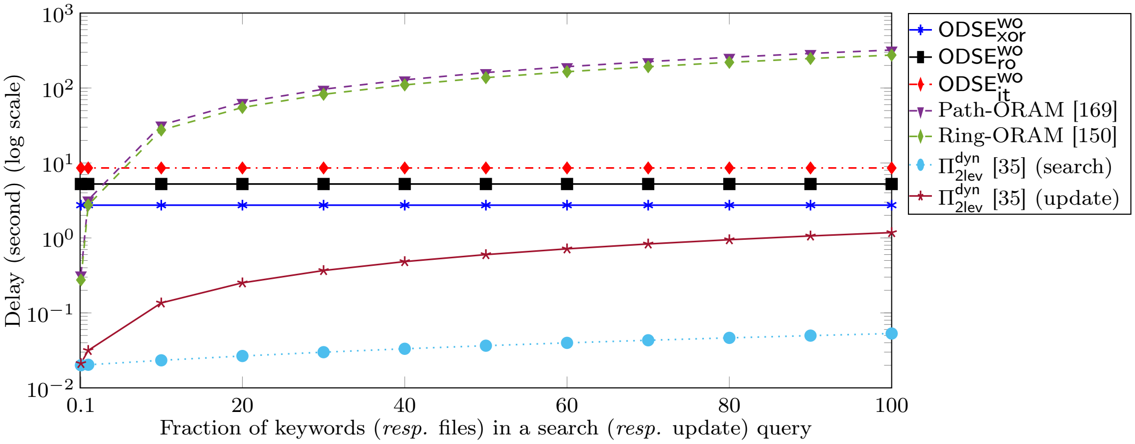
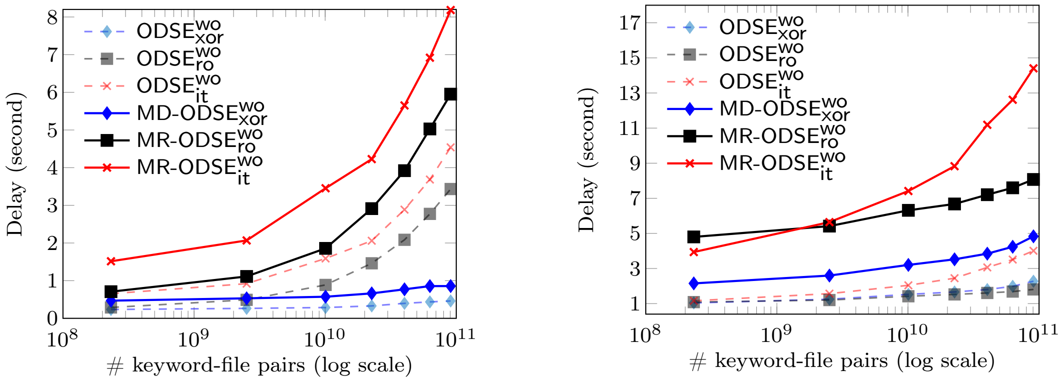


Figure 4.71: Delay of semi-honest ODSE with different query sizes.

operations during search. However, this requires file update to access non-contiguous memory blocks. Hence, the file update in $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ incurred a higher memory access delay than that of $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ as shown in Figure 4.70.

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 $\mathsf{ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ scheme). Empirically, with the same database size discussed above, the client requires approximately 22 MB in all ODSE schemes.

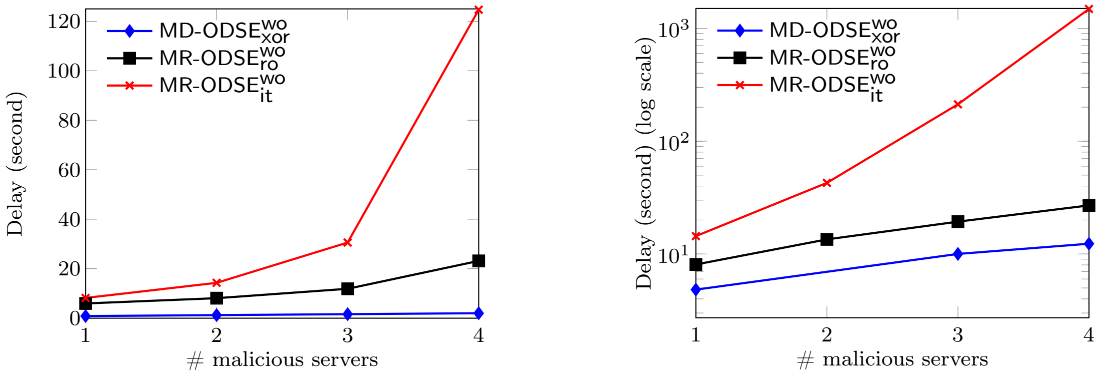
We studied the performance of our schemes and their counterparts in the context of various keyword and file numbers involved in search and update operations that we refer to as “query size". As shown in Figure 4.71, ODSE schemes are more efficient than using generic ORAMs when more than 5% of keywords/files 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



(a) search

(b) update

Figure 4.72: Delay of maliciously-secure ODSE schemes with one malicious adversary.



(a) search

(b) update

Figure 4.73: Delay of maliciously-secure ODSE schemes with several malicious servers.

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 [136] on the performance limitations of generic ORAM and DSSE composition, wherein we used the same dataset for our experiments.

we present the performance of maliciously-secure ODSE schemes described in §4.6.2.4. Figure 4.72 presents the search and update delay of $\mathsf{MD\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ 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 $\mathsf{MD\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$, $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ 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 $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ schemes are around three times slower than that of their semi-honest version. The main reason is that $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ 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, $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ schemes need $2t + 1$ and $3t + 1$ servers in total to be robust against t number of malicious servers, respectively. Figure 4.73 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 incurs not only the client bandwidth overhead to communicate with more servers, but also the client computation overhead. In the worst case, $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ 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 $\mathsf{MD\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{xor}}$ 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

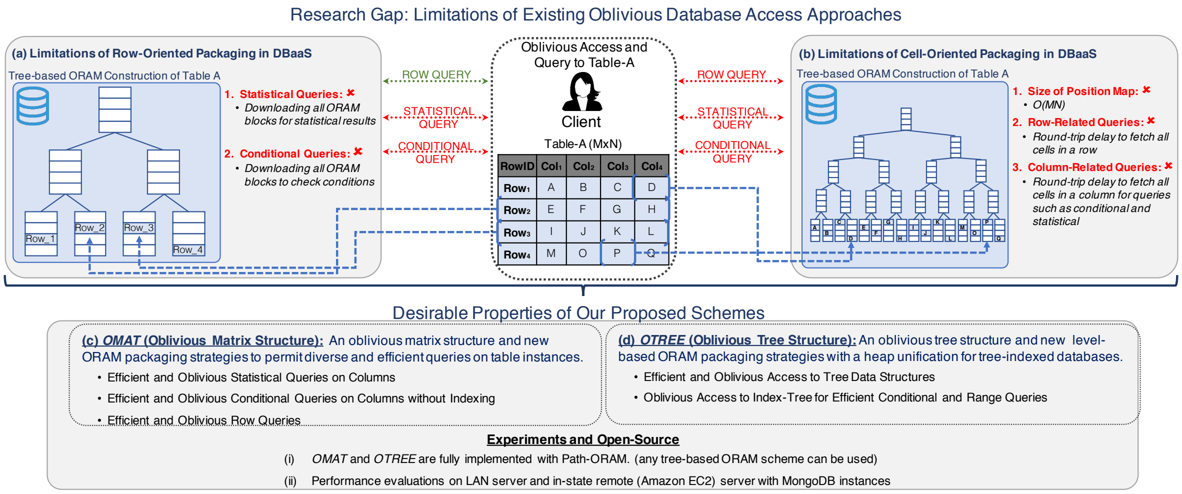
MAC tag (without trying aggressively to find an alternative authentic block as in $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{ro}}$ and $\mathsf{MR\text{-}ODSE}^{\mathsf{wo}}_{\mathsf{it}}$ schemes).

4.6.3 OMAT and OTREE: Oblivious Data Structures for Database Services

We propose two efficient oblivious data structures that permit various types of queries on the encrypted databases: *(i)* Our first scheme is referred to as *Oblivious Matrix Structure* (OMAT). The main idea behind OMAT is to create an oblivious matrix structure that permits efficient queries over table objects in the database not only for the row but also column dimension. This is achieved via various strategies that are specifically tailored for the matrix structure with a delicate balance between the query diversity and the ORAM bandwidth overhead. This allows OMAT to perform various types of oblivious queries without streaming a large number of ORAM blocks or maintaining a very large position map at the client. *(ii)* Our second scheme is referred to as *Oblivious Tree Structure* (OTREE), which is designed for oblivious accesses on tree-indexed database instances. Given a column whose values can be sorted into a tree structure (*i.e.*, numeric values), OTREE allows efficient oblivious conditional queries (*e.g.*, a range query).

We illustrate desirable properties of our schemes in Figure 4.74-(c,d), and further discuss them as follows.

- *Highly efficient and diverse oblivious queries:* OMAT supports a diverse set of queries to be executed with ORAM. Specifically, OMAT permits oblivious statistical queries over value-based columns such as `SUM`, `AVG`, `MAX` and `MIN`. Moreover, oblivious queries on rows (*e.g.*, insert, update) can be executed on an attribute with a similar cost. As shown in Table 4.9, with the given parameters and experimental setup, executing a column-related query such as statistical or conditional query with OMAT is approximately 28× more communication efficient than that


Research Gap: Limitations of Existing Oblivious Database Access Approaches
(a) Limitations of Row-Oriented Packaging in DBaaS
Tree-based ORAM Construction of Table A
Row_1
Row_2
Row_3
Row_4
1. Statistical Queries: ✗
• Downloading all ORAM blocks for statistical results
2. Conditional Queries: ✗
• Downloading all ORAM blocks to check conditions
ROW QUERY
STATISTICAL QUERY
CONDITIONAL QUERY
Oblivious Access and Query to Table-A
Client
Table-A (MxN)
ROW QUERY
STATISTICAL QUERY
CONDITIONAL QUERY
(b) Limitations of Cell-Oriented Packaging in DBaaS
Tree-based ORAM Construction of Table A
A
B
C
D
E
F
G
H
I
J
K
L
M
O
P
Q
1. Size of Position Map: ✗
• O(MN)
2. Row-Related Queries: ✗
• Round-trip delay to fetch all cells in a row
3. Column-Related Queries: ✗
• Round-trip delay to fetch all cells in a column for queries such as conditional and statistical
Desirable Properties of Our Proposed Schemes
(c) OMAT (Oblivious Matrix Structure): An oblivious matrix structure and new ORAM packaging strategies to permit diverse and efficient queries on table instances.
• Efficient and Oblivious Statistical Queries on Columns
• Efficient and Oblivious Conditional Queries on Columns without Indexing
• Efficient and Oblivious Row Queries
(d) OTREE (Oblivious Tree Structure): An oblivious tree structure and new level-based ORAM packaging strategies with a heap unification for tree-indexed databases.
• Efficient and Oblivious Access to Tree Data Structures
• Oblivious Access to Index-Tree for Efficient Conditional and Range Queries
Experiments and Open-Source
(i) OMAT and OTREE are fully implemented with Path-ORAM. (any tree-based ORAM scheme can be used)
(ii) Performance evaluations on LAN server and in-state remote (Amazon EC2) server with MongoDB instances

RowID	Col$_1$	Col$_2$	Col$_3$	Col$_4$
Row$_1$	A	B	C	D
Row$_2$	E	F	G	H
Row$_3$	I	J	K	L
Row$_4$	M	O	P	Q

Figure 4.74: OMAT and OTREE motivation.

of RowPKG and this enables OMAT to perform queries approximately 13× faster than that of RowPKG. Compared to ODS-2D, although OMAT is only 1.6 × more communication-efficient, it performs approximately 5× faster in practice due to the large number of additional round-trip delays. OTREE achieves better performance than ODS for obliviously accessing the database index, which is constructed from the values of a column as a tree data structure. The communication cost of OTREE is 1.6× less than that of ODS without caching. This gain can be increased up to 3.2× with the caching strategy.

- *Generic instantiations from tree-based ORAM schemes:* We notice that *any* tree-based ORAM scheme (*e.g.*, [169, 179]) can be used for both OMAT and OTREE instantiations. This provides a flexibility in selecting a suitable underlying ORAM scheme, which can be adjusted according to the performance requirements of specific applications. Note that we instantiated our schemes with Path-ORAM [169] due to its efficiency, simplicity and not requiring any server-side computation.

- *Comprehensive experiments and evaluations* We implemented OTREE, OMAT, and their counterparts under the same framework. We evaluated their performance with a MongoDB database

Table 4.9: Transmission cost and client storage for compared schemes.

Scheme	Communication Cost[a]	Efficiency[b]	Client Storage[c]	End-to-End Delay[d]	
				Moderate Network	High Network
single column-related query (e.g., statistical, conditional queries)					
RowPKG [42]	$Z \cdot (\_1 \cdot N) \cdot (2M - 1)$	1.00	$O(M \cdot N) \cdot w(1)$	6096 s	776 s
ODS-2D [182]	$(M/4) \cdot [Z \cdot (16 \cdot \lvert b \rvert_1) \cdot \log_2(M \cdot N/16)]$	17.04	$O(M \cdot \log(M \cdot N)) \cdot w(1)$	1245 s	292 s
OMAT	$Z^2 \cdot (\lvert b \rvert_1 \cdot M) \cdot \log_2(N)$	**28.44**	$O(M \cdot \log(N)) \cdot w(1)$	**475 s**	**60 s**
single row-related query (e.g., insert/delete/update queries)					
RowPKG [42]	$Z \cdot (\lvert b \rvert_2 \cdot N) \cdot \log_2(M)$	**1.00**	$O(N \cdot \log(M)) \cdot w(1)$	**567 ms**	**56 ms**
ODS-2D [182]	$(N/4) \cdot [Z \cdot (16 \cdot \lvert b \rvert_2) \cdot \log_2(M \cdot N/16)]$	0.19	$O(N \cdot \log(M \cdot N)) \cdot w(1)$	2380 ms	350 ms
OMAT	$Z^2 \cdot (\lvert b \rvert_2 \cdot N) \cdot \log_2(M)$	0.25	$O(N \cdot \log(M)) \cdot w(1)$	2032 ms	128 ms
traversal on database tree index (e.g., range queries)					
non-caching					
ODS-Tree [182]	$2 \cdot Z_1 \cdot \lvert b \rvert \cdot (H+1)^2$	1.00	$O(H) \cdot w(1)$	7929 ms	1318 ms
OTREE	$Z_2 \cdot \lvert b \rvert \cdot (H+1) \cdot (H+2)$	**1.60**	$O(H) \cdot w(1)$	**3762 ms**	**592 ms**
half-top caching					
ODS-Tree [182]	$2 \cdot Z_1 \cdot \lvert b \rvert \cdot \lceil \frac{H+1}{2} \rceil \cdot (H+1)$	1.00	$O(\sqrt{2^H}) + O(H) \cdot w(1)$	5979 ms	1008 ms
OTREE	$Z_2 \cdot \lvert b \rvert \cdot \lceil \frac{H+1}{2} \rceil \cdot (\lceil \frac{H+1}{2} \rceil + 1)$	**3.20**	$O(\sqrt{2^H}) + O(H) \cdot w(1)$	**1676 ms**	**272 ms**

- *Table Notations*: M and N denote the total number of (real) rows and columns in the matrix data structure, respectively. H is the height of the tree data structure. Z and $|b|$ denote the bucket size and size of each block (in bytes), respectively.
- *Settings*: We instantiate our schemes and their counterparts with underlying Path-ORAM for a fair comparison. The bottom half of the table compares OTREE and ODS-Tree when combined with tree-top caching technique proposed in [127], in which we assume the top half of tree-based ORAM is cached on the client during all access requests.
- *Server Storage*: All of the oblivious matrix structures require $O(MN)$ server storage, however, the storage of OMAT is a constant (*e.g.*, $Z = 4$) factor larger than others. OTREE is twice more storage efficient than ODS.

[a] Represents the total cost in terms of bytes to be processed (*e.g.*, communication/computation depends on the underlying ORAM scheme) between the client and the server for each request. For OMAT , ODS-2D and RowPKG, the cost is for one access operation per query. For OTREE and ODS, the cost is for traversing an arbitrary path in a binary tree.

[b] Denotes the communication cost efficiency compared to chosen baseline, where $Z = 4, |b|_1 = 64, |b|_2 = 128, M = 2^{15}, N = 2^9$ for ODS-2D, RowPKG and OMAT, and $Z_1 = 4, Z_2 = 5$ (for stability), $|b| = 4096, H = 20$ for ODS-Tree and OTREE.

[c] Client storage consists of the worst-case stash size to keep fetched data. Additionally, the position map of OMAT and RowPKG are $O((M + N)\log(M+N))$ and $O(M \cdot \log(M))$, respectively. For ODS based structures and OTREE, position map requires $O(1)$ storage due to pointers and half-top cached blocks are also included in client storage.

[d] The delays were measured with a MongoDB instance running on Amazon EC2 connected with the client on two different network settings.

instance unning on a remote AmazonEC2 server with two different network settings: (1) moderate-speed network and (2) high-speed network. This permits us to observe the impact of real network and cloud environment.

We now present our proposed oblivious data structures, which are specially designed for efficient operations in database settings. We propose two schemes including *Oblivious Matrix Structure* (OMAT) and *Oblivious Tree Structure* (OTREE). OMAT supports efficient oblivious statistical queries on generic table instances, while OTREE supports range and conditional queries on tree-indexed instances. Figure 4.75 outlines the overview of our proposed techniques. For our oblivious data structures, we choose Path-ORAM [169] as the underlying ORAM for the following

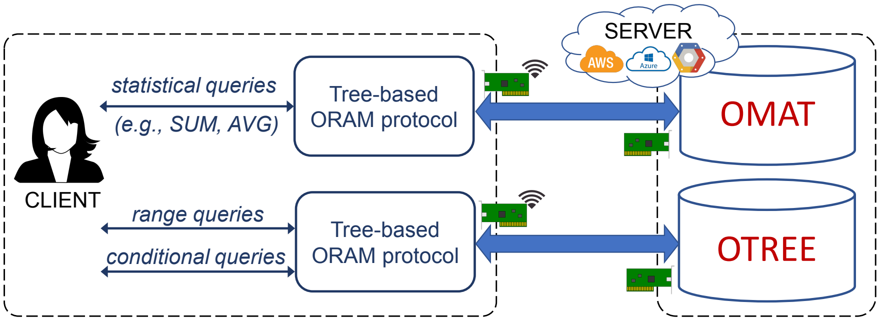
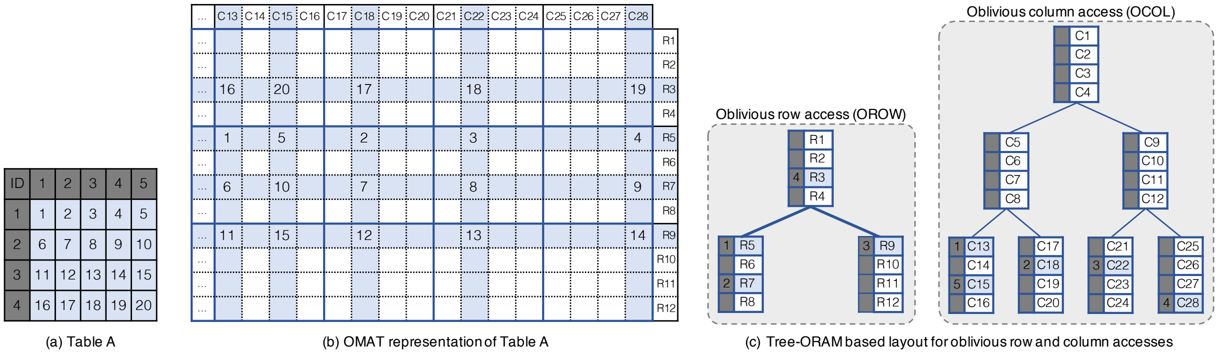


Figure 4.75: Overview of our proposed techniques.



ID	1	2	3	4	5
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12	13	14	15
4	16	17	18	19	20

(a) Table A
(b) OMAT representation of Table A
(c) Tree-ORAM based layout for oblivious row and column accesses
Oblivious row access (OROW)
Oblivious column access (OCOL)
C13 C14 C15 C16 C17 C18 C19 C20 C21 C22 C23 C24 C25 C26 C27 C28
R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12
C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12

Figure 4.76: OMAT structure for oblivious access on table.

reasons: (i) It is simple yet achieves asymptotic efficiency. (ii) Unlike some recent ORAMs [54, 150] that require computations at the server side, it requires only read/write operations. This is useful since such advanced cryptographic operations might not be readily offered by well-known database instances (*e.g.*, MongoDB, MySQL). (iii) The availability of Path-ORAM implementations on existing frameworks (*e.g.*, CURIOUS [20]) enables a fair experimental comparison of the proposed techniques with the state-of-the-art.

4.6.3.1 OMAT*: Oblivious Access on Table Structures*

The direct application of tree-based ORAMs to access encrypted tables in general [20] and database systems in specific [42] have been shown to be inefficient for large datasets. Specifically, if each row in the table is packaged into an ORAM block as in [42], then performing queries to fetch a

```
data ← OMAT.Access(op, dim, $id$):
$b \leftarrow \mathsf{pm}_{\mathsf{dim}}[id].\mathsf{pid}$
if dim = col then
    $\mathsf{pm}_{\mathsf{dim}}[id].\mathsf{pid} \xleftarrow{\$} \{1, \ldots, 2^{\lceil \log_2(N) \rceil - 1}\}$
    $H \leftarrow \lceil \log_2(N) \rceil$
else
    $\mathsf{pm}_{\mathsf{dim}}[id].\mathsf{pid} \xleftarrow{\$} \{1, \ldots, 2^{\lceil \log_2(M) \rceil - 1}\}$
    $H \leftarrow \lceil \log_2(M) \rceil$
▷ Read all rows/columns on the path $\mathcal{P}(b)$
for each $\ell \in \{0, \ldots, H\}$ do
    $\mathbf{S}_{\mathsf{dim}} \leftarrow \mathbf{S}_{\mathsf{dim}} \cup \mathsf{ReadBucket}(\mathsf{dim}, \mathcal{P}(b, \ell))$
data ← Read row/column with $id$ from $\mathbf{S}_{\mathsf{dim}}$
data ← $\mathsf{FilterDummy}(\mathsf{data}, \mathbf{S}_{\neg\mathsf{dim}})$
$\mathbf{S}_{\neg\mathsf{dim}} \leftarrow \mathsf{Update}(\mathbf{S}_{\neg\mathsf{dim}}, \mathsf{pm}_{\mathsf{dim}})$
if op = write then
    $\mathbf{S}_{\mathsf{dim}} \leftarrow (\mathbf{S}_{\mathsf{dim}} \setminus \{(id, \mathsf{data})\}) \cup \{(id, \mathsf{data}^*)\}$
▷ Evict blocks from the stash
for each $\ell \in \{H, \ldots, 0\}$ do
    $\mathbf{S}'_{\mathsf{dim}} \leftarrow \{(id', \mathsf{data}') \in \mathbf{S}_{\mathsf{dim}} | \mathcal{P}(b, \ell) = \mathcal{P}(\mathsf{pm}_{\mathsf{dim}}[id'].\mathsf{pid}, \ell)\}$
    $\mathbf{S}'_{\mathsf{dim}} \leftarrow$ Select $\min(|\mathbf{S}'_{\mathsf{dim}}|, Z)$ blocks from $= \mathbf{S}'_{\mathsf{dim}}$
    $\mathbf{S}_{\mathsf{dim}} \leftarrow \mathbf{S}_{\mathsf{dim}} \setminus \mathbf{S}'_{\mathsf{dim}}$
    $o \leftarrow 1$
    for each $(id', \mathsf{data}') \in \mathbf{S}'_{\mathsf{dim}}$ do
        $\mathsf{pm}[id'].\mathsf{level} \leftarrow \ell$
        $\mathsf{pm}[id'].\mathsf{order} \leftarrow o$ , $o \leftarrow o + 1$
    $\mathsf{WriteBucket}(\mathsf{dim}, \mathcal{P}(b, \ell), \mathbf{S}'_{\mathsf{dim}})$
return data
```

Figure 4.77: OMAT access protocol.

column in such a table (*e.g.*, statistics) would require the client to stream all blocks in the ORAM structure, which might be impractical. On the other hand, packaging each cell in the table into an ORAM incurs a high network delay and client storage overhead. Thus, we investigate on how to translate the table into an oblivious data structure so that each row and column of it can be both accessed efficiently by a given ORAM scheme. Below, we first describe our oblivious data structure and then present our OMAT access scheme on top of it.

The main data structure that we use for oblivious access on a table is a matrix. Given an input table $\mathbf{T}$ of size $M \times N$, we allocate a matrix $\mathbf{M}$ of size $Z \cdot 2^{\lceil \log_2(M) \rceil - 1} \times Z \cdot 2^{\lceil \log_2(N) \rceil - 1}$. We arrange tree-based ORAM building blocks for oblivious access as follows:

The layout of OMAT matrix $\mathbf{M}$ can be interpreted as two logical tree-based ORAMs defined as oblivious rows (denoted as OROW) and oblivious columns (denoted as OCOL) as illustrated in Figure 4.76. That is, the ORAM for row access on OROW is formed by a set of blocks $b_i := (id_i, \mathsf{data}_i)$, where id_i is either a unique identifier if b_i contains the content of a row of the table $\mathbf{T}$ or null otherwise, and $\mathsf{data}_i \leftarrow \mathbf{M}[i, *]$. We group Z subsequent rows in $\mathbf{M}$ to form a bucket (*i.e.*, node) in the OROW structure. Similarly, the ORAM for column access on OCOL is formed by $b_j = (id_j, \mathsf{data}_j)$. Each (bucket) node in OCOL is formed by grouping Z subsequent blocks.

We assign each row $\mathbf{T}[i', *]$ $(i' = 1, \ldots, M)$ and each column $\mathbf{T}[*, j']$ $(j' = 1, \ldots, N)$ with a random leaf node IDs $u_{i'}$ and $v_{j'}$ in OROW and OCOL, respectively. That is, the data of $\mathbf{T}[i, *]$ and $\mathbf{T}[*, j]$ reside in some rows and columns of $\mathbf{M}$ along the assigned paths $\mathcal{P}(u_i)$ in OROW and $\mathcal{P}(v_j)$ in OCOL, respectively. In other words, $\mathbf{M}[i, j] \leftarrow \mathbf{T}[i', j']$, where $\mathbf{M}[i, *] \in \mathcal{P}(u_{i'})$ in OROW and $\mathbf{M}[*, j] \in \mathcal{P}(v_{j'})$ in OCOL. Our construction requires two position maps ($\mathsf{pm}_{\mathsf{row}}$ and $\mathsf{pm}_{\mathsf{col}}$) to store the assigned path for each row $\mathbf{T}[i', *]$ and each column $\mathbf{T}[*, j']$ of table $\mathbf{T}$ in OROW and OCOL, respectively. Our position maps store all necessary information to locate the exact position of a row/column data in the tree-based ORAM structures as $\mathsf{pm} := (\mathsf{id}, \langle \mathsf{pid}, \mathsf{level}, \mathsf{order} \rangle)$, where $0 \leq \mathsf{level} \leq \log_2(N)$ indicates the level of the bucket, in which the row/column with id resides, and $1 \leq \mathsf{order} \leq Z$ indicates its order in the bucket.

We present our OMAT scheme, which is instantiated with Path-ORAM, in Figure 4.77. Specifically, given a *column* (resp. *row*) identifier (id) to be accessed[22], the client retrieves its location from the *column* (resp. *row*) position map (step 1). The client then assigns the *column* (resp. *row*) to a new location selected uniformly at random (steps 2–7). The client reads all *columns* (resp. *rows*) residing on the same path according to tree-based ORAM layout (as depicted in Figure 4.76(c)

[22] The access can be any types of operation such as read/add/delete/modify.

to the stash (steps 8–9). In this case, we modify the original ReadBucket subroutine of Path-ORAM, where it now takes an extra parameter (dim) that indicates the dimension to be read, and outputs the corresponding Z columns/rows in the bucket. The client retrieves the *column* (resp. *row*) with *id* from the stash (step 10). One might observe that according to OMAT structure, the retrieved *column* (resp. *row*) will contain data from dummy *rows* (resp. *columns*) as depicted by empty blue cells in Figure 4.76(b). Therefore, to obtain only the real data of the requested *column* (resp. *row*), the client filters all data from dummy *rows* (resp. *columns*) (step 11). Moreover, since the position of the retrieved column (resp. *row*) is moved to a new random position (steps 2–7), it is required to update all *rows* (resp. *columns*) that are currently stored in the stash at this column (resp. *row*) position to achieve the consistency (step 12). If the access is to update, the client then updates the *column* (resp. *row*) with new data (steps 13–14) Finally, the client performs eviction as described in Path-ORAM to flush *columns* (resp. *rows*) from the stash back to the OMAT structure in the server (steps 15–23).

Notice that all columns/rows are IND-CPA decrypted and re-encrypted as they are read and written to/from the server, respectively. We assume that it is not required to hide the information whether a column or a row is being accessed. However, this can be achieved with the cost of performing oblivious accesses on both row and column (one of them is dummy selected randomly) for each access.

Recall that, in row-oriented packaging, implementing secure statistical queries on a column requires downloading the entire ORAM blocks from the database. In contrast, OMAT structure allows queries such as `add`, `delete`, `update` not only on its row but also on its column dimension. Thus, we can implement statistical queries (*e.g.*, `MAX`, `MIN`, `AVG`, `SUM`, `COUNT`, etc.) over a column in an efficient manner via OMAT. Note that OMAT can also permit conditional query on rows with `WHERE` statement. Similar to statistical queries, the query can be implemented by reading the attribute column on which the `WHERE` clause looks up OCOL first to determine appropriate records that satisfy the condition, and then obliviously fetching such records on OROW structure. For example, assume that we have the following SQL-like conditional search.

```
SELECT * FROM A WHERE C > k
```

It can be implemented by:

1. Read the column C with id' on OCOL as $\mathbf{C}[*, id'] \leftarrow \mathsf{OMAT.Access}(\mathsf{read}, \mathsf{col}, id')$.

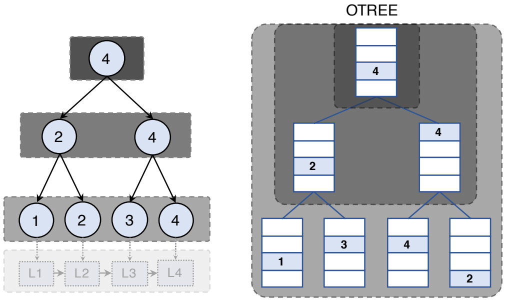
2. Get IDs of rows whose value larger than k, and such IDs are in $\mathsf{pm}_{\mathsf{row}}$ as $\mathcal{I} \leftarrow \{id | id \in \mathsf{pm}_{\mathsf{row}}.id \wedge \mathbf{C}[id, id'] > k\}$

3. Access on OROW to get the desired result as $\mathbf{R}[id, *] \leftarrow \mathsf{OMAT.Access}(\mathsf{read}, \mathsf{row}, id)$, for each $id \in \mathcal{I}$.

The aforementioned approach can work with any unindexed columns. In the next section, we propose an alternative approach that can offer a better performance if the columns can be indexed with certain restrictions.

4.6.3.2 OTREE*: Oblivious Access on Tree Structures*

In the unencrypted database setting, conditional queries can be performed more efficiently, if column values can be indexed by a search-efficient tree data structure (*e.g.*, Range tree, B+ tree, AVL tree). Figure 4.82 illustrates an example of a column indexed by a range tree for (non)-equality/range queries, in which each leaf node points to a node in another linked-list structure that stores the list of matching IDs. We propose an oblivious tree structure called OTREE, in which indexed data for



(a) Tree-like data T (b) OTREE representation of T

Figure 4.78: The OTREE layout for a tree data.

such queries are translated into a balanced tree structure. As in OMAT, OTREE *can be instantiated from any tree-based ORAM scheme.* Notice that oblivious access on a tree was previously studied in [182]. Our method requires less amount of data to be transmitted and processed, since the structure of indexed values (*i.e.*, the tree data structure) is not required to be hidden, and the client is merely required to traverse an arbitrary path of the tree. We present the construction of OTREE as follows.

Given a tree-indexed data $\mathbf{T}$ of height H as input, we first construct the OTREE structure of height H with ORAM buckets as illustrated in Figure 4.78. Then, each node of $\mathbf{T}$ at level ℓ is assigned to a random path and placed into a bucket of OTREE which resides on the assigned path at level ℓ' where $\ell' \leq \ell$. In other words, *any node of* $\mathbf{T}$ *at level* $0 \leq \ell \leq H$ *will reside in a bucket at level* ℓ *or lower in* OTREE. If there is no empty slot in the path, the node will be stored in the stash if OTREE is instantiated with stash-required ORAM schemes (*e.g.*, Path-ORAM).

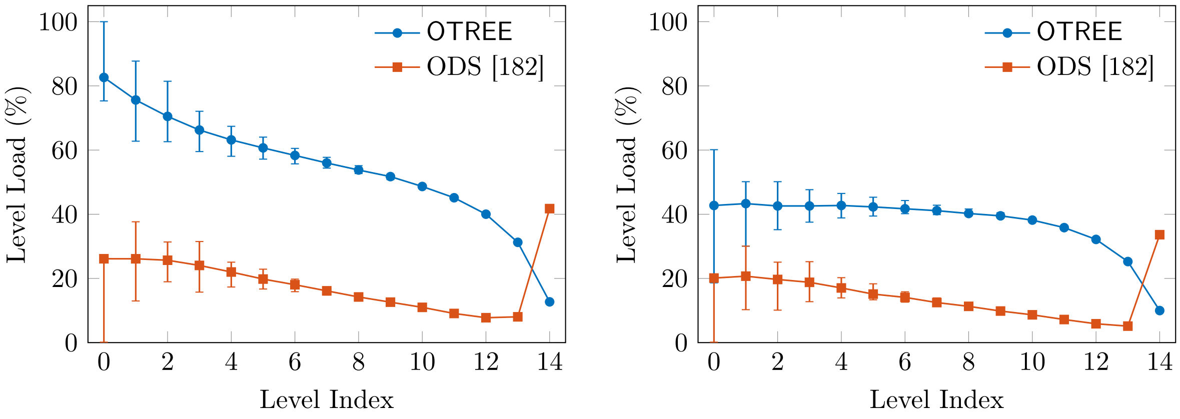
We assume $\mathbf{T}$ is sorted by nodes' *id* and the position of nodes at level ℓ is stored in its parent node at level $\ell - 1$ using the pointer technique proposed in [182]. Hence, each node of $\mathbf{T}$ is considered as a separate block in OTREE structure as: $b := (id, \mathsf{data}, \mathsf{childmap})$, where *id* is the node identifier sorted in $\mathbf{T}$ (*e.g.*, indexed column value), data indicates the node data, and childmap is of structure $\langle id, \mathsf{pos}\rangle$ that stores the position information of node's children.

```
(data) ← OTREE.Access(op, $id$, data$^*$):
1:  $x_0 \leftarrow$ RootPos
2:  $\mathbf{S} \leftarrow \mathbf{S} \cup$ ReadBucket$(\mathcal{P}(x_0, 0), 0)$
3:  $b_0 \leftarrow$ Read block with $id_0 = 0$ from $\mathbf{S}$
4:  for each $\ell \in \{0, \ldots, H-1\}$ do
5:      if compare$(id, id_\ell)$ = go_right then
6:          $(id_{\ell+1}, x_{\ell+1}) \leftarrow b_\ell$.child$[1]$
7:          $b_\ell$.child$[1]$.pos $\xleftarrow{\$} \{0, \ldots, 2^\ell - 1\}$
8:      else
9:          $(id_{\ell+1}, x_{\ell+1}) \leftarrow b_\ell$.child$[0]$
10:         $b_\ell$.child$[0]$.pos $\xleftarrow{\$} \{0, \ldots, 2^\ell\}$
11:     $\mathbf{S} \leftarrow \mathbf{S} \cup$ ReadBucket$(\mathcal{P}_{\ell+1}(x_{\ell+1}, \ell+1))$
12:     $b_\ell \leftarrow$ Read block $id_\ell$ from $\mathbf{S}$
13:     if $id = id_\ell$ then
14:         data $\leftarrow b_\ell$.data
15:         if op = write then
16:             $S \leftarrow (S \setminus \{b_\ell\}) \cup \{(id, \text{data}^*, \text{child})\}$
17:     for each $\ell' \in \{\ell, \ldots, 0\}$ do
18:         $\mathbf{S}' \leftarrow \{b' \in \mathbf{S} : \mathcal{P}_\ell(b'.\text{pos}, \ell') = \mathcal{P}_\ell(b_\ell.\text{pos}, \ell') \wedge b'.\text{level} = \ell\}$
19:         $\mathbf{S}' \leftarrow$ Select $\min(|\mathbf{S}'|, z)$ blocks from $\mathbf{S}'$
20:         $\mathbf{S} \leftarrow \mathbf{S} \setminus \mathbf{S}'$
21:         WriteBucket$(\mathcal{P}_\ell(x_\ell, \ell'), \mathbf{S}')$
22: return data
```

Figure 4.79: OTREE access protocol.

OTREE can be instantiated with any tree-based ORAM schemes (*e.g.*, Ring-ORAM [150], Circuit-ORAM [179]) , as similar to OMAT in §4.6.3.1, by modifying corresponding retrieval/eviction procedures while preserving the constraints of OTREE regarding the deepest level of nodes. OTREE also receives a significant benefit from caching mechanisms like top-tree caching [127], which can speed up bulk access requests.

We give the proposed OTREE scheme instantiated with Path-ORAM in Figure 4.79. Specifically, given the node identifier id to be accessed in the id-sorted tree structure, the client first reads the root bucket of Path-ORAM structure to obtain the root node of the tree (steps 1–3). The client then compares the requested id with the root id to decide which child of the root node should be accessed in the next step. The client accesses this child by reading its path in the Path-ORAM structure from level 0 to level 1. We notice that for each node at level l in the tree to be accessed,



(a) $Z = 4$ (b) $Z = 5$

Figure 4.80: Average bucket load in OTREE.

the client only accesses the path in the Path-ORAM structure up to level l. The process repeats until the desired *id* is found (steps 4–16). Finally, the client performs eviction to flush read nodes back to the Path-ORAM structure, wherein nodes at level l in the tree must reside somewhere in the Path-ORAM structure from level 0 to level l (steps 17–21).

The construction and constraints of OTREE require a stability analysis to ensure that tree-based ORAM scheme on OTREE behaves similarly to ODS in terms of the stash overflow probability. We provide an empirical stability analysis of OTREE with Path-ORAM as follows.

We analyze the stability of OTREE in terms of the average bucket load in each level of the ORAM tree. Intuitively, one would expect an increase in average bucket load near the top of the ORAM tree, and a possible increase in the average client stash size if a Path-ORAM variant (*e.g.*, [127, 150]) is used. We show empirically by our simulations, that OTREE behaves almost similar to ODS with a bucket size of $Z \geq 4$ with Path-ORAM. With $Z = 5$, bucket usage with OTREE structure approaches that of the stationary distribution when using an infinitely large bucket size.

Our empirical study considered experiments with an ORAM tree of height $H = 14$ storing $N = 2^{15} - 1$ blocks. We ran the experiments with different bucket sizes to observe its effect on the

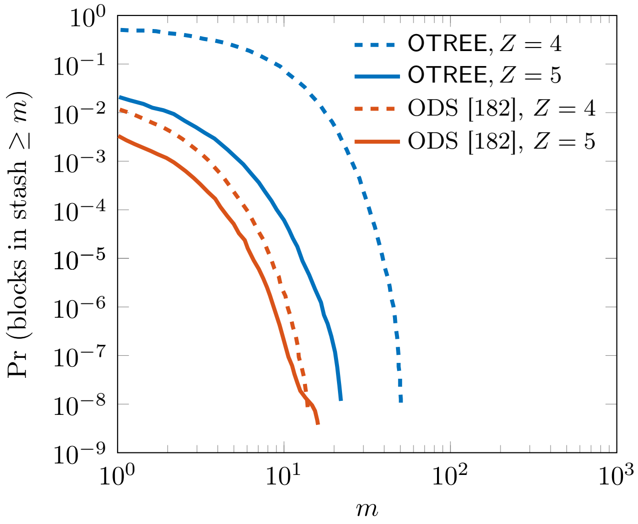
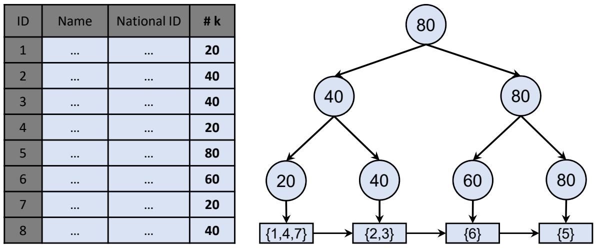


Figure 4.81: Probability of stash size exceeding the threshold.

stash size and bucket usage. We treated ORAM blocks as nodes in a full binary tree of $H = 14$. We inserted nodes into storage according to the breadth-first order via access functions followed by a series of $(H + 1)$-length access requests, each of which consists of accessing a path of nodes from the root to a random leaf node in the binary tree. A single-round experiment was the execution of 2^{14} random root-to-leaf access sequences as described.

Figure 4.80 and Figure 4.81 show the results of these experiments for ODS and OTREE with different bucket sizes. The results were generated by first running 1000 warm-up rounds after the initialization, and then collecting statistics over 1000 test rounds. Figure 4.80 depicts that with a bucket size $Z = 5$, buckets near the root of the OTREE structure contain roughly two non-empty blocks (one more than the average number of blocks assigned to them). Figure 4.81 illustrates that with $Z \leq 4$, the probability of the stash size exceeding $O(H)$ for OTREE diminishes quickly. These results suggest that using $Z = 5$ for OTREE in order to make underlying ORAM scheme in OTREE behaves similarly to that with $Z = 4$ on ODS.



ID	Name	National ID	# k
1	...	...	20
2	...	...	40
3	...	...	40
4	...	...	20
5	...	...	80
6	...	...	60
7	...	...	20
8	...	...	40

(a) Table A

(b) Database Index as Range Tree for Values of k

Figure 4.82: OTREE example for range queries.

We exemplify an implementation of a database index structured as OTREE for conditional queries as follows: Consider a column whose values are indexed by a sorted tree **T** of height h by putting distinct values as keys on leaf nodes as depicted in Figure 4.82. The leaf nodes of **T** points to a node ID in a linked-list structure that contains a list of matching IDs with the key. We translate **T** into OTREE, where each node at level $\ell < H$ stores the position maps of its children. We store a list of IDs in each linked-list node using an inverted index with compression. As the data structure for the linked-list, we employ ODS to store it in another ORAM structure (see [182] for details). Hence, each leaf node of **T** stores the position map of a linked-list node in ODS it points to. An example of a given conditional query is `SELECT` $*$ `FROM A WHERE C` $=$ `k`, where the column `C` is indexed into OTREE. It can be executed obliviously as follows.

1. Traverse a path with OTREE to get a leaf node as $b \leftarrow \mathsf{OTREE.Access}(k)$.

2. Get ID and position map of linked-list node which b points to as $(id, \mathsf{pos}) \leftarrow b.\mathsf{childmap}$.

3. Access on ODS to get the desired result as $\mathcal{R} \leftarrow \text{ODS}.\mathsf{Access}(id, \mathsf{pos}, \cdot)$.

The overall cost for this approach is: $O(\log^2 N + k \cdot O(\log(N))$, where k is the distance from the first element of the linked-list. The first part is the overhead of OTREE and the second part is the overhead of ODS (without padding).

4.6.3.3 Security Analysis

Corollary 8. Accessing OMAT *leaks no information beyond* (i) *the size of rows and columns,* (ii) *whether the row or column dimension being accessed, given that the ORAM scheme being used on top is secure by Definition 5.*

Proof. Let $\mathbf{M}$ be an OMAT structure consisting of two logical tree-based ORAM structures OROW and OCOL with dimensions M and N, respectively. Let the bit $|b|= 0$ if the query is on OROW and $|b|= 1$, otherwise. A construction providing OMAT leaks no information about the location of a node u being accessed in $\mathbf{M}$ beyond the bit $|b|$ and dimensions (M, N). This is due to the fact that OMAT uses a secure ORAM that satisfies Definition 5 to access each block of OROW and OCOL in $\mathbf{M}$. Thus, as long as the node accessed within OROW or OCOL is not distinguishable from any other node within that OROW and OCOL through the number of access requests, it is indistinguishable by Definition 5. □

Note that the information on whether the row or column was accessed can be hidden by performing a simultaneous row and column access on both dimensions for each query. This poses a security-performance trade-off. One can also hide the size of row and column by setting OMAT matrix with equal dimensions, but this may introduce some cost for certain applications.

Corollary 9. Accessing OTREE *leaks no information about the actual path being traversed, given that the ORAM scheme being used on top is secure by Definition 5.*

Proof. Let $\mathbf{T}$ be a tree data structure of height H. Let $\mathbf{T}_\ell$ be the set of nodes at level $0 \leq \ell \leq H$ in the tree. A construction providing OTREE leaks no information about the location of a node $u \in \mathbf{T}_\ell$ being accessed in the tree beyond that it is from $\mathbf{T}_\ell$. This is due to OTREE uses a secure ORAM that satisfies Definition 5 to access each level of the tree. Thus, as long as a node accessed within level ℓ is not distinguishable from any other node within that level through the number of access requests, it will be indistinguishable according to Definition 5. □

There are several side-channel attacks on Path-ORAM (*e.g.*, [13, 64]) when it is executed by the secure CPU playing on behalf of the ORAM client. In this context, since the secure CPU resides in the untrusted party, the adversary has a partial view on it to exploit the timing leakage (*e.g.*, [13]). In our model, we assume that the client is fully trusted and it is totally apart from the adversary view (*i.e.*, untrusted database server). Therefore, we do not consider these side-channel leakages due to the difference between our model and the secure CPU context.

4.6.3.4 Experimental Evaluation

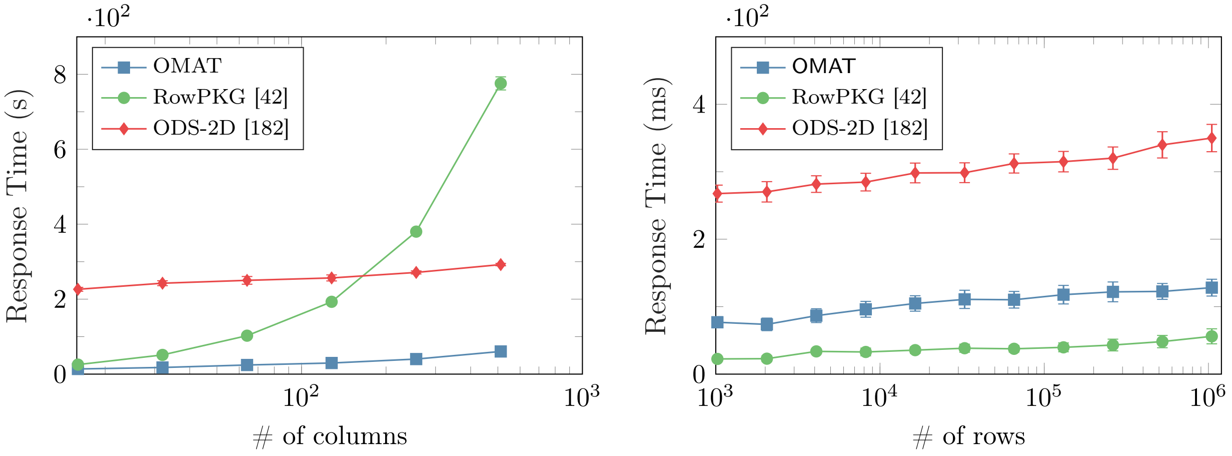
We implemented our schemes and their counterparts on CURIOUS framework [20]. We integrated additional functionalities into the framework to perform batch read/write operations to prevent unnecessary round-trip delays, and also to communicate with MongoDB instance via MongoDB Java Driver. We chose MongoDB as our database and storage engine. We preferred MongoDB since its Java Driver library is well-documented and easy to use. Moreover, it supports batch updates without restrictions, which is important for consistent performance analysis.

We created our database table with randomly generated data with a different number of rows, columns, and field sizes. We then used the table to construct tree-based ORAMs for compared schemes. For instance, in OMAT, we created OROW and OCOL structures from this table, while an oblivious tree structure is created for OTREE.

For our experiments, we used two different client machines on two different network settings: (i) A desktop computer that runs CentOS 7.2 and is equipped with Intel Xeon CPU E3-1230, 16 GB RAM; (ii) A laptop computer that runs Ubuntu 16.04 and is equipped with Intel i7-6700HQ, 16 GB RAM. For our remote server, we used AmazonEC2 with t2.large instance type that runs Ubuntu Server 16.04. While the connection between the desktop and the server was a *high-speed network* with download/upload speeds of 500/400 Mbps and an average latency of 11 ms, the connection between the laptop and the server was a *moderate-speed network* with download/upload speeds of 80/6 Mbps and an average latency of 30 ms.

We evaluated the performance of our schemes and their counterparts based on the following metrics: (*i*) The Response time (*i.e.*, end-to-end delay) including decryption, re-encryption and transmission times to perform a query; (*ii*) Client storage including the size of stash and position map; (*iii*) Server storage including the size of OMAT or OTREE. We compared the response times of OMAT and its counterparts for both row- and column-related queries (*e.g.*, statistical, conditional). For OTREE and ODS-Tree, we compared the response times of traversing an arbitrary path on the tree-indexed database. To measure the end-to-end delay, we used the `std::chrono` C++ library to get the actual duration at the client side, from the time the client sends the first command until he receives the last response from the Amazon server. For each experiment, we ran 50 times and took the average number as the final response time reported in this section. We now describe our experimental evaluation results and compare our schemes with their counterparts.

We first analyze the response time of column-related queries for OMAT, ODS-2D and RowPKG. With these queries, the client can fetch a column from the encrypted database for statistical analysis or a conditional search. Given a column-related query, the total number of bytes to be transmitted and processed by each scheme are shown in Table 4.9. RowPKG's transmission



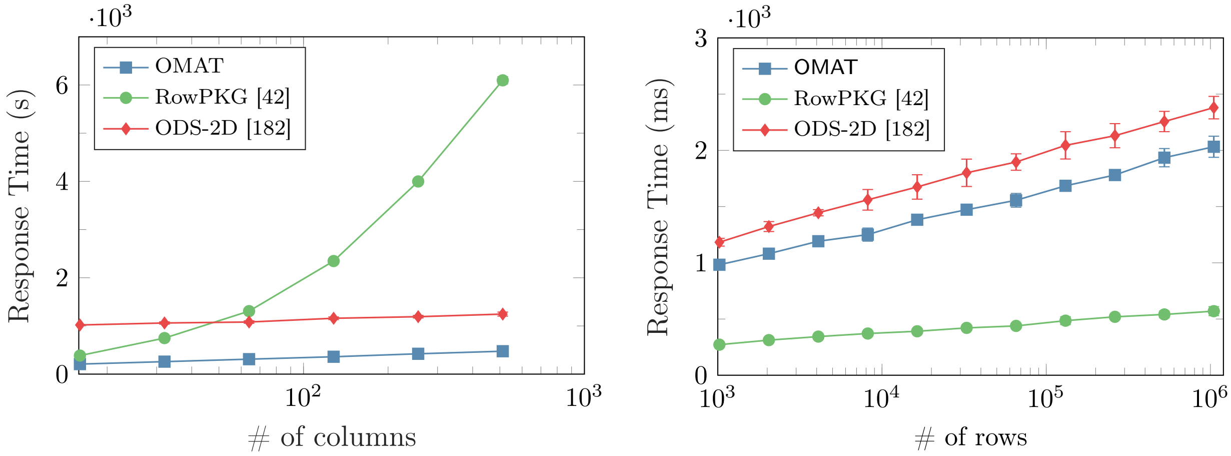
(a) Column-related queries with 2^{15} rows

(b) Row-related queries with 2^5 columns

Figure 4.83: Delay of OMAT and its counterparts with fast network.

cost is the size of all ORAM buckets, where $H \cdot (|b| \cdot N)$ and $(2M - 1)$ denote the bucket size and the total number of buckets, respectively. As for OMAT, its oblivious data structure OCOL allows efficient queries on column dimension with $O(\log(N))$ communication overhead, which outperforms the linear overhead of $O(N)$ of RowPKG. While OMAT and RowPKG can fetch the whole column with one request, it requires $M/4$ synchronous requests for ODS-2D where each request costs $Z \cdot (16 \cdot |b|_1) \cdot \log_2(M \cdot N/16)$ bytes due to 4×4 clustering of the cells.

We measured the performance of OMAT and its counterparts with arbitrary column queries. In this experiment, we set parameters as $|b|= 64$ bytes and $Z = 4$. The number of columns N varies from 2^4 to 2^9, where the number of rows is *fixed* to be $M = 2^{15}$. Figure 4.83a and Figure 4.84a illustrate the performance of the schemes on two different network settings with two different client machines. For a database table with 2^{10} rows and 2^9 columns, OMAT's average query times are $60\,s$ and $475\,s$ compared to RowPKG's $775\,s$ and $6100\,s$, and ODS-2D's $292\,s$ and $1245\,s$ on high- and moderate-speed networks, respectively. This makes OMAT about 13× faster than RowPKG. While OMAT performs 2.6× faster than ODS-2D on the moderate-speed network, it becomes 4.9×



(a) Column-related queries with 2^{15} rows

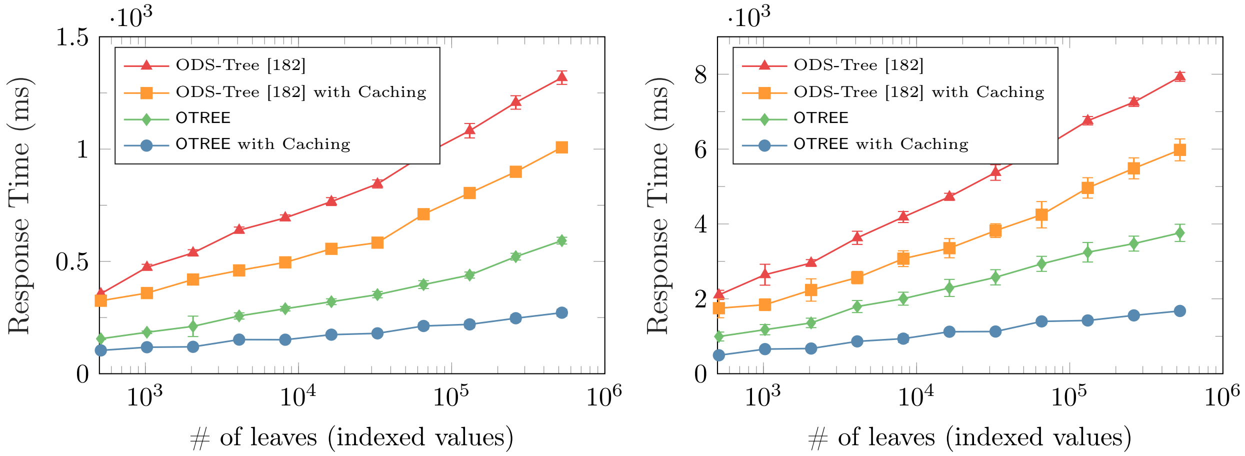
(b) Row-related queries with 2^5 columns

Figure 4.84: Delay of OMAT and its counterparts with moderate network.

on high-speed network since the latency starts to dominate the response time of ODS-2D with $M/4$ requests due to its construction with pointers.

We now analyze the response time of row-related queries for OMAT and its counterparts. Given a row-related query, the total number of bytes to be transmitted and processed by OMAT and its counterparts are summarized in Table 4.9. For OMAT and RowPKG, $(|b|\cdot N)$ and $Z \cdot \log_2(M)$ denote the total row size and the overhead of Path-ORAM, respectively. Due to OMAT's OCOL and OROW structures, OMAT is always a constant factor of $Z = 4$ more costly than RowPKG. Clustering strategy of ODS-2D also introduces more cost and makes ODS-2D 4.2× more costly than RowPKG when $N = 32$.

We measured the performance of OMAT and its counterparts with arbitrary row queries, where the number of rows M varies from 2^{10} to 2^{20}. The block size is $|b|= 128$ bytes and the number of columns is fixed as $N = 32$. By this setting, the total row/record size is $|b|\cdot N = 4096$ KB. Figure 4.83b and Figure 4.84b illustrate the performance of the compared schemes for both network settings. We can see that OMAT performs slower than RowPKG by a constant factor of



(a) Moderate-speed network

(b) High-speed network

Figure 4.85: Delay of OTREE and ODS-Tree.

approximately 2.3× and 3.6× on high and moderate-speed network, respectively. As for ODS-2D, Figure 4.83b explicitly shows the effect of the round-trip delay introduced by network latency on ODS-2D due to $N/4$ synchronous requests. Although ODS-2D has similar cost with OMAT, it performs approximately $220ms$ and $380ms$ slower than OMAT.

We analyze the response time of oblivious traversal on database index that is constructed as a range tree by putting distinct values of a column to the leaf of the tree. Figure 4.82 exemplifies the constructed range tree, and this structure is used along with its linked list to perform conditional queries (*e.g.*, equality, range) on an indexed column, and fetch matching IDs. We compare our proposed OTREE and ODS-Tree with no caching and half-top caching strategies.

Given a database index tree constructed with values of the column, the total number of bytes to be transmitted and processed by OTREE and ODS-Tree without caching are $Z_2 \cdot |b| \cdot (H+1) \cdot (H+2)$ and $2 \cdot Z_1 \cdot |b| \cdot (H+1)^2$, respectively, where H is the height of tree data structure. While ODS traverses the tree with $O(H)$, the additional overhead of Path-ORAM makes the total overhead to be $O(H^2)$. As for OTREE, its level restriction on ORAM storage reduces the transmission overhead by 1.6×. With half-top caching strategy, overheads of both schemes reduce as shown in Table 4.9, however, OTREE's construction benefits more from caching by performing traversal 3.2× less costly than ODS-Tree.

For this experiment, we set the block size $|b|$= 4 KB, the number of blocks inside a bucket for ODS-Tree is $Z_1 = 4$, and the number of blocks inside a bucket for OTREE is $Z_2 = 5$. We benchmarked OTREE and ODS-Tree with arbitrary equality queries when the number of indexed values varies from 2^9 to 2^{19}. The number of indexed values is set to 2^{19} for large database setting. For both network settings, Figure 4.85 demonstrates the effect of half-top caching strategy and how the structure of OTREE gives more leverage in response time. While OTREE without caching performs around 2× faster than its counterpart, caching allows OTREE to perform 3.6× faster than ODS-Tree with caching for both network settings.

We now analyze the client storage overhead of our schemes and their counterparts. The position map of OMAT requires $O((M + N) \cdot \log(M + N))$ storage, while RowPKG requires $O(M \cdot \log(M))$, since only the position map of rows are stored. However, the dominating factor is M, since large databases have more rows than columns. ODS's pointer technique allows it to operate with O(1) storage for position map. Moreover, the worst-case stash size changes with the query type, because stash is also used to store currently fetched data and the worst-case storage costs are summarized in Table 4.9. For row-related queries, the worst-case stash storage is the same for both OMAT and RowPKG but ODS-2D requires more storage due to clustering. For column-related queries, RowPKG requires storing $O(M \cdot N)$ that corresponds to all ORAM buckets. Besides the query performance issues, this also makes RowPKG infeasible for very large databases to perform column-related queries. In addition, ODS-2D also requires $O(\log(M))$ times more client storage compared to OMAT. While RowPKG and ODS-2D have the same server storage size, OMAT requires constant $Z\times$ more storage due to additional dummy blocks.

Since OTREE and ODS do not require the position map to operate, the client storage consists of the stash and additionally cached block according to the caching strategy used. For the worst-case, both schemes have the same client storage with the same caching strategy; however, the stash of OTREE may be more loaded than ODS as shown in Figure 4.81 due to its level restriction. Moreover, server storage of OTREE is 2× less than ODS, since Path-ORAM of ODS requires one more level than OTREE.

Chapter 5: Hardware-Supported Oblivious Storage and Query Platforms[23]

5.1 Introduction

Search and update are two fundamental functions in any data outsourcing and analytic platforms. While ORAM can protect the access pattern and confidentiality while conducting encrypted search and update operations, it incurs high communication/computation, or deployment costs (*i.e.*, by hiring multiple servers). To address this issue, one line of research harnessed secure hardware to support ORAM [8, 62, 127, 152, 159]. Initial studies designed a custom hardware (*e.g.*, FPGA) to enhance the ORAM performance, and therefore, they might not be easily integrated into commercial server systems with a legacy architecture [62, 127, 152]. Thanks to the advent of trusted execution environments recently on commodity hardware such as Intel SGX [46, 100, 101], the deployment of hardware-supported cryptographic primitives has become more feasible [8, 60, 159]. Several oblivious encrypted search platforms have been proposed [66, 172]; however, they incur a high delay when dealing with a large amount of outsourced data since its cost grows *linearly* with the database size. Moreover, state-of-the-art solutions did not fully investigate the update capability, which is an essential feature of data-outsourcing applications.

It is also common that the outsourced database can be shared by multiple users, in the sense that every user may have different permissions to access a specific data item. While ORAM can be extended to support the multi-user setting, doing so brings up new challenges such as network bandwidth overhead, complexity of handling concurrent access, asynchronicity, access control, etc.

[23] This chapter was published in [86, 91]. Permission is included in Appendix A.

Although there exist several ORAM schemes, which support either access control [128, 129] or parallelization [28, 41, 43, 140], they may not fully satisfy all the functionality, performance, and security requirements. Maffei *et al.* in [129] showed that there exists a computation lower bound of $\Omega(N)$ to achieve both the access control and access pattern obliviousness against an active adversary. A more efficient approach is to use a trusted proxy to handle multi-user concurrent accesses and enforce access control [20, 158, 166, 167, 186]. However, fully asynchronous proxy designs [20, 167] were shown insecure against timing attacks [158] exploiting access pattern leakages in processing concurrent requests. To fix this vulnerability, Sahin *et al.* proposed TaoStore [158], the most efficient and secure proxy design to-date, which allows passive ORAM protocols (*e.g.*, Path-ORAM) to be securely executed in parallel over the network. Despite their merits, all proxy designs execute the standard ORAM protocols over the network so that their performance is capped by the limitations of the network bandwidth and latency, and fail to scale beyond that limit. Since all proxy designs are originally proposed to exploit the maximum network throughput for concurrent accesses, they do not focus on access control enforcement or handling an active adversary. Maffei *et al.* in [129] pointed out that proxy-based designs can offer access control enforcement with the incorporation of an access control data structure and authentication at the proxy. A detailed exploration of this enhancement in practical settings is a worthy investigation.

Our objective is to take the ORAM supported by commodity secure hardware to the next levels, in which we develop privacy-preserving and functional platforms using Intel SGX that can support (*i*) practical search/update operations on very large database, (*ii*) concurrent access from multiple users (*iii*) access control enforcement while, at the same time, (*iv*) concealing the access pattern and achieving confidentiality and integrity. We implemented our techniques to demonstrate their efficiency compared with state-of-the-art approaches.

5.2 Related Work

To enable multi-user concurrent access, several Parallel-ORAM schemes have been introduced [28, 41, 43, 140]. However, these constructions feature large block size [41], or add poly-logarithmic communication blowup atop standard ORAM [28, 43, 140]. Maffei *et al.* proposed several access control-supported ORAM schemes (*e.g.*, [128, 129]), and provided a computation lower bound of $\Omega(N)$ for the composition of access control and ORAM against active adversaries.

ObliviStore [168] and its extended multi-server version [166] were among the first to exploit a trusted proxy for concurrent oblivious access in the multi-user setting. In these systems, the proxy schedules concurrent user's requests and then, executes Partition-ORAM [168] with the storage server [20, 158, 166, 167]. Since ObliviStore only parallelizes requests on different blocks, but *sequentializes* same-block requests, it leaks timing information. This allows an adversary to distinguish access patterns from multi-users. To seal this leakage, CURIOUS framework [20] also parallelizes same-block requests by invoking one actual ORAM operation along with several dummy ORAM operations between the proxy and storage server. Although this design enables CURIOUS to be *fully asynchronous* (*i.e.*, one request can be processed immediately without waiting for the previous requests to be finished), Sahin *et al.* [158] showed that it still leaks timing information. Specifically, because dummy ORAM operations do not return the actual block being requested to the proxy, concurrent requests on the same block cannot be answered until the real one is finished. Given that the adversary can observe all network traffic coming from the proxy, and even reschedule the network package delivery from the storage server, it can learn the timing difference in replying same-block requests *vs.* different-block requests of the proxy. Another limitation is that, they rely on Partition-ORAM [168], which incurs costly communication overhead due to the background eviction and high proxy storage overhead (*i.e.*, $\mathcal{O}(\sqrt{N})$).

To seal timing information leakage, Sahin *et al.* [158] indicated that the proxy must reply users' requests *sequentially* according to their arrival order. The authors proposed TaoStore, which implements two modules called *Processor* and *Sequencer* at the proxy, where the former executes a standard ORAM (*i.e.*, Path-ORAM [169]) with the storage server in parallel via multiple threads, while the latter is to reply users' requests sequentially. Once a thread obtains data from the storage server due to ORAM operation, *Processor* will lock the proxy's local memory and synchronize it with the fetched data, and then transfers the desired block to the *Sequencer* for user reply. TaoStore is also more efficient than previous designs because it employs Path-ORAM [169].

Given that the ORAM communication lower bound has been well-established [74, 119, 179], recent studies start to look for the support of secure hardware to make ORAM for client-server applications more practical. The idea of ORAM and secure-hardware composition was first suggested by concurrent studies in 2013 [62, 125, 127, 152]. However, these these techniques harnessed a custom hardware such as FPGA, which may not be widely available on commodity cloud platforms. With the advent of widely available trusted execution environments on commodity hardware (*e.g.*, Intel SGX), the deployment of hardware-supported cryptographic primitives has become more feasible. For instance, ZeroTrace [159] and Obliviate [8] leveraged Intel SGX with ORAM to enable oblivious memory primitives and file access operations, respectively. Intel Intel SGX was also used to design a functional encryption framework in [60]. These works did not focus on multi-client setting and access control enforcement. Another relevant work is [55], which harnesses SGX to enforce access control policy and encryption services. However, it does not provide the access pattern obliviousness.

5.3 Intel-SGX

Intel Software Guard eXtension (SGX) [100, 101], available from Skylake since 2015, is the Intel's commodity implementation of Trusted Execution Environment (TEE) as an extension to the commodity x86 Instruction Set Architecture (ISA) Its motivation is to have a minimal Trusted Computing Base (TCB), which is only the program that runs in the TEE, by putting a hardware trust anchor in the CPU, thus SGX even excluding the operating system from its trust model.

SGX provides an *Enclave*, which isolates its memory from untrusted system components other than CPU and also protects the integrity and confidentiality of its memory. Intel SGX isolates the enclave's execution by providing a private memory called the *enclave page cache* (EPC), which resides in a reserved space in DRAM (the processor reserved memory, PRM) [46]. The EPC is isolated from the other software security domains by SGX's hardware mechanism. Thus, SGX blocks any software attempt to read/write enclave's memory from user-level as well as privileged-level (including operating systems and virtual machine monitor) attackers. Moreover, because SGX encrypts (with integrity check) the data before storing it on to DRAM, EPC stores only encrypted data on it. Therefore, any hardware attempt to read enclave's memory will not leak any meaningful information, and any tampering to enclave's memory will be detected (and then SGX stops the enclave' execution).

SGX supports the remote attestation of the enclave to authenticate whether the configuration of the enclave is correct and to share a secret key for secure communication [46, 102] only after the authentication. When a remote attestation request initiated by a client is delivered to the enclave, the SGX subsystem will run the trusted quoting enclave, which creates a measurement (*i.e.*, hash of configurations, loaded program with a nonce, and a public key material for key exchange) of the enclave and signs it (with the quoting enclave's key). This measurement will be submitted to the

Intel Attestation Service (IAS) to verify the quoting enclave's signature; the result will be signed by IAS and then will be delivered to the client. By verifying IAS's signature on the result, the client ensures that a correct enclave is running on the server. The attestation message also includes public key parameters of the Diffie-Hellman key exchange of the client and the enclave so that a client can securely communicate with the enclave after this process, by encrypting data using the shared secret.

The root of trust of Intel SGX relies on an infrastructure provided by Intel (*e.g.*, the Intel Attestation Service and the quoting enclave). Our current implementation uses Intel SGX for its secure enclave; therefore, its privacy is bound to Intel's discretion. However, we believe that this is just an implementation-specific issue and that the use of alternative open-source secure enclaves such as Sanctum [47] on the RISC-V architecture [183] could relax this restriction, *e.g.*, by distributing trust over the Public Key Infrastructure (PKI), similar to how Transport Layer Security (SSL/TLS) works in practice.

Because an enclave is a part of the user-level process, Intel SGX does not provide any protection on privileged operations such as system calls, *e.g.*, file and network I/O, etc. Because such operations have to be performed by the untrusted OS, the enclave must encrypt data before transferring them to the OS. For example, a network communication between the client and the enclave should apply encryption to their connection, and a file write operation should store only encrypted data. For this purpose, Intel provides cryptographic libraries and tools for secure data migration between the enclave and the OS so the enclave can securely communicate across the security boundary if it is provisioned with a secret key for the encryption; this can be done securely via remote attestation.

5.4 POSUP: Practical Oblivious Search and Update Platform

We design a new hardware-assisted privacy-enhancing platform that we refer to as *Practical Oblivious Search and Update Platform* (POSUP). Our proposed POSUP enables oblivious (single/multi)-keyword search and update operations on very large datasets in a much more efficient and practical manner compared with existing techniques. Our system design is inspired from ZeroTrace [159], where we synergize SGX-supported ORAM with Oblivious Data Structure (ODS) [182] to enable oblivious keyword search and update operations on encrypted data. This synergy (*i*) addresses the network bandwidth and communication hurdles of ORAM-SE composition in the client-server setting; (*ii*) eliminates the cost of processing the entire database inside Intel SGX as in [66, 172]; and more importantly, (*iii*) allows for operation on a large outsourced database without being restricted by Intel SGX memory as in [8]. This composition also enables efficient oblivious keyword update capacity. We further outline our contributions as follows:

1. *New oblivious search and update platform design with SGX:* We construct ODS instantiations for EIDX and EDB by harnessing Intel SGX with Path-ORAM [169] and Circuit-ORAM [179]. POSUP allows for some query types such as single keyword and multi-keyword queries. Moreover, POSUP supports an efficient oblivious update via our optimization tricks that exploit some special characteristics of the underlying oblivious data structures.

2. *Full-fledged implementation and evaluation:* We implemented POSUP and evaluated its performance on commodity hardware with a large dataset (*e.g.*, a full-size Wikipedia English corpus) containing hundreds millions of keyword-file pairs and millions of files. Our implementation is efficient, taking only 1 ms to obliviously access a 3 KB block with SGX hardware. Our experi-

mental results showed that POSUP incurs much lower end-to-end delay than state-of-the-art solutions as follows:

- Compared with the ORAM-SE composition in the conventional client-server model (without a secure hardware), POSUP incurs 100× less network bandwidth overhead and and 1000× fewer network communication round-trips. As a result, the end-to-end delay of POSUP is two orders of magnitude lower than that of this approach for both keyword search and update operations (see §5.4.6 for detailed experiments).
- Compared with processing the entire database in Intel SGX (*e.g.*, [66, 172]), POSUP requires less data to be processed by the enclave (*i.e.* $O(\log N)$ *vs.* $O(N)$, where N is the size of outsourced database). This results in POSUP having 10× lower end-to-end delay than the existing techniques for up to 99.5% of keywords (see §5.4.6). If the number of searched files is small, POSUP can be 100× faster. Moreover, POSUP allows oblivious updating, which does not seem to be fully investigated in state-of-the-art SGX-assisted platforms (*e.g.*, [66]).

3. *Putting hardware-supported ORAM in real effect:* We take the concept of hardware-supported ORAM primitives to the next level, wherein we develop oblivious data structures and routines with optimizations to provide practical search and update functionalities on large databases, which was not investigated by existing hardware-supported memory primitives (*e.g.*, ZeroTrace). Our implementation will be available at: www.github.com/thanghoang/POSUP.

POSUP's objective is to utilize a public cloud server, which is equipped with commodity secure hardware, as secure storage that supports search and dynamic update operations over very large encrypted datasets. To this end, POSUP aims at deploying a practical oblivious encrypted search and update platform to guarantee data confidentiality and no access pattern leakage during

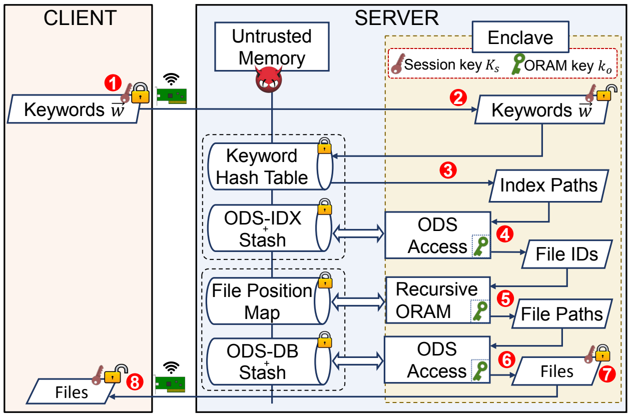


Figure 5.1: An overview of POSUP workflow.

search and update operations, along with a trusted execution. Specifically, to defeat attacks against data confidentiality and access pattern leakages, POSUP creates a commodity hardware-supported ORAM and Oblivious Data Structure (ODS) platform, which enables oblivious searches and updates efficiently, even for very large datasets. To defeat attacks against the server's execution logic, POSUP runs its ORAM and ODS controllers in an enclave protected by Intel SGX.

5.4.1 System and Threat Models

5.4.1.1 System Model

We first describe our system composition and then summarize our POSUP workflow.

Figure 5.1 illustrates the components of POSUP and its composition: a client, an untrusted server, and a trusted enclave on the server. A client (the box on the left side) is a remote entity that generates and manages recursive ORAM and ODS components on the untrusted server via an encryption key k_o. After initializing the system, the client can send a query to update data on the server or to search data and then receive the search results from the server.

The server comprises two parts: an untrusted server and a trusted enclave. (i) The untrusted server provides storage for recursive ORAM and ODS data structures. (ii) The enclave is a trusted part of the server and executes ORAM and ODS controller (while protected by Intel SGX). On behalf of the client, the enclave performs all oblivious search/update operations upon the client's request on the encrypted data structures stored on the untrusted server. To do this, the enclave receives the encryption key k_o from the client via a secure channel at the initialization.

Figure 5.1 outlines the overview of oblivious search and update in POSUP.

The client first performs the remote attestation of the enclave (provided by Intel SGX), which is running on the untrusted server. This attestation step not only verifies that the program running in the enclave is intact but also exchanges cryptographic keys (a session key K_s) to establish a secure (encrypted) channel between the client and the enclave. After establishing the secure channel, the client also sends a key k_o to the enclave, which will be used for ORAM operations.

Upon receiving the key, the enclave on the server will initialize encrypted data structures. Our POSUP is composed of two main encrypted data structures: ODS-IDX, which is an encrypted index that represents keyword-file relations, and ODS-DB, which stores encrypted files. Both data structures are stored in the server's untrusted memory. We employ the ODS techniques proposed in [182] to instantiate ODS-IDX and ODS-DB. This data structure initialization step happens only at the first connection. In other words, to perform search and update operations, the client requires only that a session key (K_s) be exchanged with the enclave.

❶ The client encrypts the *search* (resp. *update*) query with the session key K_s, and sends it to the enclave (through the untrusted server). ❷ Upon receiving the encrypted query, the enclave decrypts it with K_s. ❸ The enclave scans the entire keyword hash table[24]

All these strategies enable us to achieve oblivious keyword search/update operations more efficiently than processing the entire ODS-DB and ODS-IDX in the enclave [66, 172]. Our system is also more efficient than the direct application of existing SGX-ORAM memory primitives [159] because we harness the ODS technique for both ODS-DB and ODS-IDX, which reduces the number of recursive calls when executing an oblivious keyword search/update query.

5.4.1.2 Threat Model

We build POSUP based on the following assumptions as its threat model. The client is fully trusted and transfers only the ORAM encryption key k_o to the enclave after establishing a secure channel with the enclave (Therefore, untrusted parts of server cannot obtain this key). To establish this secure channel, we rely on the remote attestation protocol provided by Intel SGX. Thus, we need a trusted authority (right now, it is Intel) for this remote attestation protocol; however, this does not have to be Intel if we utilize a different kind of secure enclave (*e.g.*, Sanctum [47]). We assume the server is untrusted except for the enclave. Specifically, we do not trust any of the server's logic that includes a virtual machine monitor, operating system and drivers, software that manages

[24] Since keyword universe is arbitrarily large, it is mandatory to maintain a hash table that uniquely matches each keyword to a block ID in the encrypted index for a given dataset (see §5.4.3.1 for more details). to retrieve block IDs and their location (path) in ODS-IDX that correspond to the query. ❹ If the query is to search, the enclave performs ODS accesses on ODS-IDX using the ORAM key k_o to get matching file IDs. ❺ The enclave determines the location of file IDs in ODS-DB by executing recursive ORAM accesses with ORAM key k_o on the file position map structure. ❻ The enclave performs ODS accesses on ODS-DB with k_o to retrieve file(s) associated with the query. ❼ The enclave encrypts retrieved files with K_s and sends them to the client. ❽ The client recovers encrypted files with K_s. The enclave performs the same procedure as search for handling the update query, where it first performs a keyword hash table scan and ODS accesses on ODS-IDX to update blocks in the encrypted index, followed by a recursive ORAM access on the file position map and an ODS access on ODS-DB to update file blocks in the encrypted files.

storage, etc. This is a general assumption for a system that utilizes an enclave because Intel SGX isolates and applies encryption to the enclave's memory space using hardware mechanisms.

Intel SGX does not come without limitations; it suffers from various side-channel attacks in cache access [30, 78, 83, 122], memory access [31, 187], registers [121], etc [106, 184]. Unfortunately, preventing side-channel attacks against Intel SGX entirely is a very challenging task. Instead of making POSUP side-channel free, we aim only to make POSUP secure against several known side-channel attacks on Intel SGX, which are mentioned above. For simplicity, we do not focus on securing POSUP against size and timing information leakage, which can be easily achieved via padding. We refer the reader to §5.4.5 for a more detailed discussion.

5.4.2 POSUP Building Blocks

We use Intel SGX as a trusted execution environment to protect the execution of the ORAM controller on the untrusted server. We run the logic in an SGX enclave, which guarantees the isolation and confidentiality of its execution, to protect the ORAM controller logic from attacks. We utilize the remote attestation protocol of Intel SGX to check the integrity of our logic and securely exchange/provision secret keys for storing ORAM data structures, as well as to protect communication channels between the enclave and the client. We also implement our logic in the enclave using oblivious primitives in the Intel processor such as `CMOV` and `SETE` instructions to prevent potential access pattern leakage.

We implement oblivious assignment (oupt) and oblivious equality comparison (ocmp) functions based on `CMOV` and `SETE` instructions proposed in prior works [142, 149], which do not leak access patterns via control-flow side-channel attacks when POSUP executes the ORAM controller inside the enclave as follows.

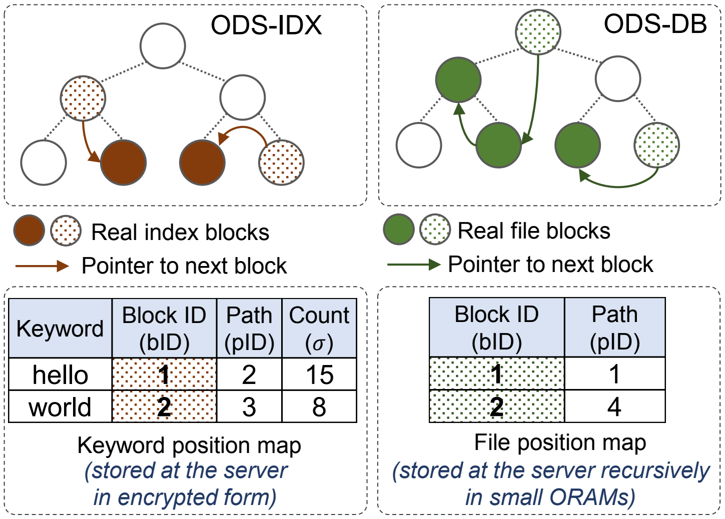
- pred ← ocmp(x, y): It takes as input two values x, y, and outputs pred $= 1$ if $x = y$ or pred $= 0$ otherwise.
- $z \leftarrow$ oupt(pred$, x, y$): It takes as input two values x, y and a boolean pred. It assigns $z \leftarrow y$ if pred $= 1$, and $z \leftarrow x$ otherwise.

We refer interested readers to prior works [142, 149] for a detailed description of these functions. Note that our ocmp function slightly differs from what was originally proposed in [142], where we employ SETE instead of SETG instruction for equality checking.

ZeroTrace [159] proposed OReadPath and OEvict, which are secure versions of ReadPath and Evict tree-based ORAM functions, respectively, both of which are executed by the enclave without leaking side-channel access patterns. We implemented our version of OReadPath and OEvict and refer readers to ZeroTrace [159] for a detailed description. In OReadPath and OEvict, we scan the entire stash and path and then use ocmp and oupt to put real blocks from the path to the stash, or vice versa. This results in Path-ORAM being more computation-expensive than Circuit-ORAM as follows. Since Path-ORAM pushes all real blocks from the path to the stash or vice-versa, its OReadPath and OEvict incur two nested loops, where we must scan the entire stash for each path slot access to hide the access pattern. Circuit-ORAM processes only one targeted block at a time and only incurs two separate loops that scan the entire path once to get target blocks and the entire stash. As a result, Circuit-ORAM is more computation-efficient than Path-ORAM when dealing with a

```
$B \leftarrow$ OGet($S$, id):
$B \leftarrow \perp$
for $i = 1, \ldots, |S|$ do
    $v \leftarrow$ ocmp($S[i]$.id, id)
    $B \leftarrow$ oupt($v, S[i], B$)
return $B$
```

Figure 5.2: OGet function in POSUP.



Keyword	Block ID (bID)	Path (pID)	Count (σ)
hello	1	2	15
world	2	3	8

Keyword position map
(stored at the server in encrypted form)

Block ID (bID)	Path (pID)
1	1
2	4

File position map
(stored at the server recursively in small ORAMs)

Figure 5.3: Illustration of ODS-IDX and ODS-DB packaged into ORAM tree in POSUP.

large dataset and large block size (see §5.4.6). We then implement the OGet function (Figure 5.2), which reads a block ID from the stash S into the enclave without leaking access patterns via ocmp and oupt.

5.4.3 The Proposed POSUP Platform

We first describe the oblivious data structures in POSUP. We then present the oblivious search and update protocol in detail.

5.4.3.1 Oblivious Data Structures

Figure 5.3 presents the overview of ODS-IDX and ODS-DB in POSUP. ODS-IDX and ODS-DB follow the tree ORAM paradigm in [161] because POSUP harnesses Path-ORAM and Circuit-ORAM as oblivious access cryptographic primitives. We create a search index (IDX) from a set of plaintext files (DB) and then package IDX and DB into ODS-IDX, ODS-DB, respectively, as follows.

We construct IDX as an inverted index, in which given DB as the input, we extract unique keywords and associate each keyword w_i with the list of corresponding file IDs id_{ij} that w_i appears in as $w_i := (\mathsf{id}_{i_1} \ldots, \mathsf{id}_{i_n})$. We divide the list of each keyword in IDX into multiple chunks of the same size and package them into separate tree ORAM blocks. We use the pointer trick (*i.e.*, linked list in [182]) to connect these blocks with each other, where the information of successive blocks is stored in their predecessors. Thus, each block is in the form of $B := (\mathsf{id}, \mathsf{DATA}, \mathsf{NextID}, \mathsf{NextPath})$, where id is the block ID; $\mathsf{DATA} := (\langle \mathsf{fid}_1, \sigma_1 \rangle, \ldots, \langle \mathsf{fid}_{n'}, \sigma_{n'} \rangle)$ is the block data, which contains a partial list of file IDs (fid) as well as their state ($\sigma \in \{0, 1\}$) indicating whether they are added or deleted; NextID and $\mathsf{NextPath}$ are the ID and the path of the next block, respectively. Finally, we create $\mathsf{ODS\text{-}IDX}$ by putting all constructed blocks into a tree ORAM structure.

Since keywords are arbitrary and can be of any length, POSUP maintains a hash table data structure (TW) to map each arbitrary keyword to an ORAM block ID (id) as well as its path (pid) in $\mathsf{ODS\text{-}IDX}$. Additionally, POSUP stores a counter (c_i) for each keyword in TW to indicate the *actual* number of $\langle \mathsf{fid}, c \rangle$ pairs that are stored in the head block of the list. So, TW is of the form $\langle \mathsf{key}, \mathsf{value} \rangle$, where key is the hash of the keyword and value contains a triplet (id, pid, c). We denote the access operation to the value component in TW as $(\mathsf{id}, \mathsf{pid}, c) \leftarrow \mathsf{TW}[w_i]$.

In POSUP, we maintain TW under the encrypted form in the storage server. This allows the client to be *stateless* and easily extensible to the multi-client setting (see §5.4.4 for further discussion). Since, to the best of our knowledge, there is no oblivious hash table mechanism, the enclave performs a linear scan on TW for reading/writing component(s) in TW to hide the access pattern. Notice that recursive-ORAM might not be applied on TW because it requires deterministic indexes to operate, while keywords to be accessed are arbitrary.

```
ODS.Setup(DB):
$\mathcal{B} \leftarrow \emptyset; \mathcal{B}' \leftarrow \emptyset$
$\mathcal{W} := (w_1 \dots, w_M) \leftarrow$ Extract all unique keywords in $\mathsf{DB}$
Construct inverted index $\mathsf{IDX} \leftarrow \left(\langle w_i, \overrightarrow{\mathsf{fid}}_i\rangle_{i=1}^{M}\right)$, where $\overrightarrow{\mathsf{fid}}_i := (\langle \mathsf{fid}_{i_1}, 1\rangle \dots, \langle \mathsf{fid}_{i_n}, 1\rangle)$ are file IDs containing $w_i$
for each file $f_i \in \mathcal{F}$ do
    Split $f_i$ into $m_i$ chunks of size $|B|$
    $B_{ij}.\mathsf{DATA} \leftarrow j$-th chunk; $B_{ij}.\mathsf{pid} \xleftarrow{\$} [2^H]\ \forall j \in [m]$
    for $j = 1, \dots, m_i - 1$ do
        $B_{ij}.\mathsf{NextID} \leftarrow B_{ij+1}.\mathsf{id}$
        $B_{ij}.\mathsf{NextPath} \leftarrow B_{ij+1}.\mathsf{pid}$
    $\mathsf{pm}_f[B_{i1}.\mathsf{id}] \leftarrow B_{i1}.\mathsf{pid}$
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{B_{i1}, \dots, B_{im_i}\}$
$\mathsf{ODS\text{-}DB} \leftarrow \mathsf{BuildORAMTree}_{k_o}(\mathcal{B})$
$\mathsf{BuildRecursiveORAM}(\mathsf{pm}_f)$
for each keyword $w_i \in \mathcal{W}$ do
    $\overrightarrow{\mathsf{fid}}_i \leftarrow \mathsf{IDX}[w_i]$
    Split $\overrightarrow{\mathsf{fid}}_i$ to $m'_i$ chunks $(c_{i1}, \dots, c_{im'_i})$ each of size $|B'|$
    $B'_{ij}.\mathsf{DATA} \leftarrow c_{ij}$; $B'_{ij}.\mathsf{pid} \xleftarrow{\$} [2^{H'}]\ \forall j \in [m']$
    for $j = 1, \dots, m'_i - 1$ do
        $B'_{ij}.\mathsf{NextID} \leftarrow B'_{ij+1}.\mathsf{id}$
        $B'_{j}.\mathsf{NextPath} \leftarrow B'_{ij+1}.\mathsf{pid}$
    $\mathcal{B}' \leftarrow \mathcal{B}' \cup \{B'_{i1}, \dots, B'_{im'_i}\}$
$\mathsf{ODS\text{-}IDX} \leftarrow \mathsf{BuildORAMTree}_{k_o}(\mathcal{B}')$
$\mathsf{TW}[w_i] \leftarrow (B_{i1}.\mathsf{id}, B_{i1}.\mathsf{pid}, |c_{i1}|)$ for each $w_i \in \mathcal{W}$
```

Figure 5.4: Setup algorithm to construct oblivious data structures in POSUP.

We apply the same principle as in the encrypted index construction to build ODS-DB from DB. Since each file is organized into a linked list, we set $B.\mathsf{id} = \mathsf{id}$, where B is the first block in the list and id is the ID of the file that B represents. Given that the size of IDX is generally smaller than that of DB, we build ODS-IDX and ODS-DB as two separate oblivious data structures, where the block size of ODS-IDX is smaller than that of ODS-DB.

POSUP maintains the file position map to keep track of the path of the head block in each ODS linked list. Note that we can index file IDs with an integer from 1 to N, where N is the total number of files in DB since POSUP focuses on search and update functionalities on keywords appearing in DB. This allows us to maintain the file position map via a recursive ORAM in the server side. Our design needs to perform only recursive ORAM access to get the path of the starting block, while the path of other blocks in the linked list is obtained from their predecessor due to the ODS technique. This reduces the number of recursive calls on the file position map compared with the direct application of oblivious memory primitives (*e.g.*, [8, 62]) for enabling oblivious query functionality.

We present the detailed algorithm to construct ODS-IDX and ODS-DB in Figure 5.4. We encrypt ODS-IDX and ODS-DB with ORAM key k_o and store both ODS-IDX and ODS-DB on the server's untrusted memory. The stash components required by underlying ORAM schemes are also encrypted with k_o and stored in the server's untrusted memory. They are loaded and decrypted in the enclave when needed.

Since search operations require us to clear stale data that might arise due to previous update operations, we first introduce the oblivious update protocol in POSUP and then describe the oblivious search.

5.4.3.2 POSUP Oblivious Update Protocol

For simplicity, we assume that the file to be updated (f_{id}) is already present at the client side. The update operation incurs f_{id} to be added/deleted/modified from EDB, and some keyword-file pairs to be added/deleted from EIDX. Intuitively, for each keyword (w_i) to be updated in f_{id}, the client requests the enclave to add *id* along with the update state (add/delete) to an empty data slot in the head block of the linked list, which represents the search result of w_i in EDB. If there is no available slot, the enclave picks an empty block[25] in EDB, adds update information into it and then links it with the current head block of the linked list by updating its NextID and NextPath values.

[25] ID of empty blocks can be stored in a separate data structure (*e.g.*, list).

This strategy results in blocks close to the head of the list containing the latest updated file IDs. Figure 5.5 outlines the oblivious update protocol with the following details.

The client updates the file content, and forms a list of keywords ($\mathbf{w}$) to be added or deleted in the updated file (f_{id}) (❶). The client encrypts the update query (q) containing id and $\mathbf{w}$ with K_s and sends it to the enclave. The enclave decrypts q with K_s (❷) and then accesses the entire keyword position map (TW) to retrieve the block ID (id), path (pid), and current counter (c) of updated keywords (❸). For each updated keyword (w_i), the enclave checks whether there is an empty slot in the head block of w_i in ODS-IDX (❸). If this does not hold, the enclave gets the ID and the path of an empty block in ODS-IDX (❺) and updates this block as the new head of the linked list of w_i in TW (❻). Notice that POSUP performs all comparison and update operations in the oblivious manner using ocmp and oupt functions described in §5.4.2 to prevent instructional leakage. Once the target block is determined, the enclave performs an ORAM access on ODS-IDX to add fid and the update state (σ) into it (❼–⓫). Specifically, for each updated keyword (w_i), the enclave first executes OReadPath with path pid_i (❼) to fetch the block (B_i) of ID id_i from ODS-IDX into the stash (S). Second, it reads B_i from S via OGet (❽) and then adds $\langle \mathsf{fid}, \sigma \rangle$ to the block data (B_i.DATA) (❾). If B_i was previously empty, it updates the pointer of B_i to link it with the current block head of w_i (❿). Finally, the enclave performs OEvict to write the updated B_i back to ODS-IDX (⓫).

Given the ID (fid) of the updated file (⓭), the enclave executes recursive ORAM accesses on the file position map to retrieve the location of fid in ODS-DB (⓮). The enclave splits the updated file f_{fid} into several chunks and then executes ODS access on ODS-DB to update the ORAM data blocks in the linked list of f_{fid} with these chunks (⓯–⓱).

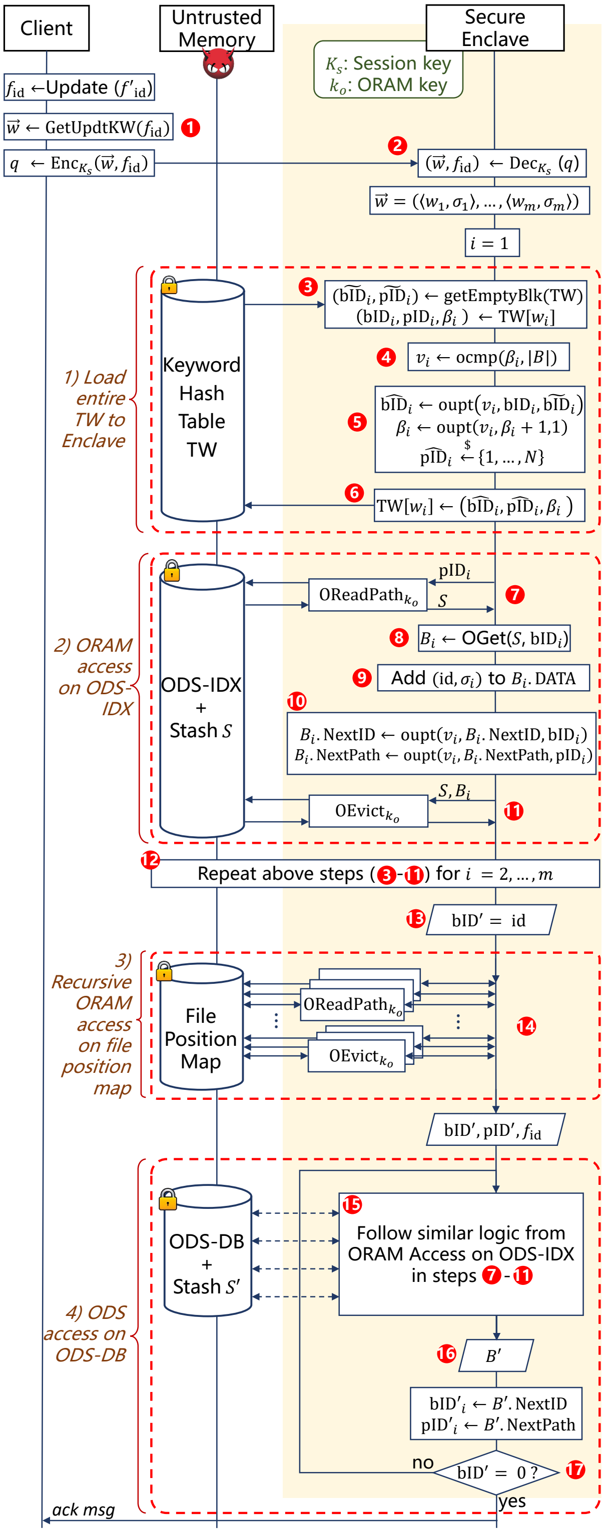

Client
Untrusted Memory
Secure Enclave
K_s: Session key
k_o: ORAM key
$f_{\text{id}} \leftarrow$ Update (f'_{id})
$\vec{w} \leftarrow \text{GetUpdtKW}(f_{\text{id}})$ 1
$q \leftarrow \text{Enc}_{K_s}(\vec{w}, f_{\text{id}})$
2 $(\vec{w}, f_{\text{id}}) \leftarrow \text{Dec}_{K_s}(q)$
$\vec{w} = (\langle w_1, \sigma_1 \rangle, \ldots, \langle w_m, \sigma_m \rangle)$
$i = 1$
1) Load entire TW to Enclave
Keyword Hash Table TW
3 $(\widetilde{\text{bID}}_i, \widetilde{\text{pID}}_i) \leftarrow \text{getEmptyBlk(TW)}$
$(\text{bID}_i, \text{pID}_i, \beta_i) \leftarrow \text{TW}[w_i]$
4 $v_i \leftarrow \text{ocmp}(\beta_i, |B|)$
5 $\widetilde{\text{bID}}_i \leftarrow \text{oupt}(v_i, \text{bID}_i, \widetilde{\text{bID}}_i)$
$\beta_i \leftarrow \text{oupt}(v_i, \beta_i + 1, 1)$
$\widetilde{\text{pID}}_i \xleftarrow{\$} \{1, \ldots, N\}$
6 $\text{TW}[w_i] \leftarrow (\widetilde{\text{bID}}_i, \widetilde{\text{pID}}_i, \beta_i)$
2) ORAM access on ODS-IDX
ODS-IDX + Stash S
OReadPath_{k_o}
pID_i
S
7
8 $B_i \leftarrow \text{OGet}(S, \text{bID}_i)$
9 Add (id, σ_i) to $B_i.\text{DATA}$
10 $B_i.\text{NextID} \leftarrow \text{oupt}(v_i, B_i.\text{NextID}, \text{bID}_i)$
$B_i.\text{NextPath} \leftarrow \text{oupt}(v_i, B_i.\text{NextPath}, \text{pID}_i)$
S, B_i
OEvict_{k_o}
11
12 Repeat above steps (3-11) for $i = 2, \ldots, m$
13 $\text{bID}' = \text{id}$
3) Recursive ORAM access on file position map
File Position Map
OReadPath_{k_o}
OEvict_{k_o}
14
$\text{bID}', \text{pID}', f_{\text{id}}$
4) ODS access on ODS-DB
ODS-DB + Stash S'
15 Follow similar logic from ORAM Access on ODS-IDX in steps 7-11
16 B'
$\text{bID}'_i \leftarrow B'.\text{NextID}$
$\text{pID}'_i \leftarrow B'.\text{NextPath}$
no
$\text{bID}' = 0$?
17
yes
ack msg

Figure 5.5: POSUP oblivious update protocol.

We can see that the deletion in POSUP does not actually delete some real data in ODS-IDX but instead performs addition with the state bit ($\sigma = 0$). This lazy deletion enables the efficient update, which only requires $O(1)$ access on the encrypted index compared with $O(r)$ in the actual deletion due to the search, where r is the number of files in which the updated keyword appears. The price to pay for this efficiency is the cost of increasing the *actual* size of the search index and the search complexity. To mitigate this impact, after k (system parameter) successive updates, POSUP performs a *dummy* search on the most frequently updated keyword to do garbage collection, since the search operation in POSUP will clear stale data appearing in blocks toward the tail of the linked list, as described in the next section.

5.4.3.3 POSUP Oblivious Search Protocol

Figure 5.6 presents the oblivious search protocol in POSUP. First, the client executes the remote attestation protocol with the enclave to establish a secure communication channel with a session key (K_s). The client then encrypts the search query of the form $q = w_1 \star_1 \ldots \star_{m-1} w_m$ with K_s, where $\star_i \in \{\vee, \wedge\}$ (❶) and sends it to the enclave. The enclave decrypts q with K_s to obtain the list of searched keywords (w_i) and performs the following operations.

The enclave scans TW entirely to get the block ID (id_i) and its path (pid_i) in the index (ODS-IDX) of each w_i (❷).

For each $\langle \mathsf{id}_i, \mathsf{pid}_i \rangle$ pair, the enclave performs an ODS access on ODS-IDX (containing multiple ORAM accesses) to retrieve all blocks in the linked list, all of which form the entire list of file IDs (R_i) matching w_i (❸–⓫). According to our update strategy (see §5.4.3.2), blocks toward the head of the oblivious linked list will contain file IDs with the *most up-to-date* update state. Hence, during the block update operation (❺), the enclave will clear stale file IDs in the accessed block if they are

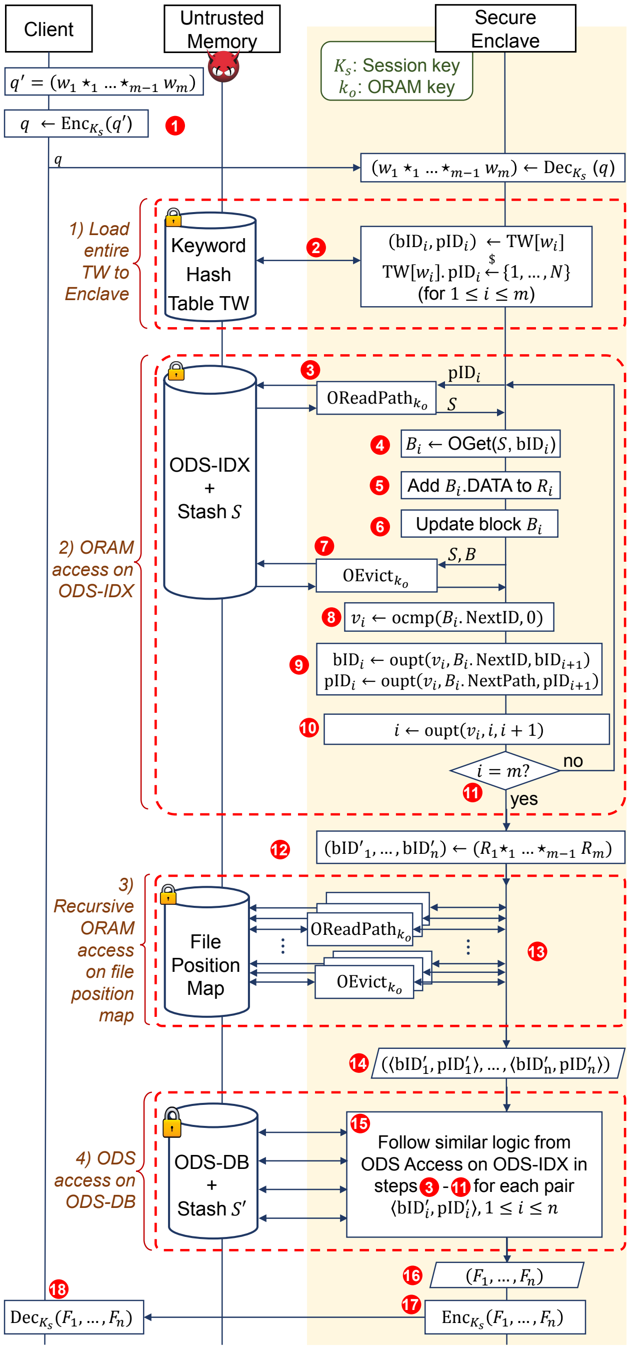

Client
Untrusted Memory
Secure Enclave
K_s: Session key
k_o: ORAM key
$q' = (w_1 \star_1 \ldots \star_{m-1} w_m)$
$q \leftarrow \mathrm{Enc}_{K_s}(q')$
1
q
$(w_1 \star_1 \ldots \star_{m-1} w_m) \leftarrow \mathrm{Dec}_{K_s}(q)$
1) Load entire TW to Enclave
Keyword Hash Table TW
2
$(\mathrm{bID}_i, \mathrm{pID}_i) \leftarrow \mathrm{TW}[w_i]$
$\mathrm{TW}[w_i].\mathrm{pID}_i \overset{\$}{\leftarrow} \{1, \ldots, N\}$
(for $1 \le i \le m$)
2) ORAM access on ODS-IDX
ODS-IDX + Stash S
3
$\mathrm{OReadPath}_{k_o}$
pID_i
S
4
$B_i \leftarrow \mathrm{OGet}(S, \mathrm{bID}_i)$
5
Add B_i.DATA to R_i
6
Update block B_i
7
S, B
OEvict_{k_o}
8
$v_i \leftarrow \mathrm{ocmp}(B_i.\mathrm{NextID}, 0)$
9
$\mathrm{bID}_i \leftarrow \mathrm{oupt}(v_i, B_i.\mathrm{NextID}, \mathrm{bID}_{i+1})$
$\mathrm{pID}_i \leftarrow \mathrm{oupt}(v_i, B_i.\mathrm{NextPath}, \mathrm{pID}_{i+1})$
10
$i \leftarrow \mathrm{oupt}(v_i, i, i+1)$
$i = m?$
no
11
yes
12
$(\mathrm{bID'}_1, \ldots, \mathrm{bID}'_n) \leftarrow (R_1 \star_1 \ldots \star_{m-1} R_m)$
3) Recursive ORAM access on file position map
File Position Map
$\mathrm{OReadPath}_{k_o}$
OEvict_{k_o}
13
14
$(\langle \mathrm{bID}'_1, \mathrm{pID}'_1 \rangle, \ldots, \langle \mathrm{bID}'_n, \mathrm{pID}'_n \rangle)$
4) ODS access on ODS-DB
ODS-DB + Stash S'
15
Follow similar logic from ODS Access on ODS-IDX in steps 3 - 11 for each pair $\langle \mathrm{bID}'_i, \mathrm{pID}'_i \rangle, 1 \le i \le n$
16
$(F_1, \ldots, F_n)$
17
$\mathrm{Enc}_{K_s}(F_1, \ldots, F_n)$
18
$\mathrm{Dec}_{K_s}(F_1, \ldots, F_n)$

Figure 5.6: POSUP search protocol.

already present in R_i, or their most recent update status is "delete." Once the block becomes empty (meaning it does not contain any file IDs left), the enclave will store its ID in the empty list in TWso that it can be re-used later in the update operation. After the block is accessed with ORAM, the enclave determines if it links with another block via ocmp (❽). If this holds, the enclave gets the next block information (❾) and continues the oblivious access as above. Otherwise, it processes the next searched keyword (❿).

Note that we perform all these conditional checks and processing in an oblivious manner via ocmp and oupt functions. This prevents POSUP from leaking the information regarding size of individual searched keywords (⓫) but only the total amount of data that are processed due to the search query.

After the file IDs (stored in R_i) of each keyword w_i are retrieved, the enclave performs union/intersection on R_i according to $\star_i$ to get the final list of file IDs matching q (⓬)[26]. For each block id'_i in the joint list, the enclave performs recursive ORAM access (⓭) on the file position map structure to retrieve the corresponding path pid'_i of id' in ODS-DB (⓮).

The enclave performs a sequence of ODS accesses on ODS-DB with the same logic as ODS-IDX accesses to retrieve the file content of each id'_i (⓯–⓰). Finally, the enclave encrypts all the retrieved files with the session key K_s (⓱), and sends them to the client for decryption (⓲).

5.4.4 Extension to Multi-User Setting

In POSUP, the client is stateless. Thus, it is easy to extend POSUP into the multi-user setting including a data owner (who owns n outsourced files), a storage server, and k users (who want to search/update on n files) as follows. The data owner creates an access control data structure (ACDS) to grant permission (*e.g.*, search/update) for k users on n files. The data owner encrypts and sends ACDS to the enclave, along with the encrypted index (ODS-DB) and encrypted files (ODS-DB)

[26] Since the file ID (fid) is assigned to the ID (id) of the head block in the linked list, we use fid and id interchangeably.

that are constructed, as described in §5.4.3.1, all of which are stored in the server's untrusted memory (*e.g.*, SSD).

Given that a user wants to search for a keyword, he/she will authenticate with the enclave using, for example, the user identifier and password. If authenticated, the enclave performs oblivious access on ODS-IDX (as described in §5.4.3.3) to obtain file IDs matching the query. For each file ID, the enclave accesses ACDS with ORAM to check whether the user has the read permission on the file. If so, the enclave performs oblivious access on ODS-DB to retrieve the file and sends it to the user. The same principle applies to the file update procedure. Roughly speaking, the enclave first authenticates the user and then obliviously accesses ACDS to check whether the user has the update permission on the file. If permitted, the enclave executes the oblivious update protocol as presented in §5.4.3.2.

5.4.5 Security Analysis

We design and build POSUP by using ODS and ORAM, and therefore, its security is inherited from the security of these tools. Specifically, these tools guarantee that POSUP hides all access patterns on ODS-IDX and ODS-DB, given that they have the *same* length as in Definition 5.

In POSUP, we can observe from Figure 5.5 and Figure 5.6 that search and update operations incur the same oblivious access procedures on encrypted data structures. In particular, given a search/update query, the enclave first performs (*i*) access on the entire keyword hash table and then, (*ii*) ODS access(es) on ODS-IDX, followed by (*iii*) recursive ORAM access(es) on the file position map, and finally (*iv*) ODS access(es) on ODS-DB. In the update protocol, add and delete operations also invoke the same oblivious access procedure, where they differ from each other only in terms of the state bit value (σ), which is encrypted in the view of the attacker. Hence, in general, any search/update queries that are of the *same* size and incur the *same* number of (recursive) ORAM and ODS accesses are *indistinguishable*.

Since ORAM and our linked list ODS do not hide the number of oblivious accesses, POSUP might leak the size of the query, which can allow the attacker to distinguish access patterns, thereby learning information about the query. The size information can be inferred in several points when the enclave performs oblivious operations as follows. In the search protocol (Figure 5.6), the size can be learned by an attacker from (*i*) the search query (❶); (*ii*) the number of accesses on the keyword hash table (❷) and the encrypted index (⓫,⓬); (*iii*) the number of recursive ORAM accesses on the file position map (⓭,⓮) and encrypted files (⓰); (*iv*) and the result returned to the client (⓱). Similarly, in the update protocol (Figure 5.5), the size can be leaked from the update query (❶,❷), or the number of accesses on encrypted data structures (⓬,⓱).

To mitigate the impact of size leakage, we can apply padding to all aforementioned positions. For instance, for the query that requires less than n' total ORAM accesses, one can add dummy ORAM accesses on both ODS-IDX and ODS-DB, and dummy recursive ORAM accesses on the file position map to quantize the total number to be n', thus making the query size *indistinguishable* by the attacker. We can further apply padding to obfuscate the actual size of the search/update query as well as the size of (search) results being sent from the enclave to the client at the end of the protocol. However, we notice that such padding strategies are generally application-specific, which fully depends on the characteristics of a particular dataset and user preferences, and also might incur heavy bandwidth and processing overhead as the trade-off. This is because padding will increase the cost of oblivious operations less than n' (suppose the number of required operations is n) to be equal to that of n' actual operations (where $n' > n$). We refer the reader to Ryoan [98] for its parts of data-oblivious communication as well as quantizing processing time to learn more on how the size quantization by padding can thwart such a side channel attack.

Although the enclave of Intel SGX provides security guarantees such as data confidentiality and integrity against direct memory access attacks, it is not free from side-channel attacks. POSUP does not aim to defeat all sorts of side-channel attacks, which seems to be a very difficult task; instead, we try to build POSUP as a best-effort approach to make it secure against known side-channel attacks. The use of recursive ORAM and ODS in POSUP naturally defeats side-channel attacks on data access patterns such as cache side-channel attacks [30, 78, 83]. Employing oblivious data comparison (ocmp) and oblivious data assignment (oupt) in POSUP (see §5.4.2) defeats attacks on the control-flow side channel [31, 122, 187] because these primitives eliminate conditional branches on processing secrets. Therefore, such attacks cannot measure a difference in control-flow for different secrets.

5.4.6 Experimental Evaluation

5.4.6.1 Implementation

We implemented POSUP with C/C++ using the Intel SGX SDK v1.7. Our implementation contains a total of around 4.9K lines of code for trusted and untrusted modules. For cryptographic operations inside the enclave, we leveraged Intel SGX SDK library functions including sgx_aes_ctr_encrypt for encrypting ORAM with AES-CTR mode and sgx_read_rand for pseudorandom number generation. We implemented Path-ORAM and Circuit-ORAM controllers in an enclave to execute ODS access on ODS-DB and ODS-IDX. As mentioned in §5.4.3, our platform stores ORAM stashes encrypted in the untrusted memory (RAM/SSD), and they are loaded into the enclave when needed.

We describe our configuration and evaluation methodology, followed by the main experimental results.

5.4.6.2 Configuration and Methodology

We evaluated the performance of our system on a commodity HP Desktop, which supports Intel SGX and is equipped with Intel E3-1230 v5 @ 3.4 GHz CPU, 16 GB RAM and 512 GB SSD.

Our dataset is the full Wikipedia English corpus `enwiki v.20180120`. To extract text data from the corpus, we used `WikiExtractor` [11] `Python` script and extracted 5,554,594 distinct text-only articles (*i.e.*, files in our term) from `enwiki`. To collect the keywords for the search, we implemented a standard tokenization method to extract unique alphabetical and non-alphabetical keywords from the dataset. The total number of unique keywords in the dataset is 7,075,917 and the total number of keyword-file pairs is 863,782,383. The total size of the database (DB) is 27 GB (on the disk), and the total size of the search index (IDX) is 6.9 GB. Figure 5.7 presents the size distribution of text articles in the `enwiki` dataset.

To assume a general use case of a mobile client and a cloud server, we use Wi-Fi as the communication channel between the client and the server and then mimic the bandwidth and latency of using Amazon EC2 from our lab. The average network latency and transmission throughput are 18 ms and 150 Mbps, respectively.

We compare POSUP with the implementation of existing designs, namely ORAM-SE and EntireSGX. The following represents the configuration of each implementation and how we compare each with POSUP.

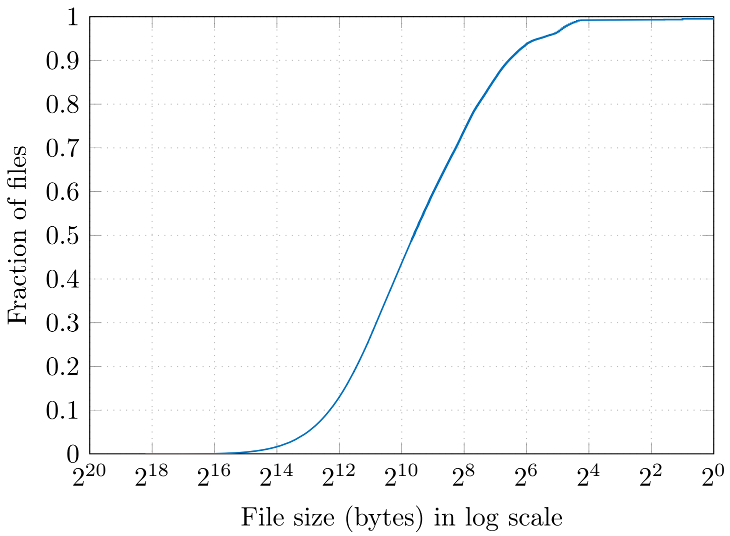


Figure 5.7: File size distribution in `enwiki` dataset in CDF.

- POSUP: We constructed ODS-IDX and ODS-DB with ORAM tree structures with 24 and 23 levels, respectively, to store the entire files ($7{,}075{,}917 \leq 2^{23}$). Because more than 50% of files in our dataset are smaller than 3 KB (shown in Figure 5.7), we selected the block size of ODS-DB to be 3 KB to balance the efficiency and storage overhead. Likewise, we selected the block size of ODS-IDX to be 512 B, because the majority of search keywords appears in less than 512 files (see Figure 5.8c). We represent a file identifier with a 4-byte integer. This results in ODS-IDX being obliviously accessed up to four times for most search cases. We set the stash size of both ORAM schemes as 80 to achieve negligible overflow probability [169, 179].

- ORAM-ODS-SE – Direct ORAM-SE composition in a traditional client-server model (without secure hardware): In this setting, we used the same configuration as in POSUP, where we integrate ODS into the ORAM-SE composition for fair comparison with POSUP. We also set the size of ODS-IDX and ODS-DB to be identical to POSUP. ORAM-ODS-SE differs from POSUP in terms of the ORAM communication channel (over network *vs.* local bus) and the keyword position map location (client *vs.* server). For the ORAM scheme, we employed Path-ORAM for

ORAM-ODS-SE because it requires less access to ORAM, so it is more efficient than Circuit-ORAM in the conventional client-server network setting. For the evaluation, we measured all delays when a client is accessing ODS and recursive ORAM on ODS-IDX and ODS-DB stored on the Amazon EC2 with the above network throughput and latency (18 ms and 150 Mbps). Note that our analysis is *conservative*, because we tend not to take into account the impact of side factors (*e.g.*, disk I/O).

- EntireSGX – Processing the entire outsourced data in Intel SGX: We measured the search delay by decrypting the entire EIDX and EDB inside the enclave. We used the maximum heap size (*i.e.*, 95 MB) allowed to the enclave to subsequently decrypt EIDX and EDB to maximize its performance. In other words, EIDX and EDB are loaded and processed (decrypt/encrypt) in 95 MB chunks sequentially inside the enclave. For the update cost, we measured the delay of decryption and re-encryption of the entire EIDX and EDB inside the enclave. Notice that in POSUP, we selected the size of ODS-IDX and ODS-DB that have sufficient empty spaces for later addition of the same amount of dataset size in the setup phase (*i.e.*, 27 GB file with 6.9 GB index). Hence, we double the size of EIDX and EDB in EntireSGX to assume it can also support addition, similar to POSUP, for fair comparison between two techniques.

5.4.6.3 Micro Benchmark

We first conducted a micro benchmark of POSUP to investigate the delay of performing a single recursive ORAM and ODS access. Three factors cause delay in each operation: (i) the time to read/write ORAM data from the hard disk to the memory and vice versa (*i.e.*, I/O access); (ii) the time for an enclave to secure ORAM operations, such as applying encryption and decryption on the data (*i.e.*, encryption overhead); (iii) the amount of data to be processed in each ODS operation on

Table 5.1: Micro-benchmark of POSUP.

Operation	Execution Time (μs)	
	Path-ORAM	Circuit-ORAM
ODS access on ODS-IDX		
I/O Access	134	144
Enclave Process	2,362	686
Total	2,496	830
ODS access on ODS-DB		
I/O Access	156	285
Enclave Process	3,909	746
Total	4,065	1,031
Recursive ORAM on file position map		
I/O Access	34	41
Enclave Process	13,246	4,631
Total	13,280	4,672

ODS-IDX and ODS-DB, expressed as

$$D_{\mathsf{ODS}} = H \cdot |B| \cdot Z \cdot k, \tag{5.1}$$

where H and $|B|$ are the height and block size of ODS-IDX (or ODS-DB), respectively; and $(Z, k) = (4, 2)$ are the bucket size and the number of read/write operations in Path-ORAM, respectively. When Circuit-ORAM is used, $(Z, k) = (2, 5)$. The amount of data (in bytes) to be processed for each recursive ORAM on the file position map pm_f is:

$$D_{\mathsf{pm}_f} = \sum_{i=1}^{l} \Big(|B| \cdot Z \cdot k \cdot \Big(\log_2 \Big(\frac{N}{R^i} \Big) + 1 \Big) \Big), \tag{5.2}$$

where N is the total number of files, $R = |B|/4$ is the compression ratio (assume that a path ID is represented by 4 bytes) and $l = \lfloor \log_R N \rfloor$.

Table 5.1 presents the execution time of each ODS access on ODS-IDX and ODS-DB and recursive ORAM on pm_f in our current configuration. Note that the performance changes for the different parameter configurations (*e.g.*, for a different H, $|B|$, Z, or k) according to Equation 5.1 and Equation 5.2.

The I/O access in POSUP is efficient because we implemented the caching technique proposed in [127], where we cache the first K levels of the ORAM tree structures on the RAM. In this experiment, we used 4 GB of memory to cache 2/3 levels of both ODS-IDX and ODS-DB, which significantly reduced I/O delay from 520-767μs to $\leq$285μs. Compared to Path-ORAM, Circuit-ORAM incurs 1.25× more I/O access, and therefore, its I/O latency is slightly higher than that of Path-ORAM. Because the recursive ORAM structure of the file position map is small (*i.e.*, $\approx$0.2 GB), we store the entire map directly on the RAM. This results in its I/O access delay being negligible (*i.e.*, $\leq$41 μs).

Processing data in the enclave has more effect on the access delay than I/O access because it handles encryption and decryption when reading data from ORAM. Another point that we observed from Table 5.1 is that the cost of executing the Path-ORAM controller in the enclave is much higher than that of Circuit-ORAM because its read/eviction is more aggressive. Specifically, when using Path-ORAM controller, the enclave must perform $O(\log N) \cdot |\mathbf{S}|$ number of encryptions/decryptions, where $|\mathbf{S}|= 80$ is the stash size. In contrast, using the Circuit-ORAM controller requires $O(\log N)+|\mathbf{S}|$ number of encryptions/decryptions. Therefore, our benchmarked result has shown that integrating Path-ORAM with secure hardware is much less efficient than Circuit-ORAM due to the multiplied factor $|\mathbf{S}|$, which is 80. The processing delay of recursive ORAM is high because this requires the enclave to perform additional ORAM encryptions and decryptions on $O(\log N)$ recursion levels.

Table 5.1 also illustrates that it takes 830 μs to obliviously access a 512 B block in ODS-IDX with Circuit-ORAM. That is, the latencies of performing single-keyword searches on ODS-IDX in many cases are likely similar to each other, and they are mostly dominated by the number of files to be returned. We now illustrate the formula for calculating the number of ODS and recursive ORAM accesses incurred in each search and update query. Given a search query q with n keywords w_i, let m_i and m' be the number of files matching w_i and the final q, respectively. The search q on

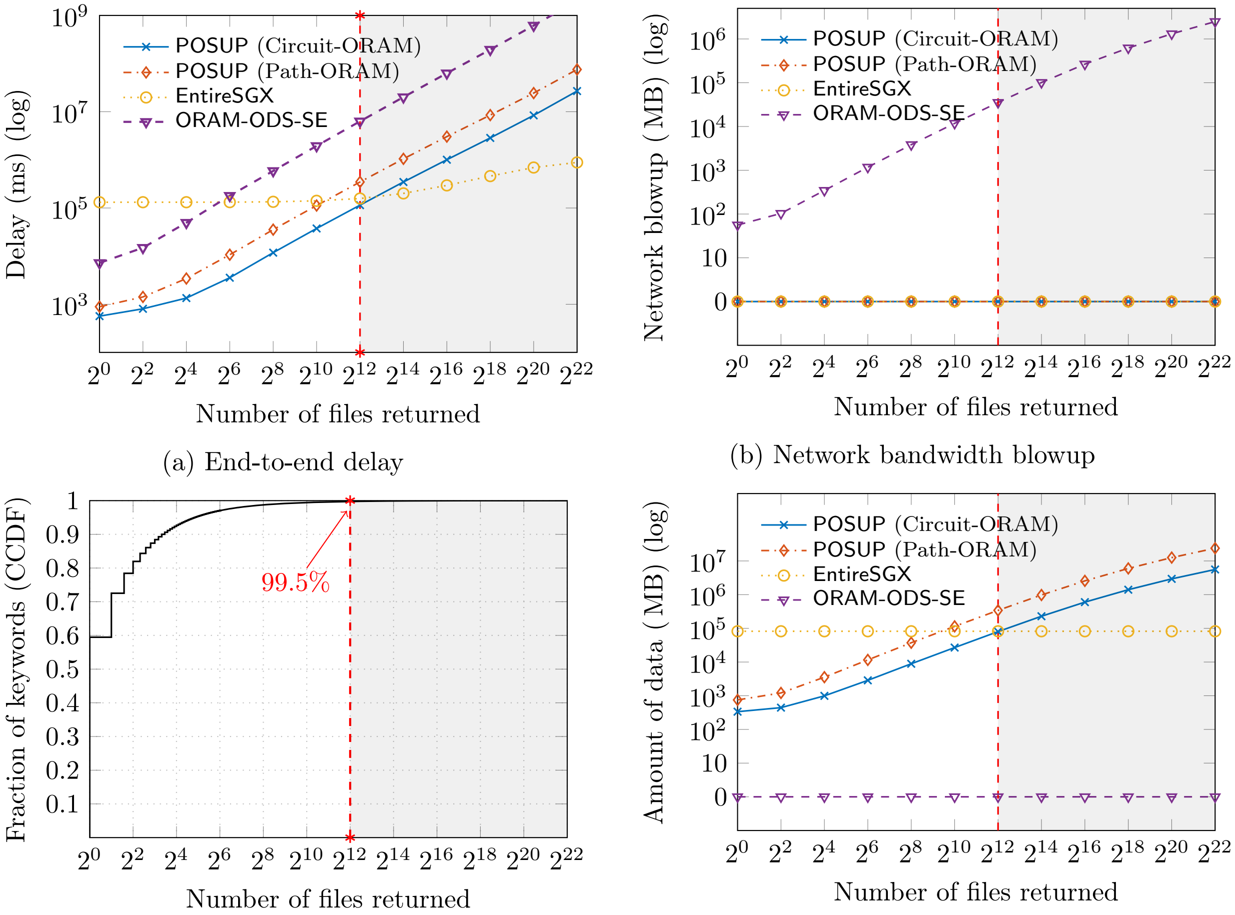
POSUP incurs $\sum_{i=1}^{n} \lceil \frac{m_i}{|B|} \rceil$ accesses on ODS-IDX plus m' recursive ORAM accesses on pm_f and plus $\sum_{i=1}^{m'} \lceil \frac{|f_i|}{|B'|} \rceil$ accesses on ODS-DB, where B, B' are block sizes of ODS-IDX and ODS-DB, respectively, and $|f_i|$ is the size of file f_i in m' files. Given an updated file f with m updated keywords in it, the cost is m accesses on ODS-IDX plus one recursive ORAM access on pm_f plus $\lceil \frac{|f|}{|B|} \rceil$ accesses on ODS-DB.

Because the delay in I/O access and encryption in an enclave is stable (*i.e.*, does not change between accesses), our measurement of the actual search and update delay in POSUP respected the above formulas and the micro benchmark in Table 5.1. Moreover, as explained in §5.4.3.1, each search/update operation in POSUP additionally incurs one-time decryption and re-encryption of the entire keyword hash table (TW), which costs 210 ms for 188 MB-sized TW constructed from the `enwiki` dataset.

In the following, we present actual benchmarked delay for search and update operations to showcase the efficiency of our system compared with other techniques.

5.4.6.4 Search Delay

Figure 5.8a presents the end-to-end delay of processing a keyword search query in POSUP, compared with ORAM-ODS-SE and EntireSGX techniques. POSUP (the blue line) is hundreds of times faster than ORAM-ODS-SE (the purple line) for any search query being performed. This is mainly because POSUP performs ORAM-controlling operations in an enclave, so it does not incur significant network communication overhead like ORAM-ODS-SE, as shown in Figure 5.8b; instead, the enclave reads a large amount of data from the memory, which is faster and cheaper than accessing over the network. ORAM-ODS-SE incurs overhead not only in bandwidth (*i.e.*, 100× more than POSUP) but also in generating a large number of network round-trips (*i.e.*, 1000× more than



(a) End-to-end delay

(b) Network bandwidth blowup

(c) Keyword distribution in `enwiki` dataset

(d) Amount of data being processed by SGX

Figure 5.8: Detailed search delay of POSUP and its counterparts.

POSUP) due to multiple rounds incurred in the recursive ORAM and ODS operations. This is the main bottleneck of ORAM-ODS-SE, given that the network latency is hard to improve in practice.

When compared with EntireSGX, POSUP is one to two orders of magnitude faster than EntireSGX for more than 99.5% of keywords that can be searched. In Figure 5.8a, when searching keywords that returns $\leq 2^{12}$ files, POSUP is more efficient than EntireSGX. Figure 5.8c presents the (accumulative) keyword distribution on `enwiki`. The Zipf's law distribution [141] shown in Figure 5.8c indicates that the cases returning $\leq 2^{12}$ files are the majority (99.5%), and this indicates that POSUP is more efficient than EntireSGX for a large fraction of keywords in practice. This is because the enclave in POSUP only works with a small amount of data per ORAM access, while EntireSGX works with the entire index and dataset as its working set, as presented in Figure 5.8d. For a small fraction of keywords ($< 0.5\%$), the end-to-end delay of POSUP is slower than that of EntireSGX.This is because a large number of ORAM and ODS accesses on ODS-IDX and ODS-DB require data processing in enclave more than processing the entire dataset. The cost of executing Path-ORAM and Circuit-ORAM in POSUP is $C \cdot r \cdot 8 \log_2(N)$ and $C \cdot r \cdot 10 \log_2 N$, respectively, where r is the number of file blocks matched with the search query, $N = 2^{23}$ is the total number of file blocks in ODS-DB, and C is a constant factor. This formula implies that if $r \geq \frac{N}{C \cdot k \cdot \log_2 N}$, where $k \in \{8, 10\}$, then processing the entire database in the enclave is better than performing ORAM. Our benchmark result in Figure 5.8a respects this formula. Theoretically, POSUP should incur more memory accesses than accessing the entire memory when it processes more than 2^{15} files. The graph shows that overhead start to become significant when 2^{13} files are returned, we can assume the constant C as 4, and then POSUP processes more than $\frac{N}{4 \cdot k \cdot \log_2 N}$ file blocks in the enclave.

Padding so that a search query has the same size as another query will result in a total delay of two queries becoming similar. For example, padding on one-file-involved queries to make their size the same to four-file-involved queries incurs 46% extra delay (239ms) compared with the non-padding case.

5.4.6.5 Update Delay

We selected the file with the largest size (*i.e.*, 290 KB) in `enwiki` and performed the update benchmark on that file for a different number of unique keywords that can be updated (add/delete) in it. Figure 5.9 presents the end-to-end update delay of POSUP and its counterparts. POSUP is one order of magnitude faster (40×) than ORAM-ODS-SE because it does not increase bandwidth

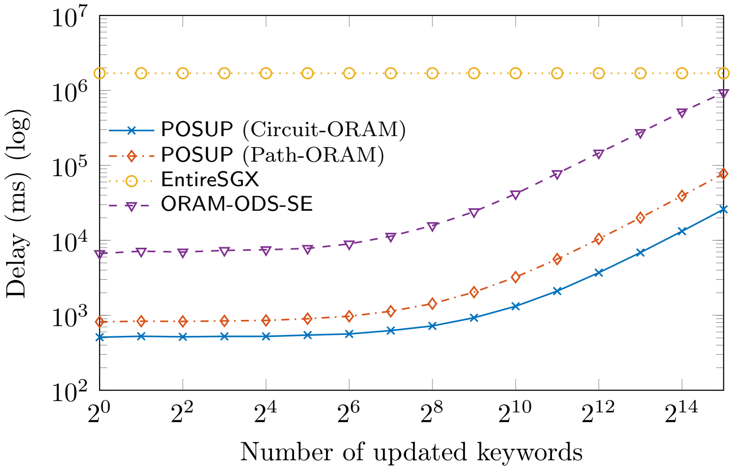


Figure 5.9: End-to-end update delay of POSUP with a 290 KB file size.

and round-trip overhead, as analyzed above. POSUP with Circuit-ORAM produces the highest throughput so that it achieves the lowest update delay among its counterparts. POSUP is up to $3,300\times$ faster than EntireSGX due to the inevitable overhead in I/O writing required by the design of EntireSGX. An update in EntireSGX requires re-encrypting the entire data and write them back to the disk. Therefore, the update delay of POSUP is *three orders of magnitude* faster than EntireSGX.

5.4.6.6 Storage Overhead

The server storage of EntireSGX is more efficient than POSUP and ORAM-ODS-SE. This is because, in POSUP and ORAM-ODS-SE, IDX and EDB are arranged as tree-ORAM structures, which incur a constant (*e.g.*, 1.5×-2×) size blowup. Specifically, the total server storage of POSUP is $|\mathsf{TW}|+|\mathsf{ODS\text{-}IDX}|+|\mathsf{ODS\text{-}DB}|+|\mathsf{pm}_f| = 0.19+34+97+0.19 \approx 131$ GB if POSUP uses Circuit-ORAM. The corresponding overhead is $0.19+68+194+0.38 \approx 262$ GB if POSUP uses Path-ORAM. Note that some capacity of the ORAM structure is reserved to enable oblivious update (*e.g.*, addition/deletion). Therefore, our server storage overhead presented above can allow the further addition of 3× more IDX and DB to what being used in this experiment.

5.5 MOSE: Multi-User Oblivious Storage Platform

We design a multi-user oblivious storage system called MOSE, which employs secure enclave to drastically reduce network overhead, support security functionalities against active adversaries, and exploit parallelism in the server at once. In particular, we first develop the trusted logics (similar to the trusted proxy) on an untrusted server (*i.e.*, not on the network) by adopting a commodity secure enclave (SGX) as a secure isolation mechanism. This design is inspired by recent systems and frameworks assisted by SGX (*e.g.*, [8, 55, 60, 66, 91, 142, 159]), which employ a commodity secure hardware to enhance the efficiency of the underlying cryptographic operations. We are motivated by these approaches to overcome network bandwidth limitations of the previous proxy-assisted ORAM systems.

MOSE offers the following desirable properties.

- *Minimal network bandwidth overhead:* MOSE removes network-related problems in existing proxy-assisted oblivious storage systems by putting the trusted logic on an enclave. In this setting, the enclave communicates with the storage server via a local memory bus, which is several orders of magnitude faster than the network communication in terms of both bandwidth and latency. This design does not incur network bandwidth overhead because MOSE will transfer only the encrypted, bulk data over the network.

- *Access control enforcement against active adversary:* MOSE, running in a secure enclave, securely enforces access control policies and hides the access patterns against an active adversary corrupting the users and/or all components of the server except for the CPU. To this end, we define a security notion for enclave-assisted multi-user ORAM with access control in the presence of an active adversary. We show that MOSE achieves the security according to this model (see §5.5.4).

- *Scalable to multi-user concurrent access:* MOSE achieves a scalable performance by utilizing a parallel optimization, which achieves maximum concurrency at the server while preserving the ORAM security. In particular, MOSE is scalable proportional to the available system resources, *e.g.*, the number of CPU cores in the server. These properties allow MOSE to achieve a high throughput in serving a large number of concurrent requests from multiple users.

- *Implementation and experimental evaluations:* We implemented MOSE and evaluated its performance on a commodity desktop that supports SGX, and the experimental result indicated that our system is efficient and practical (see §5.5.5). For example, with 96 GB database, MOSE can process 374 concurrent requests while achieving less than one second delay to all requests, and it can process 788 concurrent requests while achieving, on average, one second delay per request. MOSE is two orders of magnitudes faster than the state-of-the-art [158] and thus more suitable in serving multi-users requests. Specifically, MOSE can support more than 300 concurrent users with a reasonable delay for accessing 96 GB database, compared with 10 users on 13 GB database in TaoStore with 1 Gbps network bandwidth. To achieve these, MOSE incurs 10 GB of additional memory usage and doubles the storage overhead.

Based on these results, we believe that MOSE fills an important practical gap between security and performance, and facilitates secure data outsourcing in practice.

5.5.1 High-Level Architecture

At a high level, MOSE is an outsourced oblivious storage in an untrusted cloud server with a trusted proxy that handles user's requests. MOSE runs its trusted logic such as user authentication, en/decryption, and the ORAM controller logic in the enclave Unlike previous approaches that put the trusted proxy in a physically separated server on the network, MOSE utilizes hardware-based isolation provided by SGX and can guarantee a comparable grade of security to such approaches. To make an access request to MOSE, a user must be authenticated by the enclave logic and also pass the access control checking. After that, the enclave will perform oblivious access to the encrypted database stored in untrusted storage. The trusted logic will decrypt the content, and finally, the result will be returned to the user. Note that the communication between the user and the enclave is encrypted by the secret key shared by the remote attestation protocol provided by SGX.

On the other hand, the storage is untrusted, thereby the database and the access control structure must be encrypted. This proxy construction ensures *three* security requirements for data outsourcing. First, the ORAM access from trusted proxy can guarantee the access pattern obliviousness if our secure enclave does not leak any critical data via side-channel analysis. Second, to guarantee security, MOSE blocks control-flow side channel via secure (non-branch) comparison and assignment logic and blocks cache side-channel by accessing the entire data for critical ORAM components such as requested paths and the stash. Third, for access control and resiliency, MOSE implements access control logic in the trusted enclave and uses ORAM to obliviously check the permission, so even when a malicious user colludes with the server, they cannot infer any information about the honest user(s).

Regarding the performance, MOSE achieves the two following design requirements. First, by having a trusted proxy in a secure enclave, MOSE eliminates efficiency issues related to network bandwidth/latency applied between the proxy and the storage server. Instead of communicating via a network link, the trusted proxy of MOSE uses memory bus, which is two orders of magnitude faster (273 Gbps *vs.* 1 Gbps) in bandwidth and much faster (350 ns *vs.* ≈ 50 ms) in latency. This design unleashes the bottleneck in serving concurrent requests as in TaoStore [158], which can only support *ten* users for achieving their optimal performance.

Second, MOSE offers scalable performance for supporting multi-user access by speeding up a single ORAM access via parallelization while processing the entire user access requests sequentially. Unlike previous works aiming at parallelizing multiple ORAM access in a concurrent manner, this construction makes MOSE free from the asynchronicity attack. In particular, MOSE applies parallelization to a single ORAM access request regarding encryption and decryption computations, and I/O access. MOSE splits each block in ORAM structures into m chunks, where m is the number of parallel threads. When processing an access request, each thread processes the reading of each chunk from the storage as well as cryptographic computation in parallel. Thanks to this parallel construction, MOSE's performance increases in proportion to the number of parallel threads, and therefore, it is scalable.

5.5.2 System and Threat Models

5.5.2.1 System Model

Our system is comprised of a data owner, k users, and an untrusted storage server $\mathcal{S}$ equipped with SGX. The data owner owns a database (DB) containing N blocks and grants permissions such as read (R), read&write (RW) for k users to access N blocks via an access control data structure (ACDS). Only the data owner can update ACDS entries. The data owner encrypts DB and ACDS forming EDB and EAC, respectively, both of which are stored in the untrusted memory region of $\mathcal{S}$ such as solid-state drive (SSD). To access an entry in DB/ACDS, the users/data owner interacts with an enclave created by SGX, which acts as a trusted proxy to execute ORAM operations with $\mathcal{S}$.

For the sake of simplicity, we say accessing $\mathcal{S}$ to imply accessing the untrusted memory region in $\mathcal{S}$. We consider the enclave as the ORAM client (OClient) in our models because it is the only entity that executes the ORAM protocol with $\mathcal{S}$.

Inspired by [129, 158], we present the definition of multi-user ORAM with trusted proxy in Definition 17. Our model differs from the ones in [129] and [158] in the sense that the former does not use proxy while the latter does not consider the access control enforcement. We denote the execution of protocol A by $\mathsf{OClient}$ with the server $\mathcal{S}$ as $(\overrightarrow{o}_C; \overrightarrow{o}_S) \leftarrow A(\overrightarrow{i}_C; \overrightarrow{i}_S)$, where the input/output vectors of two parties are separated by a semicolon (;).

Definition 17 (Enclave-assisted multi-user ORAM with access control). A multi-user ORAM with access control is comprised of the following (interactive) PPT algorithms:

- $(k_o, \mathsf{EDB}, \mathsf{EAC}) \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{DB}, k, N)$: It takes as input a security parameter λ, and a database DB containing N data blocks. It initializes a data structure ACDS managing the access policy of k users for N blocks. It returns EDB and EAC as the encrypted form of DB and ACDS, respectively.

- $(p; \perp) \leftarrow \mathsf{ReadAC}(\mathsf{uid}, \mathsf{id}; \mathsf{EAC})$: It takes a user ID uid and a block ID id from $\mathsf{OClient}$ and EAC from $\mathcal{S}$ as input. It reads the permission on EAC for entries $\mathsf{id}$and uidas $p \leftarrow \mathsf{EAC}(\mathsf{uid}, \mathsf{id})$. It outputs a permission $p \in \{\mathsf{R}, \mathsf{RW}, \perp\}$ to $\mathsf{OClient}$ and $\perp$ to $\mathcal{S}$.

- $(p; \mathsf{EAC}') \leftarrow \mathsf{WriteAC}(\mathsf{uid}, \mathsf{id}, p; \mathsf{EAC})$: It takes $(\mathsf{uid}, \mathsf{id}, p)$ from $\mathsf{OClient}$, where $p \in \{\mathsf{R}, \mathsf{W}, \mathsf{RW}, \perp\}$ and EAC from $\mathcal{S}$ as input. It updates as $\mathsf{EAC}(\mathsf{uid}, \mathsf{id}) \leftarrow p$. It outputs p to $\mathsf{OClient}$, and the updated EAC' to $\mathcal{S}$.

- $(\mathsf{data}; \perp) \leftarrow \mathsf{ReadDB}(\mathsf{uid}, \mathsf{id}; \mathsf{EAC}, \mathsf{EDB})$: It takes $(\mathsf{uid}, \mathsf{id})$ from $\mathsf{OClient}$ and $(\mathsf{EAC}, \mathsf{EDB})$ from $\mathcal{S}$ as input. It executes $(p; \perp) \leftarrow \mathsf{ReadAC}(\mathsf{uid}, \mathsf{id}; \mathsf{EAC})$. If $p \notin \{\mathsf{R}, \mathsf{RW}\}$, it gets $\mathsf{data} \leftarrow \mathsf{EDB}[i]$, where i is a dummy block ID. Otherwise, it gets $\mathsf{data} \leftarrow \mathsf{EDB}[\mathsf{id}]$. Finally, it returns data to $\mathsf{OClient}$ and $\perp$ to $\mathcal{S}$.

- $(\mathsf{data}; \mathsf{EDB}') \leftarrow \mathsf{WriteDB}(\mathsf{uid}, \mathsf{id}, \mathsf{data}^*; \mathsf{EAC}, \mathsf{EDB})$: It takes $(\mathsf{uid}, \mathsf{id}, \mathsf{data}^*)$ from $\mathsf{OClient}$, where data^* is the new data to be written, and $(\mathsf{EDB}, \mathsf{EAC})$ from $\mathcal{S}$ as input. It executes $(p, \perp) \leftarrow$

$\mathsf{ReadAC}(\mathsf{uid}, \mathsf{id}; \mathsf{EAC})$. If $p \neq \mathsf{RW}$, it gets $\mathsf{data} \leftarrow \mathsf{EDB}[i]$, where i is an dummy block ID. Otherwise, it gets $\mathsf{data} \leftarrow \mathsf{EDB}[\mathsf{id}]$, and updates $\mathsf{EDB}[\mathsf{id}] \leftarrow \mathsf{data}^*$. It returns data to $\mathsf{OClient}$, and EDB' to $\mathcal{S}$ indicating EDB is (possibly) updated.

- $\mathsf{data}' \leftarrow \mathsf{Response}(K_s, \mathsf{data})$: It encrypts data with key K_s.

5.5.2.2 Threat Model

We build MOSE based on following assumptions as its threat model.

We trust the data owner, OClient and the ORAM controller logic, both of which run by the enclave. We trust the remote attestation protocol by SGX, on which we rely on for verifying the integrity of the enclave and establishing a secure communication channel between the user and the enclave. We also trust the hardware key generated by the enclave that seals the ORAM key.

We assume that the server $\mathcal{S}$ is completely untrusted except the enclave. We assume attackers on the server can freely monitor access patterns of the users and the enclave such as network access, storage and memory access; however, they cannot compromise data confidentiality for such accesses. We do not trust any of server's logic that includes virtual machine monitor, operating system and drivers, software that manages storage, etc. This is a general assumption for a system that utilizes secure enclaves because SGX isolates and applies encryption to the enclave's memory space via hardware mechanisms. Additionally, data users and the server can also be *active* adversaries, meaning that they may attempt to inject/tamper their input or even collude with each other to break the security of other honest users.

5.5.3 Design of MOSE

5.5.3.1 System Initialization

Figure 5.10 presents the initialization workflow of MOSE. Given a database DB containing (upto) N block entries being shared among (upto) k users ($\mathsf{uid}_1, \ldots, \mathsf{uid}_k$), the data owner executes the Gen algorithm to construct mandatory data structures outsourced to the cloud (Step ❶). The algorithms first generates an ORAM symmetric key k_o and initializes an access control matrix ACDS (see §5.5.3.3 for our argument on why matrix structure is preferred) for each user uid_i with each database entry id_j. For simplicity, we consider basic permission attributes including read and write. DB and ACDS are then packaged and encrypted into two separate recursive Circuit-ORAM structures called EDB and EAC, respectively. Figure 5.11 presents the Gen algorithm, which executes Circuit-ORAM setup algorithm to construct recursive Circuit-ORAM structures for the database and the access control structure. After the algorithm is executed, the data owner sends EAC and EDB as well as their encrypted position map (all in the form of Circuit-ORAM trees) to the server, all of which are stored in the untrusted memory region (*e.g.*, SSD, at Step ❷). The data owner performs a remote attestation of an enclave (provided by SGX) running on the server to ensure if the enclave is intact and to exchange cryptographic key for establishing an encrypted communication channel between the data owner and the enclave. To this end, the data owner sends k_o to the enclave via the established channel (Step ❸). To ensure security, the enclave always keeps k_o in the trusted memory, and stores it to the disk only after encrypting it with the enclave's hardware key (*i.e.*, data sealing feature provided by SGX).

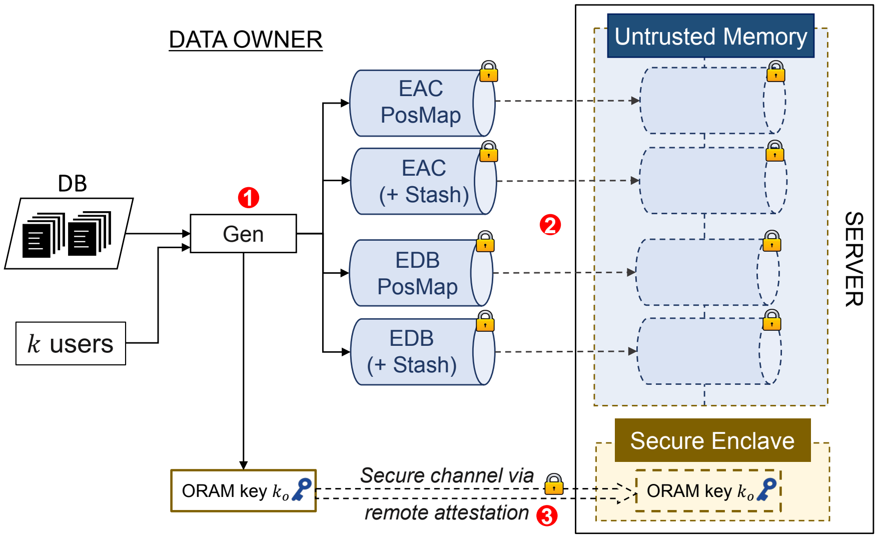


Figure 5.10: MOSE initialization.

```
$\mathsf{MOSE.Gen}(\lambda, \mathsf{DB}, k, N)$:
$k_o \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)$ and initialize $k \times N$ matrix $\mathsf{ACDS}$
$\mathsf{EDB} \leftarrow \text{Circuit-ORAM}.\mathsf{Setup}(k_o, \mathsf{DB})$
$\mathsf{ACDS}[\mathsf{uid}, \mathsf{id}] \leftarrow \{\mathsf{R}, \mathsf{RW}, \perp\}$ for $1 \leq \mathsf{id} \leq N$ and $1 \leq \mathsf{uid} \leq k$
$\mathsf{ACDS}' \leftarrow$ Split $\mathsf{ACDS}$ into $N'$ blocks each of size $|B'|$ bits
$\mathsf{EAC} \leftarrow \text{Circuit-ORAM}.\mathsf{Setup}(k_o, \mathsf{ACDS}')$
return $(k_o, \mathsf{EDB}, \mathsf{EAC})$
```

Figure 5.11: Setting up EAC and EDB components in MOSE.

In MOSE, we employ authenticated encryption technique (*i.e.*, AES-CTR for encryption and HMAC with SHA-256 for authentication) to achieve the integrity for each node in the Circuit-ORAM structures against a malicious adversary.

5.5.3.2 Handling User Request with Access Control

Figure 5.12 illustrates the workflow of MOSE in processing the user request with the access control check. Let id be the ID of the block in EDB the user uid wants to access. The user will first establish a secure channel (via a shared key K_s) with the enclave via remote attestation. The user encrypts the 4-tuple $(\mathsf{uid}, \mathsf{pwd}, \mathsf{id}, \mathsf{op})$ with the shared key K_s where pwd is user's authentication password and op denotes the access type (*e.g.*, $\mathsf{op} \in \{\text{read/write}\}$), and then sends the encrypted

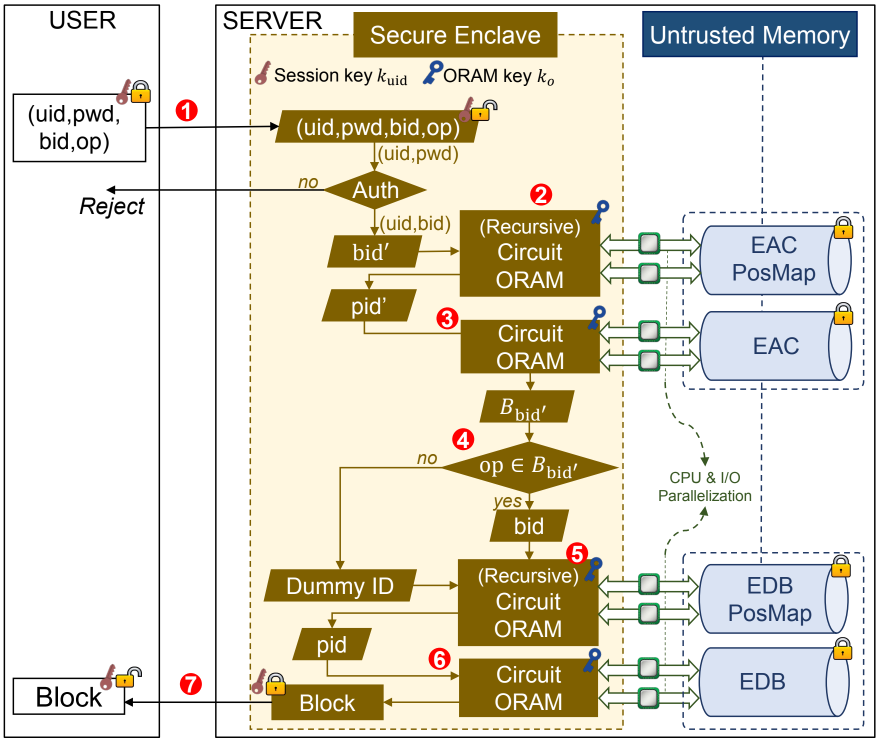


Figure 5.12: Oblivious access workflow in MOSE.

```
MOSE.AccessDB(uid, id, op, $b^*$):
$\mathsf{id}' \leftarrow (2N \cdot \mathsf{uid} + \mathsf{id})/|b'|$ # $N$:  # DB blocks, $|B'|$:  AC block size
$b' \leftarrow$ Circuit-ORAM.Access$\big(\mathsf{read}, \mathsf{id}', \perp; \mathsf{EAC}\big)$
perm $\leftarrow$ Get the permission of user uid on id from $b'$
$v \leftarrow \mathsf{ocmp}(\mathsf{op}, \mathsf{perm})$ # Check if op = perm
did $\xleftarrow{\$} \{1, \dots, N\}$
id $\leftarrow \mathsf{oupt}(v, \mathsf{id}, \mathsf{did})$; $b^* \leftarrow \mathsf{oupt}(v, b^*, \{0\}^{|b^*|})$
$b \leftarrow$ Circuit-ORAM.Access$\big(\mathsf{op}, \mathsf{id}, b^*; \mathsf{EDB}\big)$
$b \leftarrow \mathsf{oupt}(v, b, \{0\}^{|B|})$
$b' \leftarrow \mathcal{E}.\mathsf{Enc}_{K_s}(b)$
return $b'$ # Send $b'$ to uid
```

Figure 5.13: Processing user request in MOSE.

request to the storage server, which passes it into the enclave (Step ❶). The enclave decrypts the tuple (uid, pwd, id,op) using K_s, and authenticates the user with the uid and password pwd. If authenticated, it derives the ID id′ of a block in EAC containing the permission of uid with id. The enclave performs recursive Circuit-ORAM accesses on the position map component of EAC to retrieve the path location (pid′) of id′ in EAC (Step ❷). After that, the enclave performs a

Circuit-ORAM access on EAC to retrieve the block id' from the path pid' (Step ③), and checks whether op is permitted (*i.e.*, $\mathsf{op} \in \mathsf{id}'$) or not (Step ④). If $\mathsf{op} \in \mathsf{id}'$, the enclave performs recursive Circuit-ORAM accesses on the position map component of EDB to retrieve the path location of id (Step ⑤), and then executes a Circuit-ORAM access on EDB to retrieve the requested block id from the path (Step ⑥). Otherwise, if $\mathsf{op} \notin \mathsf{id}'$, the enclave performs *dummy* (recursive) accesses on both EDB's position map and EDB. Such dummy accesses are necessary in order to prevent the server from learning whether the user request is permitted or not. Finally, the enclave encrypts the accessed block (or dummy data if uid is not permitted) with K_s, and then sends the encrypted data to the user (Step ⑦).

Figure 5.13 presents how our enclave processes the access request algorithmically after the user is authenticated. The algorithm invokes the recursive Circuit-ORAM protocol (steps 2, 7). Generally speaking, the enclave needs to perform a conditional check to verify whether the user has permission or not (step step 4) and also require a check when the enclave obliviously accesses the database block from the path of the Circuit-ORAM tree. To prevent information leakage from diverging execution on a conditional branch, we implemented the oblivious comparison and oblivious update functions using `CMOV` and `SETE` instructions proposed in [142, 149] defined as follows.

- $b \leftarrow \mathsf{ocmp}(x, y)$: It takes as input two values x, y and outputs $b \leftarrow 0$ if $x = y$ or $b \leftarrow 1$ otherwise.
- $z \leftarrow \mathsf{oupt}(b, x, y)$: It takes as input two values x, y and a boolean b, and assigns $z \leftarrow x$ if $b = 1$ or $z \leftarrow y$ otherwise.

Notice that read and write operations (*i.e.*, ReadDB and $\mathsf{WriteDB}$ algorithms in Definition 17) must incur the same procedure presented in Figure 5.13 to achieve security. To be complete, Figure 5.14 presents the detailed $\mathsf{WriteAC}$ algorithm defined in Definition 17. Notice that this process

```
MOSE.WriteAC(uid, id, $p$):
$\mathsf{id}' \leftarrow (2N \cdot \mathsf{uid} + \mathsf{id})/|b'|$ # $N$:  # DB blocks, $|B'|$:  AC block size
$b' \leftarrow$ Circuit-ORAM.Access(write, $\mathsf{id}', p$; EAC)
```

Figure 5.14: User permission update in MOSE.

can only be triggered by the data owner, and the enclave executes this algorithm only after the data owner is authenticated.

5.5.3.3 Towards Scalable Oblivious Access

To achieve a scalable performance, we optimize each Circuit-ORAM access on encrypted data components stored in the untrusted memory via secure enclaves and parallelization of computation and I/O accesses. As briefly discussed in §5.5.1, the state-of-the-art proxy-based oblivious storage design (TaoStore [158]) pointed out the concurrency limitation of ORAM. That is, any optimization that parallelizes multiple ORAM access must reply such requests sequentially in their arrival order to prevent information leakages via their timings. Although multiple user's requests can arrive and be processed at the proxy simultaneously, the total response time for a user is equal to the total processing time of its prior requests plus the processing time for the target request, which includes local proxy computation and ORAM network communication delay.

Holding the restriction that all requests have to be replied in a sequential order to ensure security, the only solution to optimize MOSE's performance is to employ parallelization tricks, not for the concurrent ORAM access, but for minimizing the processing time of a single request. With this approach, we can minimize the delay of processing one user request at a time and also make MOSE scalable by increasing its concurrent request processing performance in proportional to the number of available CPU cores.

Another bottleneck that limits the support of concurrent requests is the network bandwidth overhead. The 1 Gbps link between the proxy and the storage server only allows *ten* concurrent access requests in a second, and this number cannot reflect concurrency needs in real-world scenarios. Thanks to MOSE's design that utilizes secure enclaves, the network overhead does not exist in MOSE because the enclave communicate with the storage via a high-speed memory bus, which is a 273 Gbps link for a regular dual-channel DDR4-2133 memory.

MOSE parallelizes computations as well as I/O accesses in each Circuit-ORAM access by utilizing a multi-core CPU as follows. We split each (real/dummy) block of Circuit-ORAM into multiple chunks, preferably as the number of parallelized threads, and encrypt each chunk separately and independently from one another using parallelizable encryption techniques such as AES-CTR. On performing a Circuit-ORAM access, the enclave spawns multiple threads to access such chunks in parallel. Each thread is responsible to read/write its assigned chunks from/to the server's storage device and also perform encryption/decryption. This parallelization is simple, but more efficient than [43], in which each CPU is assigned to an independent subtree in the ORAM tree structure, which might incur a costly synchronization and unbalanced CPU workloads. Moreover, because it only parallelizes a single ORAM access request, this parallelization is not vulnerable to the asynchronousity attack [158].

A caveat on applying this parallelization is that meta-data components in the Circuit-ORAM tree is very small in practice (less than 16 B), the parallelization for meta-data computation is not beneficial because initializing threads costs some delay. Therefore in MOSE, we use only one thread to load the encrypted meta-data from the storage disk into the enclave memory space first, and then creating multiple threads to process block data by chunks.

In this study, we instantiate ACDS with matrix, which is the basic structure for access control management. This design decision follows the same scalability principle: optimizing a single request. Although matrix is storage-costly (sized as # of user × # of database files) than more compact alternatives such as linked-list, it allows to get the user permission on a specific database block directly when packaged into the ORAM blocks. Specifically, the cell $\mathsf{ACDS}[\mathsf{uid}, \mathsf{id}]$ can be packaged into an ORAM block with ID $\mathsf{id}' = (2 \cdot \mathsf{uid} \cdot N + \mathsf{id})/|B'|$, where N is the number of database blocks, and $|B'|$ is the selected AC block size (in bits). On the contrary, with a linked-list structure, each entry for uid stores an array of block IDs that uid can access to (or in reverse). Because the list is different from each user, one might have to scan at least logarithmic number of entries in the list with multiple ORAM executions (*e.g.*, [182]), which will incur a high latency. Moreover, such structure might be vulnerable to information leakage via list size and access timing, which cannot be prevented by ORAM. Padding can mitigate this leakage, but it incurs more delay and is application-specific.

Note that the entire operation of MOSE is orthogonal to the selection of AC structure, so any structure can be utilized. However, for the given scalability goal and security requirements, we decide to sacrifice the storage for the other properties.

5.5.4 Security Analysis

We present the security notion for an enclave-assisted multi-user ORAM with access control, and show how MOSE achieves them. Our model is inspired from [128, 129, 158] with the following differences. In [128, 129], the security model captures multi-user ORAM and access control *without* a trusted proxy (modeled as a secure enclave in MOSE). In [158], it captures a multi-user ORAM with a trusted proxy over the asynchronous network setting but does not offer access control measures. In MOSE, the enclave resides on an untrusted server and therefore, the network setting does not apply. We consider two main adversary properties including *maliciousness*, where they can actively inject/tamper with the input and *collusion*, where both the user and server can collude with each other.

Intuitively, an enclave-assisted multi-user ORAM with access control security is secure if the server and an arbitrary subset of users, aside from what is trivially leaked by corrupted users, cannot learn any information regarding the access patterns of honest users. We define this notion as follows.

Definition 18 (Enclave-assisted multi-user ORAM with access control security). Consider the following experiment between an adversary $\mathcal{A}$, which contains a set of malicious and colluding entities (*e.g.*, users and server) and a challenger $\mathcal{C}$ (which works as $\mathsf{OClient}$).

- *Setup:* $\mathcal{C}$ generates a database DB with N blocks for k users and initializes two empty lists $\mathcal{UL}$ and $\mathcal{QL}$. It executes $(k_o, \mathsf{EAC}, \mathsf{EDB}) \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{DB}, k, N)$, and returns EAC and EDB to $\mathcal{A}$.
- *Learning phase:*
  - $O\mathsf{addU}(\mathsf{uid})$: This oracle adds a user and its secret key to the list of colluding users as $\mathcal{UL} \leftarrow \mathcal{UL} \cup \{\mathsf{uid}, k_S\}$.
  - $\mathsf{OAccess}(\mathsf{op}, \mathsf{uid}, \mathsf{id}, \mathsf{data})$: If $\mathsf{op} = \mathsf{read}$, it executes $\mathsf{ReadDB}(\mathsf{uid}, \mathsf{id}; \mathsf{EAC}, \mathsf{EDB})$. Otherwise, it executes $\mathsf{WriteDB}(\mathsf{uid}, \mathsf{id}, \mathsf{data}; \mathsf{EAC}, \mathsf{EDB})$ (*i.e.*, when $\mathsf{op} = \mathsf{write}$). It then executes $\mathsf{data}' \leftarrow \mathsf{Response}(K_s, \mathsf{data})$ and add $o_i = \langle \mathsf{op}, \mathsf{uid}, \mathsf{id}, \mathsf{data} \rangle, \mathbf{AP}(o_i)$ to $\mathcal{QL}$, where $\mathbf{AP}(o)_i$ is the access pattern when executing $\mathsf{WriteDB}$, ReadDB and $\mathsf{Response}$ protocols.
- *Distinguish phase:* $\mathcal{A}$ prepares two access requests $o_1 = \langle \mathsf{op}_1, \mathsf{uid}_1,$

  $\mathsf{id}_1, \mathsf{data}_1 \rangle$ and $o_2 = \langle \mathsf{op}_2, \mathsf{uid}_2, \mathsf{id}_2, \mathsf{data}_2 \rangle$.

- $\mathsf{Challenge}(o_1, o_2)$: $\mathcal{A}$ queries the challenge oracle with two access requests o_1 and o_2 defined above. If $\mathsf{uid}_1 \in \mathcal{UL}$ or $\mathsf{uid}_2 \in \mathcal{UL}$, $\mathcal{C}$ aborts. Otherwise, $\mathcal{C}$ flips a random coin $b \leftarrow \{0, 1\}$ and executes ReadDB or $\mathsf{WriteDB}$ depending on op_b and returns $\mathbf{AP}(o_b)$ to the adversary. The adversary can continue the learning phase with the exception of calling $O\mathsf{addU}(\mathsf{uid}_1)$ or $O\mathsf{addU}(\mathsf{uid}_2)$. $\mathcal{A}$ eventually outputs a bit b' to indicate if $\mathbf{AP}(o_b)$ corresponds to o_1 or o_2.

At the end of the game, the adversary wins if $b' = b$.

We also require that only the authorized users can learn about the database contents. Moreover, the adversary can not violate data integrity.

Theorem 10. MOSE is secure by Definition 18 if the underlying ORAM and IND-CPA encryption scheme are secure.

Proof. The game starts by running the Gen algorithm to initialize EDB, EAC and the ORAM key k_o. During the learning phase, the adversary can corrupt any user of its choice by simply querying its key. $\mathcal{A}$ can adaptively query the $\mathsf{OAccess}$ oracle on access requests of its choice. At some point, $\mathcal{A}$ decides to query the $\mathsf{Challenge}$ oracle. $\mathcal{A}$ prepares two access requests $o_1 = \langle \mathsf{op}_1, \mathsf{uid}_1, \mathsf{id}_1, \mathsf{data}_1 \rangle$ and $o_2 = \langle \mathsf{op}_2, \mathsf{uid}_2, \mathsf{id}_2, \mathsf{data}_2 \rangle$. Note that to rule out the trivial cases, we require that $\mathsf{uid}_1 \notin \mathcal{UL}$ or $\mathsf{uid}_2 \notin \mathcal{UL}$.

Upon initiating the $\mathsf{Challenge}$ oracle, $\mathcal{C}$ commits to a random coin $b \leftarrow \{0, 1\}$. For a $o_b = \langle \mathsf{op}_b, \mathsf{uid}_b, \mathsf{id}_b, \mathsf{data}', \mathsf{data}^* \rangle$ submitted to the $\mathsf{Challenge}$ oracle, whether op_b is a read or a write operation, two ORAM accesses take place. The first ORAM access is to EAC to determine the permission of uid_b on id_b. When $(p_b, \mathsf{EAC}') \leftarrow \langle \mathsf{ReadAC}_{\mathcal{A}(\mathsf{EAC})}(\mathsf{uid}_b, \mathsf{id}_b) \rangle$ is called, EAC is accessed by the underlying ORAM and the output (*e.g.*, p_b) is returned to $\mathcal{C}$. After determining

uid_b's permission on id_b, $\mathcal{C}$ performs either a real or dummy ORAM access for data request o_b on EDB. This leads to memory access pattern $\mathbf{AP}(o_b)$ on EDB where the output is returned through $\mathsf{data} \leftarrow \langle \mathsf{Response}(K_s, \mathsf{data}) \rangle$ to uid_b.

We now analyze the view of $\mathcal{A}$ for the access patterns and transcripts generated through the above accesses. First, both EAC and EDB are encrypted via an IND-CPA encryption scheme at all times. Second, when $(p_b, \mathsf{EAC}') \leftarrow \langle \mathsf{ReadAC}_{\mathcal{A}(\mathsf{EAC})}(\mathsf{uid}_b, \mathsf{id}_b) \rangle$ is called, $\mathcal{A}$ does not have any view on outputs to $\mathcal{C}$, therefore, it cannot infer any information about p_b. Moreover, since MOSE leverages a secure ORAM, to access EAC, any memory access pattern generated by ORAM is (computationally) indistinguishable by Definition 5 [41]. Third, based on the permission p_b, whether $\mathcal{A}$ has to perform a real or dummy access for request o_b on EDB, due to the security of the underlying ORAM, the generated access patterns $\mathbf{AP}(o_b)$ for $b \in \{0, 1\}$ is (computationally) indistinguishable by Definition 5. Lastly, the output of $\mathsf{Challenge}$ to $\mathcal{A}$ is encrypted with an IND-CPA encryption and therefore, indistinguishable for $\mathcal{A}$. In all the above cases, for $\mathcal{A}$ to distinguish between $\mathbf{AP}(o_b)$ based on o_1 and o_2, it has to break the underlying ORAM or IND-CPA encryption. □

Corollary 10. MOSE offers data secrecy and tamper resistance against malicious adversary.

Proof. Data secrecy in MOSE is based to the IND-CPA property of the underlying encryption scheme. As pointed out in the above proof, EDB and EAC remain encrypted at all the times via an IND-CPA encryption. Moreover, the returned values are also encrypted in the $\mathsf{Response}$ algorithm via the underlying IND-CPA encryption. The tamper resistance of MOSE is based on the underling keyed hash function (*e.g.*, SHA-256) which is used to provide HMAC for each database block. Finally, the integrity of the enforced access control mechanism ensures users cannot access (read/write) that they do not have permission on. □

Although the enclave's data is isolated and encrypted, attackers could indirectly learn about the data via cache [78, 122, 134] or page-fault [31, 83, 187] side-channel attacks. Specifically, the execution that retrieves a block from a read path of ORAM tree or accesses stash could leak information related to data to such side channels. In this regard, MOSE provide defenses against such attacks. On one hand, we access all blocks in a single path and the stash per each ORAM access to prevent cache side-channel attacks. On the other hand, we implement our logic in the enclave using `CMOV` instructions to remove conditional branches that potentially leaks via page-fault side-channel when the enclave verifies the user permission.

5.5.5 Experimental Evaluation

5.5.5.1 Implementation

We implemented MOSE in C/C++ using the SGX SDK v1.7. Our implementation contains approximately 2,982 lines of code for the untrusted modules and 780 lines of code for trusted modules. We leveraged `sgx_aes_ctr_encrypt()` and `sgx_read_rand()` functions in the SGX SDK library, for encrypting ORAM with AES-CTR mode and random number generation (via the `RDRAND` instruction), respectively. We used `pthread` for spawning multiple threads to support parallelism in secure enclave. Remark that we stored the position map components on the untrusted memory in the form of recursive Circuit-ORAM structures. We also stored the stash components on the untrusted memory, which are encrypted and loaded as plaintext only to the enclave memory space chunk-by-chunk (an `ECALL` function handles this). In the following, we outline the configurations and methodology of our evaluation (in §5.5.5.2), and then we evaluate the effectiveness of MOSE in terms of the delay when handling single/multiple client request(s) with/without optimization and the storage overhead.

5.5.5.2 *Configurations and Methodology*

We used a commodity desktop supporting SGX, which is equipped with a six-core Intel Core i7-8700K CPU @ 3.70 GHz, 32 GB of dual-channel DDR4-2133 memory, and 4 TB NVMe SSD drive.

We constructed a database DB containing 2^{25} (33,554,432) random database blocks of size 24 KB (4 KB × six threads). We assumed basic 2-bit access policies (read and/or write) for 2^{14} (16,384) users on such 2^{25} blocks.

We evaluated the performance of MOSE in terms of latency, throughput, and memory usage, by varying the database size, number of cores and ORAM cache level (see §5.5.5.3). Afterwards, we applied various optimizations such as position map caching and k-top level caching (see §5.5.5.5). We then analyze the effectiveness of MOSE in the multi-user environment, compare its performance with TaoStore [158] under various database sizes. (see §5.5.5.6). We did not compare with other proxy-based techniques [20, 158, 167] because their design is insecure against asynchronous timing attack (see §5.2). We also did not compare MOSE with non-proxy ORAM primitives [22, 41] because they incur high communication/computation overhead due to some cryptographic operations. For scalability testing, we created a number of virtual users that send concurrent access requests to MOSE with 50 ms network latency. We setup the following system parameters.

- MOSE: We selected standard parameters for Circuit-ORAM: bucket size $Z = 2$ with deterministic eviction and stash size $|\mathbf{S}| = 80$. To exploit parallelism on accessing the ORAM structure from the NVMe disk drive, we divided each block of EDB into n_t 4 KB chunks, where $n_t = 6$ being the number of threads for parallelism. We instantiated ACDS with a matrix for access control management of 2^{14} users on 2^{25} data blocks. We divided ACDS into 12- KB blocks, and built recursive Circuit-ORAM trees for such blocks and their position map. Similar to EDB, we divided each EAC block into n_t chunks, and each chunk is of size 2 KB. For the recursion, we selected the compression ratio $r = 256$.

- TaoStore [158]: We launched a simulation experiment for TaoStore with a conservative approach. Our virtual TaoStore used Path-ORAM because it was used by default in [158] and is the most efficient ORAM for (networked) client-server applications. We used the Path-ORAM standard parameters: $Z = 4$, $|S|= 80$ [169]. We selected 1 Gbps of network throughputs with 10 ms latency for simulating the execution of the Path-ORAM on the proxy. We excluded all execution delays at the storage server and the proxy such as I/O access, decryption/encryption, thread synchronization, etc. and only simulated the network delay of transmitting Path-ORAM paths caused by executing Path-ORAM protocols (in parallel) over the network between the proxy and the server. Our logic behind this experiment is that the network delay is inherent so it must be included, and adding any of implementation to the server and proxy will incur more delay on the execution side. Any actual implementation of TaoStore will involve more delay than this simulation.

5.5.5.3 Single Request Processing Time

We first present MOSE's response time in handling a single user request for various database sizes from 6 GB to 768 GB, while not applying any optimizations such as k-level cache, etc. (see Table 5.2). The size of database will affect the path length, so our enclave will read more data from the ORAM, and therefore will result in more delay. In the base-case design, the average total delay of MOSE is from 9.21 ms to 28.05 ms. In MOSE, the accesses on EAC and EDB can be slightly pipelined. That is, right after finishing the access on EAC from one request, MOSE can issue the access on EAC

Table 5.2: Delay of base-case MOSE in processing one user request.

DB	**Acc. Control** (ms)		**Database** (ms)		**Total**
Size	pm_{EAC}	**EAC**	pm_{EDB}	**EDB**	(ms)
6 GB	1.46	2.77	1.43	3.55	9.21
12 GB	1.64	2.78	1.54	3.72	9.62
24 GB	2.36	3.49	2.55	4.63	13.03
48 GB	2.51	3.79	3.46	6.29	16.05
96 GB	2.56	3.92	4.94	7.53	18.95
192 GB	2.78	4.13	6.05	8.7	21.66
384 GB	3.52	4.84	7.06	9.61	25.03
768 GB	3.67	5.14	7.97	11.27	28.05

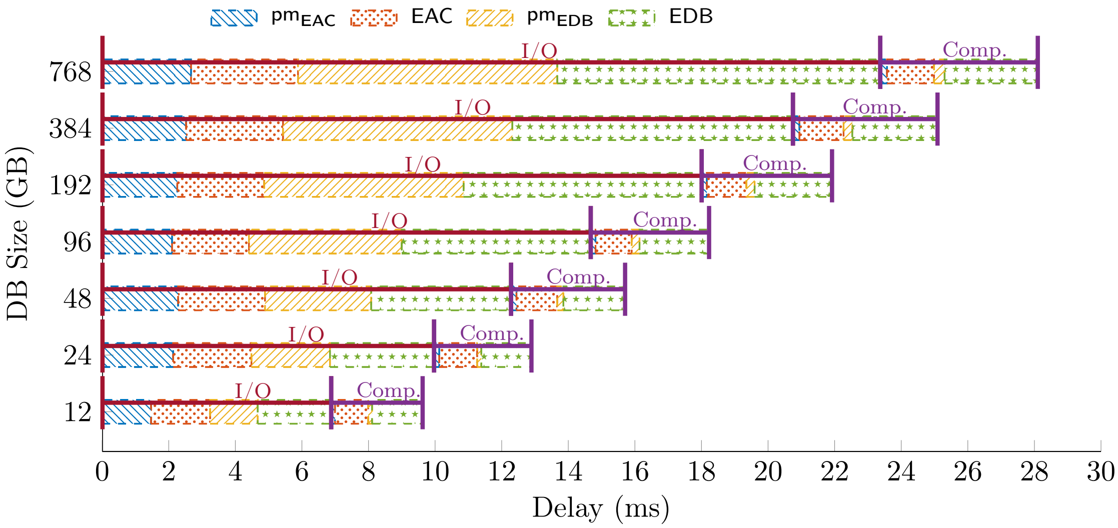


Figure 5.15: Delay breakdown of MOSE on a single user request.

of the next request while processing EDB access of the previous request. This pipelining strategy allows MOSE to achieve the throughputs ranging from 200 ops to 53 ops for database sizes from 6 GB to 768 GB, respectively.

Note that these numbers do not reflect the optimized performance. Next, we will further analyze factors that affect the total delay in MOSE, and then optimize them.

5.5.5.4 Delay Breakdown

We dissect the cost of a single request to understand the factors that affect MOSE's performance. As illustrated in Figure 5.15, MOSE has two major sources of delay: (i) the I/O accesses between enclave and disk incurred by Circuit-ORAM via encrypted read/write operations; (ii) the secure computation (*e.g.*, decryption/re-encryption/oblivious update operations) in the enclave. In the breakdown, we can observe that most delays in MOSE are caused by four different I/O accesses: The position map for the access control, the access control data, the position map for the database, and the database block. For the database accesses, the delay is caused by their tree ORAM structure, where each read/eviction operation requires accessing $\mathcal{O}(\log N)$ blocks located in random positions on the disk. For the position map accesses, 92% of its delay is caused by I/O access (uses only 8% of time for computation) because it is stored in the recursive ORAM structures, which require multiple access rounds. In the next section, we will evaluate optimization techniques applied to MOSE to reduce this I/O delay.

5.5.5.5 Optimized MOSE

To reduce the single-request delay, we implemented various optimization techniques including caching and parallelization to minimize the I/O access and computation delays.

We first cache the entire position map, stash, and metadata, all of which are required to make an ORAM access, in the main memory. For instance, the size of position map is only 5.6 MB for a 12 GB DB and 47.8 MB for a 96 GB DB, while the size of stash and metadata are approximately 400 MB. Due to their relatively small sizes, maintaining them in the main memory incurs a low overhead. The second and third bars in Figure 5.16 illustrate the outcome of this optimization. Caching the entire position map reduces the I/O delay from 14.93 ms to 8.03 ms

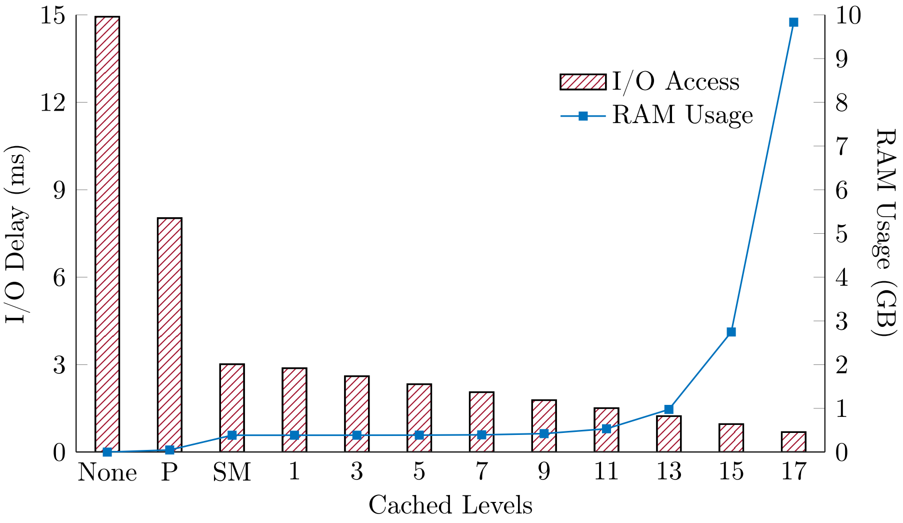


Figure 5.16: Impact of caching on the I/O delay and memory usage of MOSE scheme.

(46.21% reduction). Moreover, caching the stash and meta-data components reduces I/O delay from 8.03 ms to 3.01 ms (62.51% reduction) on top of the position map caching. In summary, applying the caching mechanisms reduces MOSE's I/O delay from 14.93 ms to 3.01 ms, which is 79.83% delay reduction (see the third bar of Figure 5.16), while incurring ≈ 500 MB of additional memory usage.

We also implemented the k-top caching strategy proposed in [127], which caches the first k levels of the EDB and EAC structures, to reduce I/O delay in ORAM access. The latter bars (4^{th}+) of Figure 5.16 illustrates the outcome of the caching. Increasing the number of cached levels reduces the I/O delay with the cost of a higher memory usage, however, note that performance gain increases in linear (by reading $H - k$ blocks where H is the tree height), and memory overhead increases exponentially (caching $2^{(k+1)}$ blocks for k-top caching). In this regard, we should set a practical limit of cached level by which the server can accommodate its memory overhead. Overall, we can reduce the I/O delay of MOSE from 14.93 ms to 0.68 ms by using around 10 GB of additional memory. As shown in Figure 5.16, caching around 50-70% levels of EDB and EAC provides a reasonable memory and I/O delay trade-off (*e.g.*, 11–13 levels with 1 GB RAM usage for 96 GB DB).

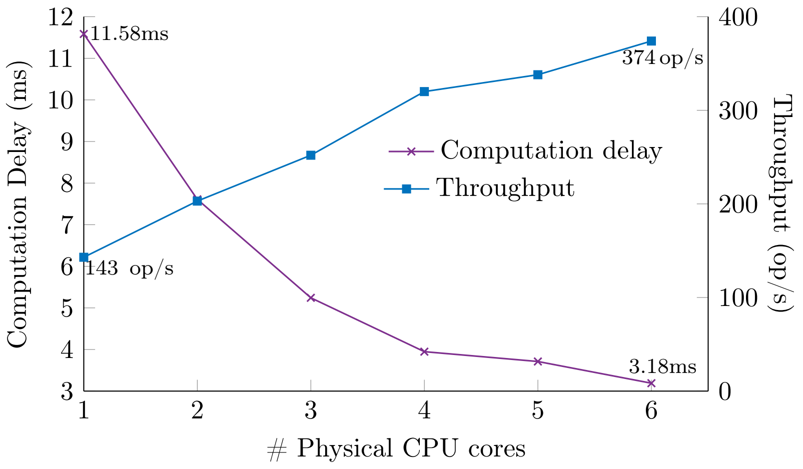


Figure 5.17: Impact of CPU cores on the efficiency and scalability of MOSE.

MOSE also leverages the parallelism in a multi-core CPU to speed up encryption/decryption operations in the enclave, and this optimization makes MOSE scalable. The purple line in Figure 5.17 illustrates the impact of utilizing multiple CPU cores in MOSE's computation logic. The performance of MOSE increases in proportion to the number of CPU cores that MOSE uses. The actual gain by a multi-core CPU is slightly lower than linear increment, *e.g.*, using 6 physical cores improved around 4× of the performance. This is because creating and assigning multiple threads into the corresponding physical CPU core costs a fixed and remarkable overhead.

When all optimization techniques above are applied, MOSE will take 3.74 ms in total to process an access request on a 24 KB block in a 96 GB DB, which consists of 0.56 ms I/O delay and 3.18 ms computation delay. This allows MOSE to process around 394 op/s with pipelining strategy.

MOSE's performance is scalable, *i.e.*, MOSE performs better with a more number of CPU cores on the server. The blue line in Figure 5.17 illustrates how the number of concurrent users for supporting less than one second delay increases as the number of cores that MOSE used for the experiment increases.

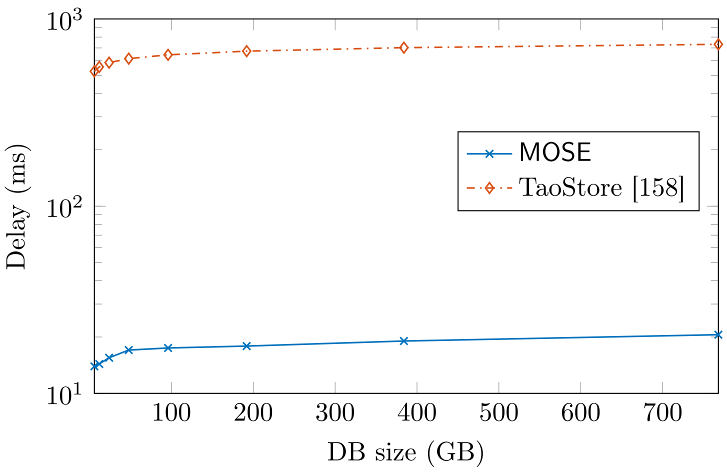


Figure 5.18: Performance of MOSE *vs.* TaoStore under 1 Gbps network.

5.5.5.6 Multi-User Concurrent Access Performance

We ran an experiment on a virtual network that assumes 50 ms for the user network latency, and it showed that on a 96 GB DB, MOSE can serve 374 concurrent requests, each accessing a 24 KB block without incurring more than a second delay for 374 users. In case if the server's service-level agreement (SLA) is to ensure average expected delay for each user to be one second, MOSE can serve around 788 concurrent requests.

We compare the performance of MOSE with TaoStore [158] with varied database sizes. We considered *ten* concurrent requests[27] as originally presented in [158] for a fair performance comparison. For the experiment, we simulated the performance of TaoStore according to 24 KB block size and varied database sizes ranging from 1 GB to 1 TB. Figure 5.18 presents the delay of MOSE and TaoStore. Note that since our database and block size are larger than that of [158], the reported delay of TaoStore in Figure 5.18 is higher than what is originally presented in [158]. Given TaoStore is equipped with a highly dedicated network (*i.e.*, 1 Gbps throughput), MOSE is still

[27] The notion of concurrency in TaoStore is limited by the communication bandwidth between the server and the trusted proxy. The reason for having maximum ten concurrent users in their evaluation is mainly because ten concurrent requests will max-out the bandwidth. Having more request than ten at the same time will drastically degrade its performance, as stated in their paper.

around 34.80–37.8× faster than TaoStore where MOSE (with optimization) achieved (on average) 13–21 ms delay for each request, compared with 525–731ms (on average) in TaoStore.

5.5.5.7 Storage Overhead

MOSE stores four encrypted components in the server's untrusted storage device: EAC and EDB, and their position maps. All of them are in the form of Circuit-ORAM tree structures, which incur a constant (≈ 2×) storage blowup. Specifically, for the 96 GB DB with 2^{14} users, EDB and EAC cost approximately 206.8 GB and 103.7 GB, respectively. The position map components cost approximately 48.8 MB (1/2000 of DB size).

Chapter 6: Conclusion

In this dissertation, we developed a series of privacy-enhancing technologies featuring a high level of privacy and security while offering mandatory functionalities for critical cyberinfrastructures. We summarize our contributions as follows.

- *Efficient searchable encryption techniques for critical cloud storage services:* We introduced IM-DSSE frameworks and FS-DSSE scheme, both of which offer mandatory security and privacy features (*e.g.*, forward privacy, backward privacy and size obliviousness) against statistical analysis attacks, while achieving a high level of efficiency (*i.e.*, low end-to-end delay, low computation overhead).

- *Efficient distributed ORAM schemes for data outsourcing applications:* We proposed novel distributed ORAM schemes for data outsourcing applications including S^{3}ORAM and MACAO. Both S^{3}ORAM and MACAO harness secret sharing techniques, which offer efficient homomorphic properties to achieve desirable performance and security such as low delay, low storage, low communication and computation overhead, and security against active adversaries. The experiments on commodity cloud platform confirmed the efficiency of S^{3}ORAM and MACAO compared with the state-of-the-art.

- *Efficient oblivious data structures for secure searchable encryption and database services:* We proposed DOD-DSSE and ODSE as oblivious data structures to mitigate/seal the search and update pattern leakage on the encrypted index in searchable encryption. These techniques offer higher efficiency than the composition of generic ORAM and searchable encryption. We proposed OMAT and OTREE as oblivious data structures that can be integrated with tree-based ORAM scheme to enable oblivious queries on legacy database management systems (*e.g.*, MongoDB, SQL). The experiments on real commodity cloud platform confirmed the efficiency of the proposed techniques.

- *Efficient hardware-supported data storage and query platforms:* We designed POSUP, an efficient oblivious search and update platform for personal data outsourcing. By using Intel SGX to implement ORAM controller directly on the untrusted server, POSUP mitigates the network impact of ORAM, thereby featuring very low latency when tested on very large dataset with commodity hardware. We further designed MOSE, an oblivious storage platform for multi-user setting sharing the same database. By implementing proxy logics with Intel SGX, MOSE supports oblivious access, access control and concurrent access in a much more efficient manner than the state-the-art solutions.

6.1 Future Work

We briefly outline some of our potential research directions as follows.

We first focus on building practical oblivious distributed access systems. Distributed File System (DFS) allows multiple clients to efficiently store and access the files remotely on the server(s) anytime anywhere. It is critical to enable essential security services on DFS such as oblivious access, access control, confidentiality and integrity. Although ORAM can offer ideal breach-resiliency properties (*e.g.*, oblivious access, confidentiality, integrity) for DFS, there is still a critical research gap toward the integration of ORAM with DFS architecture. For instance, ORAM is mostly designed in the single-client setting, and therefore it may not be directly compatible with the multi-client setting in DFS, which requires access control and parallel access features. Some client-efficient ORAMs only offer security against semi-honest adversaries [89]. In practice, an active adversary (*e.g.*, malware) will likely be present in the DFS execution environment, who may inject malicious inputs to compromise the system security and integrity. Meanwhile, the extension of such ORAMs into the malicious setting may completely invalidate all client-efficiency gains [44, 54].

Toward filling this gap, we consider two research tasks as follows. First, we will propose several new ORAM schemes for the DFS setting. Specifically, we will create new multi-client ORAM schemes, which offer critical DFS features such as parallel access, access control enforcement and authorization. We will seek possible solutions that rely only on cryptographic primitives. Our second research task will focus on the creation of a full-fledged Oblivious DFS (ODFS) platform by integrating modern DFS techniques with the new ORAM schemes. We will seek the most efficient DFS and then analyze its unique characteristics and system requirements for the integration. This research is expected to enable breach-resilient oblivious file systems that can serve as the core building block to construct trustworthy and privacy-preserving data storage and analytics platforms with a high level of security and efficiency.

Our second research direction focuses on efficient privacy-preserving machine learning. Machine Learning (ML) serves as the backbone of numerous intelligent applications such as virtual assistant platforms, intrusion detection, automation and sophisticated surveillance. Since the data to be processed by ML algorithms may be sensitive (*e.g.*, personal data, location, biometrics), it is critical to ensure their security and privacy during the processing. However, there is a substantial research gap toward the creation of such secure, privacy-preserving yet practical ML techniques. Existing Privacy-Preserving Machine Learning (PPML) methods generally rely on MPC and Fully HE (FHE) techniques. Simply using these functional encryptions in the context of big data processing may suffer from not only high computation/communication overhead but also accuracy degradation. For example, FHE and MPC offer limited arithmetic operations (*e.g.*, addition, multiplication), which may not be sufficient to realize complex yet critical ML subroutines (*e.g.*, non-linear activation layer, max-pooling functions). As a result, existing PPMLs [72, 133, 157] generally simplified ML algorithms (*e.g.*, limited non-linear layer, no convolution, limited number of instances) to fit with these techniques, and therefore, the accuracy is decreased. The extension of FHE and MPC protocols to the malicious setting incurs high communication and computation overhead.

Given that MPC, HE and TEE are three major tools to enable encrypted computation, we will investigate the synergies between them to achieve the optimal trade-offs in terms of security, performance and efficiency for specific application requirements. For instance, we will address the security limitations of MPC-based approaches (*e.g.*, collusion, active security) by bootstrapping them with TEE. We will develop secure fundamental arithmetic operations (*e.g.*, addition, multiplication, division, factorization, floating-point operations) under the MPC and TEE composition models, which will be then used as the core building blocks to develop kernel PPML algorithms. The strategic use of secure hardware to complement algorithmic limitations of MPC will enable a breach-, collision- and failure-resilient execution of ML techniques in the untrusted environments. We will propose new PPML schemes based on the hybrid composition between FHE and MPC techniques, in such a way that the computation and communication burdens are balanced, to achieve an optimal performance regarding specific application/system constraints. As polynomial approximations in the activation functions may impact both the complexity and accuracy of ML algorithms, we will also seek a correct approximation method for each privacy-preserving scenario to achieve a good efficiency and performance trade-off. Finally, instead of simplifying ML algorithms to fit with the underlying cryptographic primitives, we will realize their native form by synergizing efficient MPC, HE and TEEs altogether to preserve their original accuracy.

We will also focus on designing efficient privacy-preserving data (pre)processing and feature extraction schemes. Although these processing phases may greatly impact the overall efficiency of the system, they are less likely to be investigated in the privacy-preserving context. We will then build full-fledged PPML platforms for some critical applications (*e.g.*, anomaly detection, biometric identification/authentication) by harnessing all the developed techniques along with optimizations. To this end, we will seek the possible integration between these new PPML platforms with the oblivious access systems proposed above. This integration is expected to enable breach-resilient multi-functional cloud platforms that can offer privacy-preserving data storage and ML services simultaneously.

Finally, although PPMLs based on functional encryption offer a very high level of privacy, they are not flexible due to their strict bond to a certain ML algorithm. Because many new ML techniques are being proposed and renovated regularly, such cryptographic-based PPMLs must be re-customized frequently to cope with these changes. Therefore, We will also investigate alternative approaches such as federated learning or differential privacy, which focus more on preserving the privacy of the input/output data rather than the entire ML computation. Although these approaches may not offer the same level of security as cryptography-based PPMLs, they are more flexible and efficient to be adopted with rapid changes in ML technologies while offering an adequate level of privacy.

References

[1] The clusion library. https://github.com/encryptedsystems/Clusion/.

[2] The enron email dataaset. http://www.cs.cmu.edu/~enron/.

[3] Zeromq distributed messaging. Available at http://zeromq.org.

[4] sparsehash: An extremely memory efficient hash_map implementation. Available at https://code.google.com/p/sparsehash/, February 2012.

[5] Ittai Abraham, Christopher W Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. Asymptotically tight bounds for composing oram with pir. In *IACR Int. Workshop Public Key Cryptography*, pages 91–120. Springer, 2017.

[6] Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to efficiently evaluate ram programs with malicious security. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 702–729. Springer, 2015.

[7] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A commodity obfuscation engine on intel sgx. In *NDSS*, 2019.

[8] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *NDSS*, 2018.

[9] Anastasov Anton. Implementing onion oram: A constant bandwidth oram using ahe. https://github.com/aanastasov/onion-oram/blob/master/doc/report.pdf, 2016.

[10] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *International Workshop on Public Key Cryptography*, pages 131–148. Springer, 2014.

[11] attardi. WikiExtractor. https://github.com/attardi/wikiextractor.

[12] Adam J Aviv, Seung Geol Choi, Travis Mayberry, and Daniel S Roche. Oblivisync: Practical oblivious file backup and synchronization. In *NDSS*, 2017.

[13] Chongxi Bao and Ankur Srivastava. Exploring timing side-channel attacks on path-orams. In *2017 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 68–73. IEEE, 2017.

[14] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[15] R. Behnia, M. O. Ozmen, and A. A. Yavuz. Lattice-based public key searchable encryption from experimental perspectives. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2018.

[16] Rouzbeh Behnia, Attila Altay Yavuz, and Muslum Ozgur Ozmen. High-speed high-security public key encryption with keyword search. In Giovanni Livraga and Sencun Zhu, editors, *Data and Applications Security and Privacy XXXI*, pages 365–385, Cham, 2017. Springer International Publishing.

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

[18] Aner Ben-Efraim and Eran Omri. Turbospeedz: Double your online spdz! improving SPDZ using function dependent preprocessing. In *Applied Cryptography and Network Security — ACNS 2019*, June 5-7, 2019. To appear. Available at https://eprint.iacr.org/2019/080.

[19] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 1–10. ACM, May 2-4, 1988.

[20] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 837–849. ACM, 2015.

[21] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. *IACR Cryptology ePrint Archive*, 2019:1175, 2019.

[22] Erik-Oliver Blass, Travis Mayberry, and Guevara Noubir. Multi-client oblivious ram secure against malicious servers. In *International Conference on Applied Cryptography and Network Security*, pages 686–707. Springer, 2017.

[23] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 203–214. ACM, 2014.

[24] D. Boneh, G. D. Crescenzo, Rafail Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proc. 23th Int. Conf. the Theory and Applications of Cryptographic Techn. (EUROCRYPT '04)*, pages 506–522, 2004.

[25] Christoph Bösch, Andreas Peter, Bram Leenders, Hoon Wei Lim, Qiang Tang, Huaxiong Wang, Pieter Hartel, and Willem Jonker. Distributed searchable symmetric encryption. In *Privacy, Security and Trust (PST), 2014 Twelfth Annu. Int. Conf. on*, pages 330–337. IEEE, 2014.

[26] Raphael Bost. Sophos - forward secure searchable encryption. In *Proc. 2016 ACM Conf. Comput. Commun. Security*. ACM, 2016.

[27] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proc. 2017 ACM SIGSAC Conf. Comput. Commun. Security*, pages 1465–1482. ACM, 2017.

[28] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram and applications. In *Theory of Cryptography Conference*, pages 175–204. Springer, 2016.

[29] Elette Boyle and Moni Naor. Is there an oblivious ram lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 357–368. ACM, 2016.

[30] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical.

[31] Jo Van Bulck, Nico Weichbrodt, R. Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security*, 2017.

[32] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13(1):143–202, 2000.

[33] Ning Cao, Cong Wang, Ming Li, Kui Ren, and Wenjing Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Trans. Parallel Distributed Syst.*, 25(1):222–233, 2014.

[34] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proc. 22nd ACM CCS*, pages 668–679, 2015.

[35] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawcyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *21th Annu. Network Distributed System Security Symp. — NDSS 2014*. The Internet Soc., February 23-26, 2014.

[36] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology, CRYPTO 2013*, volume 8042 of *Lecture Notes in Comput. Sci.*, pages 353–373, 2013.

[37] David Cash and Stefano Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, pages 351–368. Springer, 2014.

[38] Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed Kosba, Ari Juels, and Elaine Shi. Solidus: Confidential distributed ledger transactions via pvorm. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 701–717, 2017.

[39] Anrin Chakraborti, Adam J Aviv, Seung Geol Choi, Travis Mayberry, Daniel S Roche, and Radu Sion. roram: Efficient range oram with o (log2 n) locality. In *NDSS*, 2019.

[40] T-H Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. More is less: Perfectly secure oblivious algorithms in the multi-server setting. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 158–188. Springer, 2018.

[41] T-H Hubert Chan and Elaine Shi. Circuit opram: Unifying statistically and computationally secure orams and oprams. In *Theory of Cryptography Conference*, pages 72–107. Springer, 2017.

[42] Zhao Chang, Dong Xie, and Feifei Li. Oblivious ram: a dissection and experimental evaluation. *Proceedings of the VLDB Endowment*, 9(12):1113–1124, 2016.

[43] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel ram: improved efficiency and generic constructions. In *Theory of Cryptography Conference*, pages 205–234. Springer, 2016.

[44] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring oram: Efficient constant bandwidth oblivious ram from (leveled) tfhe. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 345–360, 2019.

[45] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal ACM (JACM)*, 45(6):965–981, 1998.

[46] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org/2016/086.pdf.

[47] Victor Costan, Ilia Lebedev, Srinivas Devadas, et al. Secure Processors Part II: Intel SGX security analysis and MIT Sanctum Architecture. *Foundations and Trends® in Electronic Design Automation*, 11(3):249–361, 2017.

[48] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 727–743, 2018.

[49] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 79–88. ACM, 2006.

[50] Ivan Damgård and Mads Jurik. A generalisation, a simpli. cation and some applications of paillier's probabilistic public-key system. In *International Workshop on Public Key Cryptography*, pages 119–136. Springer, 2001.

[51] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012*, pages 643–662. Springer, 2012.

[52] Jonathan Dautrich and Chinya Ravishankar. Combining oram with pir to minimize bandwidth costs. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 289–296. ACM, 2015.

[53] Tom St Denis. LibTomCrypt library. Available at http://libtom.org/?page=features&newsitems=5&whatfile=crypt, Released May 12th, 2007.

[54] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*, pages 145–174. Springer, 2016.

[55] Judicael B Djoko, Jack Lange, and Adam J Lee. Nexus: Practical and secure access control on untrusted storage platforms using client-side sgx. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 401–413. IEEE, 2019.

[56] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 523–535. ACM, 2017.

[57] Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. An argument for increasing tcp's initial congestion window. *Comput. Commun. Rev.*, 40(3):26–33, 2010.

[58] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. Efficient dynamic searchable encryption with forward privacy. *Proc. Privacy Enhancing Technologies*, 2018(1):5–20, 2018.

[59] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party oram for secure computation. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 360–385. Springer, 2015.

[60] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. Iron: functional encryption using intel sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 765–782. ACM, 2017.

[61] Christopher Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket oram: single online roundtrip, constant bandwidth oblivious ram. Technical report, IACR Cryptology ePrint Archive, Report 2015, 1065, 2015.

[62] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8. ACM, 2012.

[63] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. In *ASPLOS '15*, 2015.

[64] Christopher W Fletcher, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 213–224. IEEE, 2014.

[65] Zhangjie Fu, Xinle Wu, Chaowen Guan, Xingming Sun, and Kui Ren. Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement. *IEEE Trans. Inform. Forensics Security*, 11(12):2706–2716, 2016.

[66] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. Hardidx: practical and secure index with sgx. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 386–408. Springer, 2017.

[67] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: round-optimal oblivious ram with applications to searchable encryption. Technical report, IACR Cryptology ePrint Archive, 2015: 1010, 2015.

[68] Rosario Gennaro, Michael O Rabin, and Tal Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 101–111. ACM, 1998.

[69] Craig Gentry. *A fully homomorphic encryption scheme.* PhD thesis, Stanford University, 2009.

[70] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. 41st Annu. ACM Symp. Theory of computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[71] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 1–18. Springer, 2013.

[72] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

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

[74] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194. ACM, 1987.

[75] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation oblivious rams. *Journal ACM (JACM)*, 43(3):431–473, 1996.

[76] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 513–524, New York, NY, USA, 2012. Association for Computing Machinery.

[77] S Dov Gordon, Jonathan Katz, and Xiao Wang. Simple and efficient two-server oram. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 141–157. Springer, 2018.

[78] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec)*, 2017.

[79] Matthew D Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *Security Privacy (SP), 2015 IEEE Symp. on*, pages 305–320. IEEE, 2015.

[80] Shay Gueron. White Paper: Intel Advanced Encryption Standard (AES) New Instructions Set. Available at https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf, Document Revision 3.01, September 2012.

[81] Venkatesan Guruswami and Madhu Sudan. Improved decoding of reed-solomon and algebraic-geometric codes. In *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pages 28–37. IEEE, 1998.

[82] Florian Hahn and Florian Kerschbaum. Searchable encryption with secure and efficient updates. In *Proc. 2014 ACM SIGSAC Conf. Comput. and Commun. Security*, pages 310–320. ACM, 2014.

[83] M. Hähnel, W. Cui, and M. Peinado. High-Resolution Side Channels for Untrusted Operating Systems.

[84] Ryan Henry, Amir Herzberg, and Aniket Kate. Blockchain access privacy: Challenges and directions. *IEEE Security & Privacy*, 16:38–45, 07 2018.

[85] Thang Hoang. Im-dsse framework implementation. https://github.com/thanghoang/IM-DSSE, 2017.

[86] Thang Hoang, Rouzbeh Behnia, Yeongjin Jang, and Attila A Yavuz. MOSE: Practical multi-user oblivious storage via secure enclaves. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pages 17–28. ACM, 2020. Reprinted. *The final publication is available at ACM Digital Library through https://doi.org/10.1145/3374664.3375749*.

[87] Thang Hoang, Jorge Guajardo, and Attila A Yavuz. MACAO: A maliciously-secure and client-efficient active ORAM framework. In *Annual Network and Distributed System Security Symposium, NDSS*, 2020. Reprinted. *The final publication is available at https://doi.org/10.14722/ndss.2020.24313*.

[88] Thang Hoang, Ceyhun D Ozkaptan, Gabriel Anton Hackebeil, and Attila Altay Yavuz. Efficient oblivious data structures for database services on the cloud. *IEEE Transactions on Cloud Computing*, 2018. Reprinted. *The final publication is available at IEEE Xplore through https://doi.org/10.1109/TCC.2018.2879104*.

[89] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. S3ORAM: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 491–505. ACM, 2017. Reprinted. *The final publication is available at ACM Digital Library through https://doi.org/10.1145/3133956.3134090*.

[90] Thang Hoang and Ceyhun Ozkaptan D. Implementation of s3oram. Available at https://github.com/thanghoang/S3ORAM, 2017.

[91] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. Hardware-supported oram in effect: Practical oblivious search and update on very large dataset. *Proceedings on Privacy Enhancing Technologies*, 2019(1):172–191, 2019. Reprinted. *The final publication is available at Sciendo through https://doi.org/10.2478/popets-2019-0010*.

[92] Thang Hoang, Attila Yavuz, and Jorge Guajardo. Practical and secure dynamic searchable encryption via oblivious access on distributed data structure. In *Proc. 32nd Annu. Comput. Security Applications Conf. (ACSAC)*. ACM, 2016. Reprinted. *The final publication is available at ACM Digital Library through https://doi.org/10.1145/2991079.2991088*.

[93] Thang Hoang, Attila A Yavuz, F Betül Durak, and Jorge Guajardo. A multi-server oblivious dynamic searchable encryption framework. *Journal of Computer Security*, pages 1–28. Reprinted. *The final publication is available at IOS Press through https://doi.org/10.3233/JCS-191300.*

[94] Thang Hoang, Attila A Yavuz, F Betül Durak, and Jorge Guajardo. Oblivious dynamic searchable encryption on distributed cloud systems. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 113–130. Springer, 2018. Reprinted. *The final publication is available at Springer through https://doi.org/10.1007/978-3-319-95729-6_8.*

[95] Thang Hoang, Attila A Yavuz, and Jorge Guajardo. A multi-server oram framework with constant client bandwidth blowup. *ACM Transactions on Privacy and Security (TOPS)*, 23(1):1–35, 2020. Reprinted. *The final publication is available at ACM Digital Library through https://doi.org/10.1145/3369108.*

[96] Thang Hoang, Attila Altay Yavuz, and Jorge Guajardo. Practical and secure dynamic searchable encryption via oblivious access on distributed data structure. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 302–313, 2016. Reprinted. *The final publication is available at ACM Digital Library through https://doi.org/10.1145/2991079.2991088.*

[97] Thang Hoang, Attila Altay Yavuz, and Jorge Guajardo Merchan. A secure searchable encryption framework for privacy-critical cloud storage services. *IEEE Transactions on Services Computing*, 2019. Reprinted. *The final publication is available at IEEE Xplore through https://doi.org/10.1109/TSC.2019.2897096.*

[98] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data.

[99] iMatix. Zeromq distributed messaging library 4.1.3. http://zeromq.org/community, September 2015. [Online; accessed September 2015].

[100] Intel Corporation. Intel Software Guard Extensions Programming Reference (rev1), September 2013. 329298-001US.

[101] Intel Corporation. Intel Software Guard Extensions Programming Reference (rev2), October 2014. 329298-002US.

[102] Intel Corporation. Intel Software Guard Extensions SDK for Linux OS (Developer Reference), 2016. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.7_Open_Source.pdf.

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

[104] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 235–246, 2014.

[105] Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

[106] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack.

[107] Seny Kamara and Tarik Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. *EUROCRYPT 2017*, 2017.

[108] Seny Kamara and Charalampos Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security (FC)*, volume 7859 of *Lecture Notes in Comput. Sci.*, pages 258–274. Springer Berlin Heidelberg, 2013.

[109] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proc. 2012 ACM Conf. Comput. Commun. security*, pages 965–976, New York, NY, USA, 2012. ACM.

[110] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography.* CRC Press, 2014.

[111] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1340, 2016.

[112] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making spdz great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 158–189. Springer, 2018.

[113] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for mpc. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 506–525. Springer, 2014.

[114] Marcel Keller and Avishay Yanai. Efficient maliciously secure multiparty computation for ram. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 91–124. Springer, 2018.

[115] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *Proc. 2017 ACM SIGSAC Conf. Comput. Commun. Security*, pages 1449–1463. ACM, 2017.

[116] Eyal Kushilevitz and Tamer Mour. Sub-logarithmic distributed oblivious ram with small block size. In *IACR International Workshop on Public Key Cryptography*, pages 3–33. Springer, 2019.

[117] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. *2018 IEEE Symposium on Security and Privacy (SP)*, pages 297–314, 2017.

[118] Russell WF Lai and Sherman SM Chow. Forward-secure searchable encryption on labeled bipartite graphs. In *Int. Conf. Appl. Cryptography Network Security*, pages 478–497. Springer, 2017.

[119] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *Annual International Cryptology Conference*, pages 523–542. Springer, 2018.

[120] Duc Van Le, Lizzy Tengana Hurtado, Adil Ahmad, Mohsen Minaei, Byoungyoung Lee, and Aniket Kate. A tale of two trees: One writes, and other reads. optimized oblivious accesses to large-scale blockchains. *ArXiv*, abs/1909.01531, 2019.

[121] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent B Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, pages 523–539, 2017.

[122] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*, 2017.

[123] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS '15*, 2015.

[124] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-an Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Inform. Sci.*, 265:176–188, 2014.

[125] Jacob R Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*, pages 199–213, 2013.

[126] Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In *Theory of Cryptography*, pages 377–396. Springer, 2013.

[127] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 311–324. ACM, 2013.

[128] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Privacy and access control for outsourced personal records. In *2015 IEEE Symposium on Security and Privacy*, pages 341–358. IEEE, 2015.

[129] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schroder. Maliciously secure multi-client oram. In *International Conference on Applied Cryptography and Network Security*, pages 645–664. Springer, 2017.

[130] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient private file retrieval by combining oram and pir. In *NDSS*. Citeseer, 2014.

[131] Tarik Moataz, Erik-Oliver Blass, and Travis Mayberry. Chf-oram: A constant communication oram without homomorphic encryption.

[132] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication oram with small blocksize. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 862–873. ACM, 2015.

[133] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38. IEEE, 2017.

[134] Urs Müller. Software grand exposure:{SGX} cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies,{WOOT} 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX, 2017.

[135] Vaishali Narkhede, Karuna Joshi, Adam J Aviv, Seung Geol Choi, Daniel S Roche, and Tim Finin. Managing cloud storage obliviously. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 990–993. IEEE, 2016.

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

[137] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *Proc. 22nd ACM CCS*, pages 644–655, 2015.

[138] Muhammad Naveed, Manoj Prabhakaran, and Carl A. Gunter. Dynamic searchable encryption via blind storage. In *35th IEEE Symp. Security Privacy*, pages 48–62, May 2014.

[139] Kartik Nayak, Christopher W Fletcher, Ling Ren, Nishanth Chandran, Satya V Lokam, Elaine Shi, and Vipul Goyal. Hop: Hardware makes obfuscation practical. In *NDSS*, 2017.

[140] Kartik Nayak and Jonathan Katz. An oblivious parallel ram with o (log2 n) parallel runtime blowup. *IACR Cryptology ePrint Archive*, 2016:1141, 2016.

[141] Mark EJ Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[142] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, pages 619–636, 2016.

[143] Muslum Ozgur Ozmen, Thang Hoang, and Attila A Yavuz. Forward-private dynamic searchable symmetric encryption with efficient search. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018. Reprinted. *The final publication is available at IEEE Xplore through https://doi.org/10.1109/ICC.2018.8422480*.

[144] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

[145] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Proc. 30th Annu. Conf. Advances in Cryptology*, CRYPTO'10, pages 502–519, Berlin, Heidelberg, 2010. Springer-Verlag.

[146] Julius Plenz. nocache - minimize filesystem caching effects. Available at https://github.com/Feh/nocache.

[147] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: processing queries on an encrypted database. *Communications of the ACM*, 55(9):103–111, 2012.

[148] David Pouliot and Charles V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proc. 2016 ACM Conf. Comput. Commun. Security*. ACM, 2016.

[149] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium*, pages 431–446, 2015.

[150] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. *IACR Cryptology ePrint Archive*, 2014:997, 2014.

[151] Ling Ren, Christopher W Fletcher, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. Design and implementation of the ascend secure processor. *IEEE Transactions on Dependable and Secure Computing*, 16(2):204–216, 2017.

[152] Ling Ren, Xiangyao Yu, Christopher W Fletcher, Marten Van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious ram in secure processors. *ACM SIGARCH Computer Architecture News*, 41(3):571–582, 2013.

[153] Panagiotis Rizomiliotis and Stefanos Gritzalis. Oram based forward privacy preserving dynamic searchable symmetric encryption schemes. In *Proc. 2015 ACM Workshop Cloud Computing Security Workshop*, pages 65–76. ACM, 2015.

[154] Daniel S Roche, Adam Aviv, and Seung Geol Choi. A practical oblivious map data structure with secure deletion and history independence. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 178–197. IEEE, 2016.

[155] Daniel S Roche, Adam Aviv, Seung Geol Choi, and Travis Mayberry. Deterministic, stash-free write-only oram. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 507–521. ACM, 2017.

[156] Cédric Van Rompay, Refik Molva, and Melek Önen. A leakage-abuse attack against multi-user searchable encryption. *Proceedings on Privacy Enhancing Technologies*, 2017:168 – 178, 2017.

[157] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, page 2. ACM, 2018.

[158] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 198–217. IEEE, 2016.

[159] Sajin Sasy, Sergey Gorbunov, and Christopher Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

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

[161] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In *Proc. 17th Int. Conf. The Theory and Application of Cryptology and Inform. Security*, ASIACRYPT'11, pages 197–214, Berlin, Heidelberg, 2011. Springer-Verlag.

[162] Victor Shoup. Ntl: A library for doing number theory. Available at http://www.shoup.net/ntl/, 2016.

[163] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proc. 2000 IEEE Symp. Security and Privacy*, pages 44–55, 2000.

[164] Xiangfu Song, Changyu Dong, Ddadan Yuan, Qiuliang Xu, and Minghao Zhao. Forward private searchable symmetric encryption with optimized i/o efficiency. *IEEE Trans. Dependable Secure Computing*, 2018.

[165] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014*. The Internet Society, February 23-26 2014.

[166] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 247–258. ACM, 2013.

[167] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 253–267. IEEE, 2013.

[168] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious ram. In *19th Annu. Network and Distributed System Security Symp. — NDSS 2012*. The Internet Soc., February 2012.

[169] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proc. 2013 ACM SIGSAC Conf. Comput. Commun. security*, pages 299–310. ACM, 2013.

[170] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li. Verifiable privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. *IEEE Transactions on Parallel and Distributed Systems*, 25(11):3025–3035, Nov 2014.

[171] Wenhai Sun, Bing Wang, Ning Cao, Ming Li, Wenjing Lou, Y Thomas Hou, and Hui Li. Verifiable privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking. *IEEE Trans. Parallel Distributed Syst.*, 25(11):3025–3035, 2014.

[172] Wenhai Sun, Ruide Zhang, Wenjing Lou, and Y Thomas Hou. Rearguard: Secure keyword search using trusted hardware. In *IEEE INFOCOM*, 2018.

[173] Shruti Tople, Yaoqi Jia, and Prateek Saxena. Pro-oram: Practical read-only oblivious {RAM}. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, 2020.

[174] Jonathan Trostle and Andy Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *International Conference on Information Security*, pages 114–128. Springer, 2010.

[175] B. Wang, M. Li, and L. Xiong. Fastgeo: Efficient geometric range queries on encrypted spatial data. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1–1, 2017.

[176] Boyang Wang, Yantian Hou, Ming Li, Haitao Wang, and Hui Li. Maple: Scalable multi-dimensional range search over encrypted cloud data with tree-based index. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 111–122, New York, NY, USA, 2014. ACM.

[177] Qian Wang, Meiqi He, Minxin Du, Sherman SM Chow, Russell WF Lai, and Qin Zou. Searchable encryption over feature-rich data. *IEEE Trans. Dependable Secure Computing*, 2016.

[178] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.

[179] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861. ACM, 2015.

[180] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 21–37, New York, NY, USA, 2017. Association for Computing Machinery.

[181] Xiao Shaun Wang, Yan Huang, TH Hubert Chan, Abhi Shelat, and Elaine Shi. Scoram: oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 191–202. ACM, 2014.

[182] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226. ACM, 2014.

[183] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: Base User-level ISA. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[184] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves.

[185] Lloyd R Welch and Elwyn R Berlekamp. Error correction for algebraic block codes, December 30 1986. US Patent 4,633,470.

[186] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 977–988. ACM, 2012.

[187] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[188] Andrew C Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science, 1982.*, pages 160–164. IEEE, 1982.

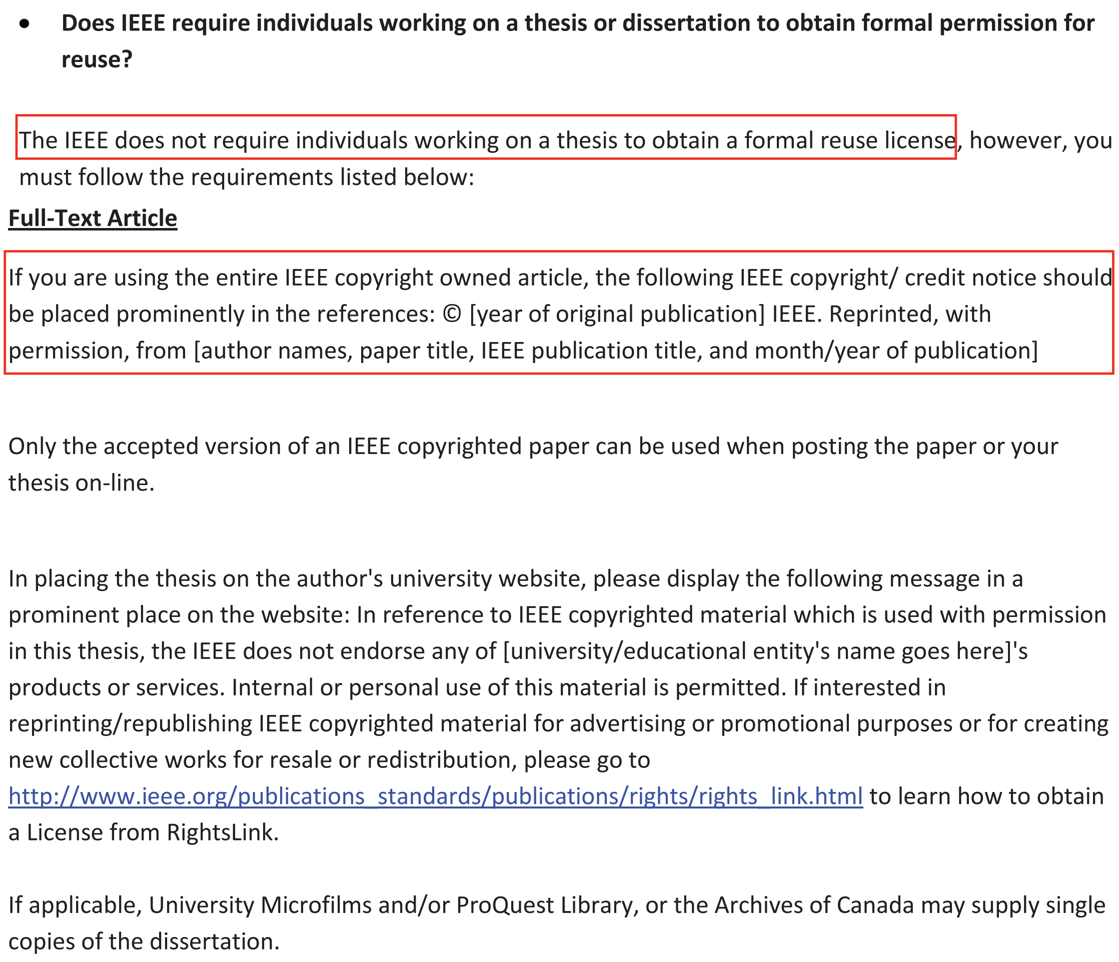
[189] Attila A. Yavuz and Jorge Guajardo. Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. In *Selected Areas in Cryptography – SAC 2015*, Lecture Notes in Computer Science. Springer International Publishing, August 2015.

[190] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security '16*, pages 707–720, Austin, TX, 2016.

Appendix A: Copyright Permissions

The permission below is for the reproduction of material in Chapter 3, Chapter 4 and Chapter 5 from the following publishers.

1. IEEE:



- **Does IEEE require individuals working on a thesis or dissertation to obtain formal permission for reuse?**

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you must follow the requirements listed below:

Full-Text Article

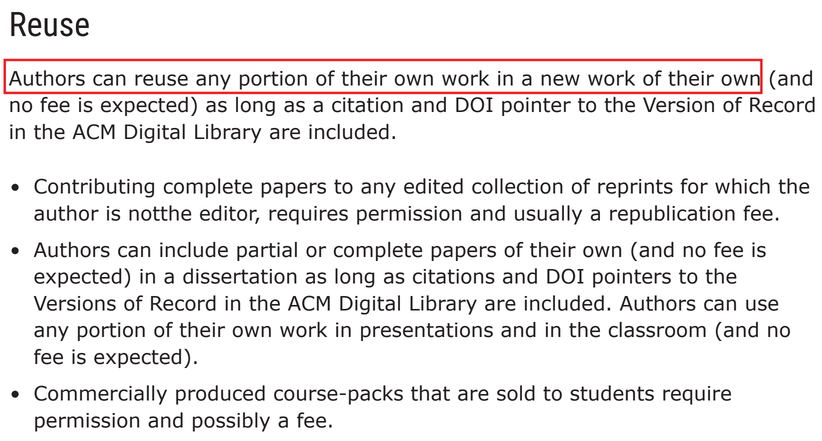
If you are using the entire IEEE copyright owned article, the following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]

Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.

In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

2. ACM:

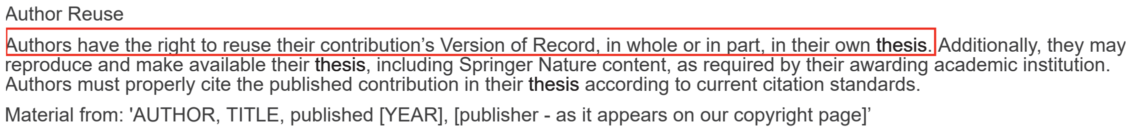


Reuse

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is notthe editor, requires permission and usually a republication fee.
- Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).
- Commercially produced course-packs that are sold to students require permission and possibly a fee.

3. Springer:

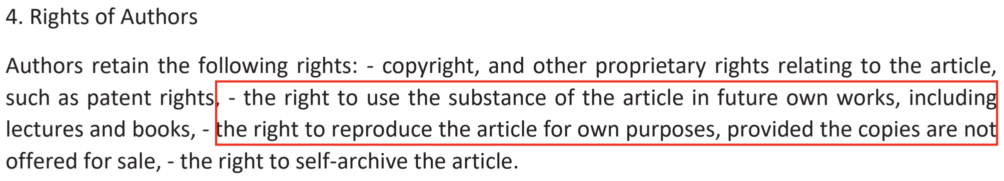


Author Reuse

Authors have the right to reuse their contribution's Version of Record, in whole or in part, in their own thesis. Additionally, they may reproduce and make available their thesis, including Springer Nature content, as required by their awarding academic institution. Authors must properly cite the published contribution in their thesis according to current citation standards.

Material from: 'AUTHOR, TITLE, published [YEAR], [publisher - as it appears on our copyright page]'

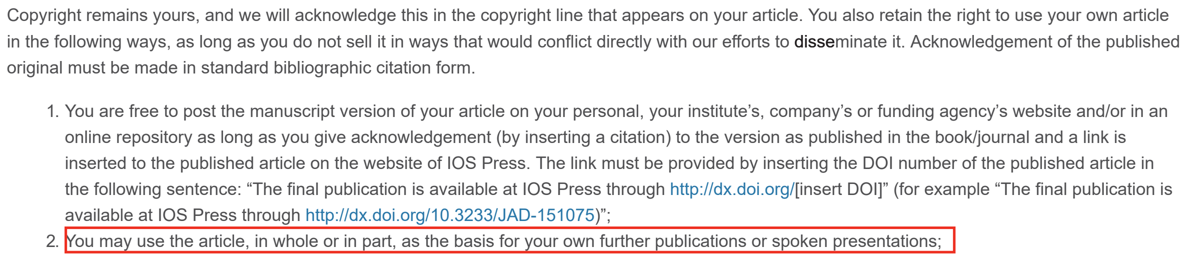
4. Sciendo:



4. Rights of Authors

Authors retain the following rights: - copyright, and other proprietary rights relating to the article, such as patent rights, - the right to use the substance of the article in future own works, including lectures and books, - the right to reproduce the article for own purposes, provided the copies are not offered for sale, - the right to self-archive the article.

5. IOS Press:



Copyright remains yours, and we will acknowledge this in the copyright line that appears on your article. You also retain the right to use your own article in the following ways, as long as you do not sell it in ways that would conflict directly with our efforts to disseminate it. Acknowledgement of the published original must be made in standard bibliographic citation form.

1. You are free to post the manuscript version of your article on your personal, your institute's, company's or funding agency's website and/or in an online repository as long as you give acknowledgement (by inserting a citation) to the version as published in the book/journal and a link is inserted to the published article on the website of IOS Press. The link must be provided by inserting the DOI number of the published article in the following sentence: "The final publication is available at IOS Press through http://dx.doi.org/[insert DOI]" (for example "The final publication is available at IOS Press through http://dx.doi.org/10.3233/JAD-151075)";
2. You may use the article, in whole or in part, as the basis for your own further publications or spoken presentations;

About the Author

Thang Hoang was born in Ho Chi Minh city, Vietnam. He was a PhD student at Oregon State University in 2015–2018 and then transferred to the University of South Florida in 2019. He received his MS degree in Computer Science from Chonnam National University (South Korea) in 2014, and his BS degree in Computer Science from the University of Science, VNU-HCMC (Vietnam) in 2010. His research interests include applied cryptography, data privacy, privacy-enhancing technologies and mobile security.