MAPLE: A Metadata-Hiding Policy-Controllable Encrypted Search Platform with Minimal Trust

Tung Le
Virginia Tech
Blacksburg, Virginia, USA
tungle@vt.edu

Thang Hoang
Virginia Tech
Blacksburg, Virginia, USA
thanghoang@vt.edu

ABSTRACT

Commodity encrypted storage platforms (e.g., IceDrive, pCloud) permit data store and sharing across multiple users while preserving data confidentiality. However, end-to-end encryption may not be sufficient since it only offers confidentiality when the data is at rest or in transit. Meanwhile, sensitive information can be leaked from metadata representing activities during data operations (e.g., query, processing). Recent encrypted search platforms such as DORY (OSDI'20) or DURASIFT (WPES'19) permit multi-user data query functionalities, while protecting metadata privacy. However, they either incur a high processing overhead or offer limited security/functionality, and require strong trust assumptions.

We propose MAPLE, a new metadata-hiding encrypted search platform that offers query functionalities (search, update) on the shared data across multiple users with complex policy controls. MAPLE protects metadata privacy all the time during query processing, while achieving significantly (asymptotically) lower processing overhead than state-of-the-art platforms. The core technique of MAPLE is the design of oblivious data structures for search index and access control coupled with secure computation techniques to enable efficient query processing with a minimal trust. We fully implemented MAPLE and evaluated its performance on commodity cloud (Amazon EC2) under real settings. Experimental results showed that MAPLE achieved a concrete performance comparable with its counterparts, while offering provably stronger security guarantees and more diverse functionalities.

KEYWORDS

oblivious access, access control, searchable encryption, encrypted database, multi-party computation

1 INTRODUCTION

The advent of cloud STorage-as-a-Service (STaaS) has brought significant benefits to individuals and businesses. Many commodity STaaS platforms have been deployed (e.g., Dropbox [1], Google Drive [3]), which permit not only data storage, but also sharing across multiple users, thereby improving data accessibility, flexibility and telecommuting. Despite their utility merits, STaaS platforms pose significant privacy concerns to users. Once the data is stored/shared on the cloud, it is unclear whether data will be misused or compromised. Since the shared data can be naturally sensitive (e.g., non-disclosure agreements, human subjects), it is mandatory to ensure its confidentiality and integrity all the time.

Several commercial encrypted storage and sharing platforms (e.g., Boxcryptor [4], Tresorit [5], Sync [2]) were proposed that employ standard security measures (e.g., AES, MAC) to enable confidentiality and integrity when the data is at-rest or in-transit. However, end-to-end encryption may not be sufficient to ensure robust confidentiality since sensitive information can still be leaked *during* operations on the encrypted data (e.g., query, computation). Specifically, an adversary can infer the data content by observing *metadata* indicating activities on the data. Such metadata includes, for example, what data is being accessed (access patterns); what data is being searched (search patterns); what data is being updated over time (update patterns); how much data involved in the users' queries (volume patterns), whom the data is shared with (identity/policy leakage). In the private communication context, metadata has been shown to be "densely interconnected, susceptible to re-identification, and enables highly sensitive inferences" [73]. A former general counsel at the US NSA has stated, "*metadata absolutely tells you everything about somebody's life. If you have enough metadata, you don't really need content*" [79].

Several attempts have been proposed to enhance the privacy and functionalities in remote data storage and sharing. One research line focuses on minimizing metadata leakage when sharing/accessing data [11, 27–29, 31, 33, 50, 57, 69, 70, 85]. However, these schemes offer limited functionalities (e.g., share and access only). Given that the shared data across multiple users is generally of large-scale, it is mandatory to offer query functionalities (i.e., search) to enhance information retrieval experience, while ensuring metadata privacy.

Another line of research focuses on enabling encrypted query functionalities over the encrypted data [20, 23, 39, 56, 62, 84]. However, they either leak significant metadata information [20], and/or only support private queries over private data, thereby not permitting sharing functionalities [56]. Many devastating attacks have shown that the adversary can recover the encrypted query and data contents by exploiting many types of metadata leakage ranging from search patterns [59, 67, 75] to access patterns [7, 59, 66] and volume patterns [15, 66] during query operations. In addition, the importance of oblivious update has also been emphasized since update patterns may also reveal sensitive information as mentioned in [12, 78]. These works concentrate on designing new efficient mechanisms to support hiding the pattern of write operations. There exists a few schemes that can offer a high level of metadata privacy, while permitting data sharing and query functionalities (e.g., [35, 37, 39]). Unfortunately, they suffer from a poor performance penalty [39], and/or require a strong assumption such as fully trusted clients [35, 39] and secure hardware [37].

Proceedings on Privacy Enhancing Technologies 2023(4), 184–203

https://doi.org/10.56553/popets-2023-0105

Table 1: Comparison of MAPLE with prior encrypted search systems.

Scheme	Security					Functionality		Complexity		Delay[‡]	
	MS	**MC**	**SP**	**UP**	**VP**	**MU**	**AC**	**Search**	**Update**	**Search** (s)	**Update** (s)
Standard SE [23]	(1,1)	✗	✗	✗	✗	✗	✗	$O(n)$	$O(w)$	0.11	0.03
DURASIFT [39]	✗	✗	✓	✗	✓	✓	✓	$O(NM)$	–	2.43	–
ODSE [56]	(t, N)	✗	✓	✓	✓	✗	✗	$O(NM)$	$O(\lambda M)$	1.80	0.22
DORY [35]	$(1, 2)$	✗	✓	✗	✓	✓	✗	$O(Nm)$	$O(m)$	0.23	0.10
Ours	$(N-1, N)$	✓	✓	✓	✓	✓	✓	$O(N \log m)$	$O(m \log N + m^2)$	0.59	6.17

- **MS**: Malicious server; **MC**: Malicious client; **SP**: Search pattern; **UP**: Update pattern; **VP**: Volumne pattern; **MU**: Multi-user; **AC**: Access control
- N: Total number of documents; M: Total number of unique keywords over *all* documents (keyword universe); m: Size of keyword representation *per* document; λ: Security parameter; n: Number of documents matched per keyword search; w: Number of keywords per updated document. In practice, $w < m \ll M$, $n \ll N$ and $m \ll N$.
- All ODSE, DURASIFT, DORY and MAPLE require distributed servers to offer sharing functionalities and/or improved security that may not be available in standard (single-server) SE. MAPLE offers fine-grained access control along with metadata privacy and integrity against malicious client(s) and server(s), all of which are not simultaneously achieved in prior works.

[‡] This result is obtained on the subset of enwiki dataset [8] with $\lambda = 128$, $N = 65536$, $m = 16384$, $M \approx 1.9M$ (see §7 for more details and results).

Given that there is a lack of multi-user encrypted search platforms with desirable functionalities and security properties, we ask the following question:

Can we design a new encrypted search platform that offers strong security (e.g., access control, metadata privacy) with critical functionalities (i.e., update, search), while achieving a practical performance for multi-user storage and sharing applications?

1.1 Our Results and Contributions

We answer the aforementioned question affirmatively with a design of MAPLE, a new encrypted search platform that offers high security (access control, metadata-privacy) and vital functionalities. MAPLE offers the following properties:

- **Document sharing with complex policy controls.** MAPLE permits document sharing across multiple users with fine-grained access control, in which the data owner can specify access permission (read/write/none) for each user to every single document.
- **Efficient keyword search on shared documents with access control.** MAPLE enables searching over the shared collection to retrieve documents that the users are authorized. The search complexity in MAPLE is efficient both asymptotically and experimentally. Specifically, the search in MAPLE is *logarithmic* to the (average) number of keywords per document, compared with linear either to the size of the keyword universe (e.g., [39, 56]), or to the number of keywords per document [35]. The asymptotic advantage permits MAPLE to offer a comparable concrete search delay over its counterparts, while offering higher security (hiding update patterns, access control). Compared with platforms offering the same security level, MAPLE outperforms by a few orders of magnitude.
- **Efficient dynamism.** MAPLE permits both the authorized users and the document owner to update the shared documents and the user permissions (i.e., granting/revocation), respectively, over the time. The update in MAPLE is only logarithmic to the number of documents, while offering obfuscated update patterns.
- **Complete metadata-hiding with minimal trust.** The most important advantage of MAPLE over all prior works is that MAPLE hides all metadata leakage when performing data operations including share, search, and update. MAPLE hides search patterns (keyword search frequency), update patterns (document/permission update frequency), volume patterns (how many documents/keywords in the search result/updated document), all of which have recently been shown vulnerable to practical attacks [7, 15, 66, 75]. To achieve all these security properties, MAPLE requires a minimal trust, in which $\ell - 1$ out of ℓ servers, as well as the users, can be untrusted and behave maliciously.
- **Technique: Oblivious table.** The core technique of MAPLE is the oblivious data structure for secure row/column operations on the table, by which the search index is instantiated. We refine the design of oblivious matrix in [53] with auxiliary components to enable efficient retrieval and recovery of the original row/column in the table. The oblivious table permits both keyword search and document update with a logarithmic cost, and thus, is more efficient than prior works that incur linear processing (e.g., [35, 39, 56]). The oblivious table is also used for a fine-grained access control that permits an efficient check/update of user permission with a logarithmic cost without leaking what permission is updated.

1.2 Technical Highlights

We rely on the blueprint design of state-of-the-art distributed encrypted search platforms (i.e., [35, 39, 56]) that instantiate the search index with a table for metadata privacy during query processing. We start by giving a brief overview of these approaches.

A brief overview of table-based search index. In table-based index, the searchable keywords are assigned to one dimension (e.g., columns), while the document identifiers are assigned to the other dimension of the table. With this arrangement, searching of a keyword is achieved by retrieving a column while document update is done by replacing a row of the table. All the table-based encrypted search platforms [35, 39, 56] employed Private Information Retrieval (PIR) techniques (e.g., [29, 43, 44]) to privately retrieve a column during search, thereby hiding search and volume patterns. Despite their merits, there exist some performance and security issues in these designs. First, early systems assigned each keyword in the *keyword universe* to a unique column (or row) of the table. This design causes the search index to be not only highly sparse, but also of large size when storing a large-scale document collection. For instance, in the collection with 5M+ documents (e.g., [8]), the size of keyword universe is more than 7M+, (and thus the index size can be in terabytes) while the majority of the dataset has less than 200 keywords per document. Later systems like DORY [35] addressed this sparsity by applying a Bloom filter for keyword representation per document, which reduces the index size at the cost of a small (yet acceptable) false positive rate in the search result.

Despite the elegant design of DORY, there are still some limitations. First, it relies on PIR techniques for keyword search, which

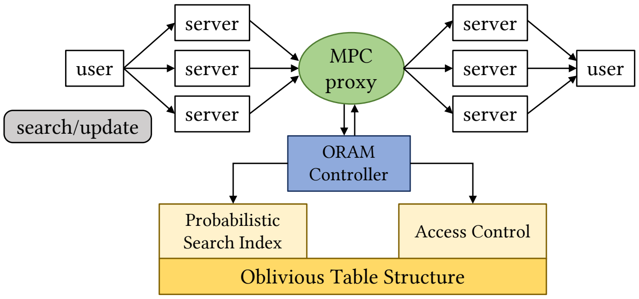


Figure 1: Key components of MAPLE and its high-level workflow.

incurs a linear processing overhead to the table size. Second, the document update in DORY leaks metadata information (i.e., update patterns), and thus may be vulnerable to statistical attacks [20]. Finally, it requires the clients that share the same set of documents to be fully trusted given that they maintain the same decryption key. When the adversary corrupts a client and a server, the update confidentiality of honest clients can be compromised. This attack cannot be prevented even when update operation is made oblivious as the adversary can simply decrypt the index before and after the update and compare the differences.

Idea 1: Oblivious table for search index representation. Our first idea is that instead of using PIR for private search, which incurs a linear processing cost, we design an oblivious data structure that permits private retrieval of a row (or column) of the table in a more asymptotically efficient manner than PIR approaches via oblivious access techniques such as Oblivious RAM [92]. Note that simply using ORAM on the original table structure only enables oblivious access to one dimension but not both. Our oblivious table permits both row and column accesses with overhead logarithmic to the number of rows and columns, respectively. We instantiate the search index with the oblivious table, which permits to conduct a private search or document update with a logarithmic overhead compared with linear as in PIR approaches.

Idea 2: Emulating oblivious table and access control via multi-party computation. Our second step is to enforce access control to enable data sharing. First, we delegate all oblivious table operations to the servers using Multi-Party Computation (MPC) techniques so that the user does not need to provide any secret state information. Second, we instantiate access control with another oblivious table that has the same number of rows with the index table (assuming documents are in row dimension). By ensuring that the row alignment of these tables is always consistent after every operation, we can enforce access control by using low-cost MPC operations (e.g., bit-wise AND) to filter out documents the user is not authorized to access during keyword search. Due to the characteristic of the index and access control tables, where their cell values are either 0, 1, or in 2 bits, we can emulate all the operations needed for keyword search/document update with boolean-friendly MPC. By using MPC protocols with dishonest majority (e.g., [95]), MAPLE requires only one server to be semi-honest while the rest (other servers and the users) can be malicious.

Putting everything together. By combining two ideas, we come up with the complete MAPLE system that offers keyword search, document/permission update functionalities with complete metadata hiding and sublinear processing overhead. We present the components and high-level workflow of our system in Figure 1. To search for a keyword, the user computes its Bloom filter, and requests the servers to emulate oblivious table accesses on several rows (or columns) indicated by the Bloom filter. The servers then emulate oblivious access on the access control table to obtain user permission on all documents, and finally filter out all documents that the user is not authorized to access using MPC techniques. To update a document, the user computes a new Bloom filter for the keyword representation of the updated document. The servers emulate oblivious access on the access control table to verify the user permission, and then emulate oblivious access on the index table to replace the old row with a new Bloom filter representation.

1.3 Related Work

Searchable encryption (SE). SE was first suggested by Song et al. [84], followed by a body of Symmetric SE (SSE) schemes proposed that offer encrypted search over encrypted data plus dynamic update via an encrypted index [21, 24, 42, 61, 62, 87, 88] (see [18, 77] for a comprehensive survey). Most SSE schemes only support a single user and leak metadata information such as search patterns, update patterns and history, access patterns, and volume patterns, all of which have been demonstrated vulnerable to many devastating leakage-abuse attacks [7, 15, 59, 66, 67, 99]. Some SSE schemes permit adjustable leakage and efficiency trade-off [36].

To improve functionality and efficiency, SE has been explored in other settings such as multi-user [64, 91] and multi-server [19, 56, 60, 65, 76]. However, most these schemes use a deterministic query encryption, and thus leak search and access patterns.

Oblivious access primitives. ORAM [47] and PIR [30] are two cryptographic primitives that can hide the access patterns. While generic ORAM hides both read and write patterns and PIR only hides read patterns, there are some special ORAM that only hide one type of operations (either read [90] or write only [16, 78]) and some PIR that hide write patterns [22]. Similar to SE, both PIR and ORAM were also explored in multi-user [13, 26, 31, 58, 71] and multi-server [22, 25, 30, 43, 45, 48, 52, 54] settings to improve the efficiency, resiliency, and functionality. All these techniques only support generic access, but not searchability.

Oblivious platforms. Some oblivious storage platforms employ ORAM and/or PIR primitives to hide metadata during data operations in data outsourcing [9, 27, 28, 68, 72, 80, 86, 97]. There are few platforms [35, 39, 82] that further enable query functionalities on top of data access while maintaining metadata privacy, some of which offer multi-user settings (e.g., [35, 39]).

Oblivious platforms with secure hardware. There is a line of research that employs trusted hardware (e.g., SGX [32]) to build practical and usable oblivious platforms with diverse functionalities [34, 38, 41, 51, 55, 74, 89]. These platforms require a strong security assumption on the hardware (e.g., isolation, tamper-free, side-channel resistance).

2 PRELIMINARIES

Notation. $||$ denotes the concatenation operator. We denote $[N]$ as a set $\{1, \ldots, N\}$. $x \overset{\$}{\leftarrow} [N]$ means x is selected uniformly at random from the set $\{1, \ldots, N\}$. We denote $H : \{0,1\}^* \rightarrow [N]$ as a mapping function. Given a table $\mathbf{T}$, $\mathbf{T}[i,*]$ and $\mathbf{T}[*,j]$ denote accessing the row i and column j of $\mathbf{T}$, respectively. $\mathbf{T}[i,j]$ denotes accessing the cell indexed at row i and column j. $\mathbf{T}^\top$ denotes the transpose of $\mathbf{T}$. Let $\mathbb{F}$ be a finite field, λ be the security parameter, and $\mathsf{negl}(\cdot)$ be a negligible function. We denote $A \overset{c}{\approx} B$ as the computational indistinguishability between two distributions A and B.

2.1 Secure Multi-Party Computation

Muti-Party Computation (MPC) permits ℓ parties to jointly evaluate a function $f(x_1, \ldots, x_\ell) \rightarrow (y_1, \ldots, y_\ell)$, in which each party $\mathcal{P}_i$ learns its output y_i without leaking its private input x_i to the other parties. In this paper, although we use MPC as a black box, we provide basic background on MPC along with its useful properties. Prior to joint evaluation, the parties can share their secret input x with each other via XOR ($x = \oplus_i x_i$), addition ($x = \sum_i x_i$), or in special forms (e.g., Yao sharings [98]).

Authentication. To achieve integrity against malicious adversaries, a Message Authentication Code (MAC) can be added. We recall the BDOZ-style MAC [14] that is mostly used in garbling protocols with constant-round (*e.g.*, [94, 95]). An authenticated share of a secret x to party $\mathcal{P}_i$ is $\langle x \rangle_i = (x_i, \{m_{i,j}\}_{i \neq j}, \{k_{j,i}\}_{i \neq j})$, where $x = \oplus_i x_i$, $k_{j,i}$ is the MAC key that $\mathcal{P}_i$ uses to authenticate the share b_j of party $\mathcal{P}_j$, and $m_{i,j} = k_{j,i} \oplus b_i \cdot \Delta_j$ is the MAC of the share b_i authenticated by party $\mathcal{P}_j$ under $\mathcal{P}_j$'s global MAC key $\Delta_j \in \{0,1\}^\lambda$.

Secure operations. All the linear computations (*e.g.*, addition, scalar multiplication) on the authenticated shares can be performed locally. We omit the subscript to denote the share of the secret being computed in MPC in general. Specifically, $\langle x + y \rangle = \langle x \rangle + \langle y \rangle$ and $\langle c \cdot x \rangle = c \cdot \langle x \rangle$, where $c \in \mathbb{F}$ is a public scalar and $+, \cdot$ denote the addition and multiplication over $\mathbb{F}$. We use $\boxplus$ to denote the secure addition protocol between authenticated shares. For multiplication between shares, it can be achieved with the aid of pre-processing that generates correlated randomness (e.g., Beaver multiplication triples $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ such that $c = a \cdot b$) in the offline phase. We use $\boxtimes$ to denote such a secure multiplication protocol as $\langle z \rangle = \langle x \rangle \boxtimes \langle y \rangle$ where $z = x \times y$, or $\langle \mathbf{z} \rangle = \langle \mathbf{x} \rangle \boxtimes \langle \mathbf{y} \rangle$ (entry-wise multiplication if $\mathbf{x}$ and $\mathbf{y}$ are vectors).

In this paper, we make use of MPC protocols for boolean operations including secure comparison and selection as follows.

- $\underline{\langle b \rangle \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{cmp}(\langle x_1 \rangle, \langle x_2 \rangle)}$: Given two authenticated shares $\langle x_1 \rangle$, $\langle x_2 \rangle$, it outputs an authenticated share bit $\langle b \rangle$ such that $b = 1$ if $x_1 = x_2$; otherwise $b = 0$ if $x_1 \neq x_2$.
- $\underline{\langle y \rangle \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{sel}(\langle b \rangle, \langle x_1 \rangle, \langle x_2 \rangle)}$: Given an authenticated share of a bit $\langle b \rangle$ and two authenticated shares of values $\langle x_1 \rangle$, $\langle x_2 \rangle$, it outputs an authenticated share $\langle y \rangle$ such that $y = x_1$ if $b = 0$; otherwise $y = x_2$ if $b = 1$.

We present how to implement these functions in §D.2. Finally, we extend the notation to denote the (authenticated) share of a vector. Given $\mathbf{v} = (v_1, \ldots, v_n)$, $\langle \mathbf{v} \rangle = (\langle v \rangle_1, \ldots, \langle v_n \rangle)$ is the (authenticated) share of $\mathbf{v}$, where each of its elements is secret-shared.

2.2 Oblivious RAM

ORAM [47] permits a client to privately access data in an untrusted memory without leaking the access pattern. We recall the Tree-ORAM paradigm by Shi et al. [83], which consists of the following PPT algorithms.

- $\underline{(st, \mathcal{M}) \leftarrow \mathsf{TreeORAM.Setup}(1^\lambda, \mathsf{DB}, N)}$: Given a security parameter λ, a database DB and its size N, it outputs a client state st and a server state $\mathcal{M}$.
- $\underline{(B, st', \mathcal{M}') \leftarrow \mathsf{TreeORAM.Access}(\mathsf{op}, \mathsf{id}, B', st, \mathcal{M})}$: Given an access operation $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}, \mathsf{update}\}$[1], a block identifier id, a data block to be written/updated B' (if $\mathsf{op} = \mathsf{update}$, B' contains only the portion of data to be updated for id), a client state st, a server state $\mathcal{M}$, it outputs the requested block B, the updated client state st' and the updated server state $\mathcal{M}'$.

In Tree-based ORAM, the database is organized into a binary tree structure maintained by the server as its state $\mathcal{M}$. Individual blocks in the database are located at random locations in the binary tree. This requires the client to maintain the state information st in the form of position map (pm) to keep track of the block locations in the tree, i.e., $\mathsf{pm}[\mathsf{bid}] \leftarrow \mathsf{pid}$, where bid and pid are the block identifier and the identifier of the path where the block resides, respectively. To remove this need, pm can be stored in smaller ORAM structures using recursion technique (see [83] for more details).

3 MODEL

System model. Our system consists of a document owner, t data users, and ℓ servers. The document owner owns a document collection DB and would like to share it with t users $\mathcal{U}_i$ with different access privileges. We consider a single owner for ease of presentation. It is easy to extend into multi-owner by reserving a dedicated space in the shared DB for each data owner. We aim to enable keyword search and document update functionalities for t users over the shared document collection stored in ℓ servers. Generally speaking, our scheme consists of the following PPT algorithms:

- $\underline{\mathcal{M} \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{DB}, \mathcal{L}, M, N, t)}$: Given a security parameter λ, a database DB, an access control list $\mathcal{L}$ where $\mathcal{L}(u,d) \in \{\mathsf{r}, \mathsf{w}, \perp\}$ contains permission of user u on document d, a bound on the number of keywords M, number of documents N, and number of users t, it returns a server state $\mathcal{M}$.
- $\underline{\mathcal{J} \leftarrow \mathsf{KSearch}(w, \mathsf{uid}, \mathcal{M})}$: Given a keyword w, a user identifier uid, and a server state $\mathcal{M}$, it outputs document identifiers $\mathcal{J} = (\mathsf{fid}_1, \ldots, \mathsf{fid}_n)$ that w appears in and uid is authorized to access.
- $\underline{\mathcal{M}' \leftarrow \mathsf{DUpdate}(\mathsf{fid}, \mathsf{uid}, \mathbf{w}, \mathcal{M})}$: Given a document identifier fid, a user identifier uid, a list of updated keywords $\mathbf{w}$, and a server state $\mathcal{M}$, it outputs a new server state $\mathcal{M}'$.
- $\underline{\mathcal{M}' \leftarrow \mathsf{PUpdate}(\mathsf{fid}, \mathsf{uid}, p, \mathcal{M})}$: Given a document identifier fid, a user identifier uid, a permission attribute p, and a server state $\mathcal{M}$, it outputs a new server state $\mathcal{M}'$.

Threat model. In our system, the data owner is fully trusted, while the users and the servers are untrusted. We also assume that the parties communicate with each other in a synchronous manner and the communication channels between end-to-end parties are

[1] Standard ORAM offers read and write as main operations. We introduce update for easy presentation of implementing ORAM with MPC in our protocol for some functionalities. In the client-server setting, update can be implemented as ORAM write.

pair-wise secure (*e.g.*, via SSL). We assume a malicious adversary that can corrupt up to $\ell' < \ell$ servers and $t' < t$ users in the system. Specifically, the adversary is not only curious about the queries of the honest users, but also deviates from the protocol. For example, the adversary is curious about (*i*) which keyword is being searched and/or the frequency (*i.e.*, search patterns); (*ii*) what document is being updated and/or the frequency (*i.e.*, update patterns); (*ii*) how many documents (or keywords) appear in the search/update query (*i.e.*, volume patterns); (*iv*) whether the user has an access privilege to a particular document. The adversary can also inject malicious inputs into the protocol to compromise the integrity of the system.

Security model. We aim to achieve confidentiality, integrity, and metadata privacy against a malicious adversary that corrupts up to $\ell' < \ell$ servers and an arbitrary number of users. Let $\mathcal{A}$ be the adversary that controls all the corrupted parties and $\mathcal{U}$ be the honest user who has access to a list of documents $\mathcal{D}'$, respectively.

- **Search obliviousness:** $\mathcal{A}$ cannot infer which keyword has been searched by $\mathcal{U}$ and whether or not $\mathcal{U}$ has searched for the same or different keywords with another honest user. $\mathcal{A}$ also cannot infer which documents (and how many) that match the search query and which documents (and how many) the user $\mathcal{U}$ is authorized to search. If $\mathcal{A}$ is the user that performs a keyword search, $\mathcal{A}$ cannot infer which documents among the list of unauthorized documents that match its query.
- **Update obliviousness:** $\mathcal{A}$ cannot infer which document has been updated by $\mathcal{U}$ and whether or not $\mathcal{U}$ has updated the same or different document with another honest user. $\mathcal{A}$ also cannot infer which keywords (or whether the same or different keywords) are added/deleted in each update operation, and whether or not $\mathcal{U}$ is authorized to perform update.

These obliviousness properties indicate that in each search/update operation, no information is leaked apart from the user identity who performs the operation. Thus, they also imply the forward and backward privacy [21] in the standard encrypted search. We present the formal definition of forward/backward privacy and search/update obliviousness via quantified leakage functions in Appendix B. In addition to obliviousness properties, our scheme aims to achieve integrity and anonymity properties as follows.

- **Search integrity:** $\mathcal{A}$ cannot tamper with the result of the search operation performed by the honest user $\mathcal{U}$.
- **Update integrity:** $\mathcal{A}$ cannot tamper with the update query of the honest user $\mathcal{U}$ (e.g., by adding/deleting some keywords on behalf of $\mathcal{U}$). If $\mathcal{A}$ is the user that performs a document update, $\mathcal{A}$ cannot update the document which is not authorized.
- **Anonymity**: $\mathcal{A}$ cannot determine the identity of the honest user $\mathcal{U}$ who performs the search/update request, given that $\mathcal{U}$ uses an anonymous communication channel to interact with the servers (e.g., Tor network).

We present an ideal functionality $\mathcal{F}_{\mathsf{MSrch}}$ that captures the main functionalities offered by our system in Figure 17 (see Appendix due to space constraint). We define the security of our system using the ideal/real-world simulation paradigm as follows.

Definition 1 ($\mathcal{L}$-simulation based security: privacy + verifiability). *Let* $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Upd}}\}$ *be leakage functions during setup, keyword search, and document update operations. A protocol* Π *is said securely realize* $\mathcal{F}_{\mathsf{MSrch}}$ *in the presence of a malicious adversary if, for every PPT adversary* $\mathcal{A}$ *in the real world, there exists a PPT simulator* $\mathcal{S}$ *in the ideal world such that*

$$|\Pr[\textsc{Real}_{\Pi,\mathcal{A}}(\lambda) = 1] - \Pr[\textsc{Ideal}_{\mathcal{F}_{\mathsf{MSrch}},\mathcal{S},\mathcal{L}}(\lambda)]| \leq \mathsf{negl}(\lambda)$$

Metadata privacy on document retrieval/update. In this paper, we only focus on the search index privacy. From the system-wide viewpoint, it is also necessary to seal leakages in the document retrieval/update phase that occurs after processing the search index as discussed in [49]. Ensuring privacy for document access can be achieved independently with existing ORAM/PIR-based file systems (e.g., [27, 28, 33, 51]). These techniques can be integrated orthogonally into our scheme to achieve system-wide security.

4 OBLIVIOUS TABLE

We present OTAB, an oblivious structure for efficient row and column access in a table without leaking row/column access patterns.

Definition 2 (Oblivious Data Structure [96]). *Let* $\mathsf{addr}_D(\vec{\mathsf{op}})$ *be the physical addresses generated during a sequence of operations* $\vec{\mathsf{op}} = ((\mathsf{op}_1, \mathsf{arg}_1), \ldots, (\mathsf{op}_L, \mathsf{arg}_L))$ *on data structure D. D is said to be* oblivious*, if there exists a PPT simulator* $\mathcal{S}$*, such that for any polynomial-length sequence of operations* $\vec{\mathsf{op}}$*, we have that*

$$\mathsf{addr}_D(\vec{\mathsf{op}}) \overset{c}{\approx} \mathcal{S}(L).$$

Our OTAB consists of the following PPT algorithms:

- $\underline{(st, \mathcal{M}) \leftarrow \mathsf{OTAB.Setup}(1^\lambda, \mathbf{T}', N, M)}$: Given a security parameter λ, an $N \times M$ table $\mathbf{T}'$, it returns a client state st and a server state $\mathcal{M}$.
- $\underline{(\mathbf{v}, st', \mathcal{M}') \leftarrow \mathsf{OTAB.Access}(\mathsf{op}, \mathsf{id}, \mathbf{v}', st, \mathcal{M})}$: Given an operation type $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}, \mathsf{update}\}$, a row/column identifier id, an updated row/column $\mathbf{v}'$, a client state st, and a server state $\mathcal{M}$, it outputs a row/column $\mathbf{v}$ in the oblivious table, an updated client state st', and an updated server state $\mathcal{M}'$.
- $\underline{\mathbf{v} \leftarrow \mathsf{OTAB.Reconstruct}(\mathbf{v}', st)}$: Given a row/column $\mathbf{v}'$ in the oblivious table and a state st, it outputs original row/column $\mathbf{v}$.

Figure 2 presents the setup and access protocols of OTAB in detail. The idea is to treat each dimension of the table as a "logical" ORAM structure, where each row (or column) is treated as an ORAM block. With this representation, OTAB permits the back-end ORAM scheme to be executed on its row (or column) dimension to access the desired row (or column). Generally speaking, given a table $\mathbf{T}'$ of size $N \times M$ as an input, we create a bigger table $\mathbf{T}$ of size $z \cdot N \times z \cdot M$, where z is the expansion factor of the back-end ORAM, which contains all data of $\mathbf{T}'$. Due to the back-end ORAM, all rows and columns of $\mathbf{T}$ need to be shuffled in such a way that each of them resides at a random path as indicated by the logical back-end ORAM structure. We denote such row and column permutations as π_r and π_c, respectively (line 3). Notice that such permutations do not need to be keyed since we store such random assignments in the position map components. Specifically, we use two position maps, pm_r and pm_c, to keep track of the logical path and the exact location of the rows and columns of $\mathbf{T}'$ in $\mathbf{T}$, respectively (line 5). We also maintain two components $\mathbf{h}_\mathsf{r}$ and $\mathbf{h}_\mathsf{c}$, which can be thought of as the row and column headers of $\mathbf{T}$, respectively (line 6). These components store the row/column indices of the original table $\mathbf{T}'$

```
• OTAB.Setup($1^\lambda, \mathbf{T}', N, M$):
1: $\mathbf{T} \leftarrow 0^{(z \cdot N+1)\times(z \cdot M+1)}$, $\mathbf{h}_\mathrm{r} \leftarrow 0^{z \cdot N}$, $\mathbf{h}_\mathrm{c} \leftarrow 0^{z \cdot M}$,
2: for $i = 1$ to $N$ and $j = 1$ to $M$ do
3:     $x_i \leftarrow \pi_\mathrm{r}(i)$, $y_i \leftarrow \pi_\mathrm{c}(j)$          ▷ $\pi_\mathrm{r}/\pi_\mathrm{c}$ is row/col permutation
4:     $\mathbf{T}[x_i, y_j] \leftarrow \mathbf{T}'[i, j]$
5:     $\mathrm{pm}_\mathrm{r}[i].\mathrm{id} \leftarrow \pi_\mathrm{r}(i)$, $\mathrm{pm}_\mathrm{c}[j].\mathrm{id} \leftarrow \pi_\mathrm{c}(j)$
6:     $\mathbf{h}_\mathrm{r}[x_i] \leftarrow i$, $\mathbf{h}_\mathrm{c}[y_j] \leftarrow j$
7:     $\mathbf{T}[0, 1 \ldots zM] \leftarrow \mathbf{h}_\mathrm{c}$, $\mathbf{T}[1 \ldots zN, 0] \leftarrow \mathbf{h}_\mathrm{r}$
8: return $(st, \mathcal{M})$, where $st = (\mathrm{pm}_\mathrm{r}, \mathrm{pm}_\mathrm{c}, \mathbf{h}_\mathrm{r}, \mathbf{h}_\mathrm{c})$ and $\mathcal{M} = \mathbf{T}$

• OTAB.Access($\mathrm{op}, \mathrm{id}, \mathbf{v}', st, \mathcal{M}$):
9: Parse $st = (\mathrm{pm}_\mathrm{r}, \mathrm{pm}_\mathrm{c}, \mathbf{h}_\mathrm{r}, \mathbf{h}_\mathrm{c})$, $\mathcal{M} = \mathbf{T}$
10: $Y \leftarrow 0$, $\mathrm{DB} \leftarrow \perp$, $\mathbf{T}' \leftarrow \mathbf{T}$
11: if op is a row access then
12:     $\mathrm{pm} \leftarrow \mathrm{pm}_\mathrm{r}$, $Y \leftarrow M$, and $\mathrm{DB} \leftarrow (\mathbf{T}[1, *], \ldots, \mathbf{T}[z \cdot N, *])$
13: else if op is a column access then
14:     $\mathrm{pm} \leftarrow \mathrm{pm}_\mathrm{c}$, $Y \leftarrow N$, and $\mathrm{DB} \leftarrow (\mathbf{T}[*, 1], \ldots, \mathbf{T}[*, z \cdot M])$
15: $\hat{\mathbf{v}} \leftarrow 0^{z \cdot Y}$
16: for $i = 1$ to $Y$ do
17:     $\hat{\mathbf{v}}[x_i] \leftarrow \mathbf{v}'[i]$ where $x_i \leftarrow \mathrm{pm}[i].\mathrm{id}$
18: $(\mathbf{v}, \mathrm{pm}', \mathrm{DB}') \leftarrow \mathrm{TreeORAM.Access}(\mathrm{op}, \mathrm{id}, \hat{\mathbf{v}}, \mathrm{pm}, \mathrm{DB})$
19: if op is a row access then
20:     $\mathrm{pm}'_\mathrm{r} \leftarrow \mathrm{pm}'$, $\mathrm{pm}'_\mathrm{c} \leftarrow \mathrm{pm}_\mathrm{c}$
21:     $(\mathbf{T}'[1, *], \ldots, \mathbf{T}'[z \cdot N, *]) \leftarrow \mathrm{DB}'$
22:     $\mathbf{h}'_\mathrm{r} \leftarrow \mathbf{T}'[1 \ldots zN, 0]$, $\mathbf{h}'_\mathrm{c} \leftarrow \mathbf{h}_\mathrm{c}$
23: else if op is a column access then
24:     $\mathrm{pm}'_\mathrm{c} \leftarrow \mathrm{pm}'$, $\mathrm{pm}'_\mathrm{r} \leftarrow \mathrm{pm}_\mathrm{r}$
25:     $(\mathbf{T}'[*, 1], \ldots, \mathbf{T}'[*, z \cdot M]) \leftarrow \mathrm{DB}'$
26:     $\mathbf{h}'_\mathrm{c} \leftarrow \mathbf{T}'[0, 1 \ldots zM]$, $\mathbf{h}'_\mathrm{r} \leftarrow \mathbf{h}_\mathrm{r}$
27: return $(\mathbf{v}, st', \mathcal{M}')$, where $st' = (\mathrm{pm}'_\mathrm{r}, \mathrm{pm}'_\mathrm{c}, \mathbf{h}'_\mathrm{r}, \mathbf{h}'_\mathrm{c})$, $\mathcal{M}' = \mathbf{T}'$
• OTAB.Reconstruct($\mathbf{v}', st$):
28: Parse $st = (\mathrm{pm}_\mathrm{r}, \mathrm{pm}_\mathrm{c}, \mathbf{h}_\mathrm{r}, \mathbf{h}_\mathrm{c})$
29: if $\mathbf{v}'$ is a row in OTAB then
30:     $\mathbf{h} \leftarrow \mathbf{h}_\mathrm{c}$, $Y \leftarrow M$
31: else if $\mathbf{v}'$ is a column in OTAB then
32:     $\mathbf{h} \leftarrow \mathbf{h}_\mathrm{r}$, $Y \leftarrow N$
33: for $i = 0$ to $z \cdot Y$ do
34:     $\mathbf{v}[y_i] \leftarrow \mathbf{v}'[i]$ where $y_i \leftarrow \mathbf{h}[i]$
35: return $\mathbf{v}$
```

Figure 2: Oblivious table protocols.

in **T**, which will be used to reconstruct the original row/column of **T′** from the retrieved row/column of **T′**. As the elements in these headers will be shuffled due to underlying Tree-ORAM operations, for simplicity and ease of presentation, we treat them as part of **T**, where they can be stored at the index 0 of **T** (line 7). In general, given that OTAB.Setup generates necessary components (two position maps, row/column headers) for later row/column access using the back-end ORAM protocol, it can be considered as the extended TreeORAM.Setup procedure for oblivious table access.

The access operation in OTAB works by arranging the underlying Tree-ORAM components according to which dimension of the table is being accessed (lines 11–14). Each time ORAM is executed on the row/column dimension of OTAB, some of the rows/columns will be shuffled. Thus, when updating a row/column of the original **T**, we need to create a corresponding row/column of the oblivious **T′**, where the data in the original row/column is (re)ordered according to the current state of **T′** (lines 15–17). Once the access procedure by the underlying Tree-ORAM is finished, the client and server update their state information accordingly (lines 19–26). Finally, we may need to reconstruct the original row/column from the one retrieved from the oblivious table by rearranging the data to the original order. This can be done with the Reconstruct algorithm, where the rearrangement is realized using the row/column header (lines 33–34).

In summary, the consistency of row/column access in OTAB is maintained via 4 auxiliary components: row/column headers and row/column position maps. The position maps keep track of locations of columns/rows of the original table when they are moved around in the OTAB structure due to Tree-ORAM operations. Meanwhile, the headers permit the client to recover the original column/row from OTAB or to create an updated column/row that is consistent with the current OTAB state. We present a toy example of column/row access in OTAB with consistency in Appendix A (Figure 12).

Complexity. The cost of OTAB depends on the back-end ORAM being used. OTAB adds a small factor of constant z to the complexity since both the rows and columns are expanded z times. Assume Circuit-ORAM [92] is the back-end ORAM, the complexity of row and column access by OTAB is $\mathcal{O}(z \cdot M \log(z \cdot N)) = \mathcal{O}(M \log N)$ and $\mathcal{O}(z \cdot N \log(z \cdot M)) = \mathcal{O}(N \log M)$, respectively, where N and M are the number of rows and columns of the original table.

5 OUR MAPLE PLATFORM

We give an overview and then present our platform in detail.

5.1 Overview

We first present the data structures in our scheme. We then present a strawman construction to illustrate our main idea.

Data structures. MAPLE consists of two main structures: an index (IDX) for keyword search and an access control (AC) for document-user permission management. We instantiate both IDX and AC with tables. WLOG, we assume the keywords and document identifiers are assigned to the columns and rows in IDX, respectively. In AC, the user and document identifiers are in the columns and rows, respectively. For instance, IDX$[i, j] = 1$ means the keyword indexed at column j appears in the document indexed at row i. AC$[i, j] =$ r/w means the user indexed at column j is permitted to read/write the document indexed at row i. For simplicity, we assume that the document identifiers fid and user identifiers uid correspond to their indices in IDX and AC, respectively, i.e., fid $\in [N]$ and uid $\in [t]$. For keywords, we use a public mapping function H that encodes each keyword to a unique binary string (e.g., ASCII) and maps its binary representation to a unique column identifier in IDX as $H : \{0, 1\}^* \rightarrow [M]$, where M is the number of columns.

We instantiate IDX and AC with tables due to the following security and efficiency advantages. First, the table hides the size patterns of the search results, which may lead to volume leakage-abuse attacks [7, 15, 66]. Second, the table permits both search and update operations to be performed in two separate dimensions in a synchronous manner. We are aware that the table may incur high storage and computation overhead when the database is large. Therefore, we will design various strategies to mitigate this limitation.

Strawman proposal. We first present a strawman scheme to demonstrate the intuition of our proposed design. Instead of relying on PIR-like techniques as prior works that incur linear processing, we use ORAM-based approach by instantiating IDX and AC with oblivious table. To search for a keyword w with a user uid, we first determine the (column) index of w in IDX as cid $\leftarrow H(w)$. We then perform oblivious access over the column dimension of IDX to obtain the column cid representing the list of document identifiers that w appears in. Next, we perform oblivious access over the column dimension AC to obtain the access permission of uid on every document, thereby filtering out some identifiers in cid that uid is not authorized to access before sending the final result to the user uid. To update a document with an identifier fid for a user uid, we create a row $\mathbf{u}$ that represents all keywords that appear in fid. We then perform an oblivious access over the row dimension of AC to check if uid is authorized to update fid (i.e., $\mathsf{AC}[\mathsf{uid}, \mathsf{fid}] \overset{?}{=}$ 'w'), and perform an oblivious access on IDX to to replace the old row as $\mathsf{IDX}[\mathsf{fid}, *] \leftarrow \mathbf{u}$.

While the strawman scheme seems to work at the first glance, there are challenges in realizing the idea in a secure and efficient manner. First, the size of IDX is $O(N \cdot M)$, where N, M are the total number of documents and unique keywords in the entire database, respectively. Given the high sparsity of keywords in nature, it is likely that M will grow drastically when N increases, thereby incurring the high complexity. For instance, in the Wikipedia corpus [8] with $N \approx$ 5M documents, the size of keyword universe is approximately $M \approx$ 7M, and thus the storage cost can grow up to several terabytes. Moreover, each search and update operation incurs $O(N \cdot \log M)$ and $O(M \cdot \log N)$ time complexity, respectively, due to OTAB. Such costs may be high in practice if M is large.

Another challenge is the emulation of oblivious table in the multi-user setting, where the adversary can be present at both user and server sides. For instance, prior platforms prevent malicious server via authentication [35]. However, this approach assumes that the trusted user maintains a private MAC key, which is not possible in the multi-user setting, where some of the users can be adversarial and therefore, are untrusted. Furthermore, our oblivious table in §4 also requires the trusted user to maintain some secret states (e.g., permutations, position maps), which is not feasible as the users in the multi-user setting are untrusted. In the following, we present techniques to address these challenges efficiently.

5.2 Reducing Index Size

To reduce the index size and processing time, we employ a Bloom filter (BF) [17] to represent the keyword set per document. Let $\mathsf{BF} = (\mathsf{Init}, \mathsf{Gen}, \mathsf{Verify})$ be the BF for a set $\mathcal{S}$ defined as follows.

- $\underline{(H_1, \ldots, H_K) \leftarrow \mathsf{BF.Init}(m, K)}$: Given two parameters m and K, it outputs K mappings $H_i : \mathcal{S} \to [m]$.
- $\underline{\mathbf{u} \leftarrow \mathsf{BF.Gen}(\mathcal{S})}$: It takes a set $\mathcal{S}$ as input. It initializes a vector $\mathbf{u} = 0^m$. For each $s \in \mathcal{S}$, it computes and sets $\mathbf{u}[H_i(s)] \leftarrow 1$ for all $i \in [K]$. It outputs $\mathbf{u}$ as the compressed representation of $\mathcal{S}$.
- $\underline{\{0,1\} \leftarrow \mathsf{BF.Verify}(\mathbf{u}, s)}$: It takes an element s as input. It computes and outputs a bit $b \leftarrow \mathbf{u}[H_1(s)] \wedge \cdots \wedge \mathbf{u}[H_K(s)]$, in which $b = 0$ if $s \notin \mathcal{S}$, otherwise $b = 1$.

For each document, we compute a BF to generate a compressed representation for the keywords in it. We assume each document has at most q distinct keywords and the compressed representation is of length m. We select m, K to achieve a low False Positive (FP) rate determined as $\Pr[\mathrm{FP}] = \left(1 - (1 - 1/m)^{qK}\right)^K$.

The search procedure for a keyword w now proceeds as follows.

- We compute K column identifiers by BF $\mathsf{cid}_i = H_i(w)$.
- For each cid_i, we execute $\hat{\mathbf{v}}_i \leftarrow \mathsf{OMAT.Access}(\mathsf{read}, \mathsf{cid}_i, \perp, \mathsf{IDX})$ and $\mathbf{v}' \leftarrow \mathsf{OMAT.Access}(\mathsf{read}, \mathsf{uid}, \perp, \mathsf{AC})$. Finally, we perform the bitwise AND of $K+1$ columns as $\mathbf{v} \leftarrow \hat{\mathbf{v}}_1 \wedge \hat{\mathbf{v}}_2 \wedge \cdots \wedge \hat{\mathbf{v}}_K \wedge \mathbf{v}'$ and return $\mathbf{v}$ to the user.

With BF, we reduce the IDX size from $O(N \cdot M)$ to $O(N \cdot m)$, where $m \ll M$. Note that due to the smaller index size, search and update will also be more efficient.

5.3 Emulating Oblivious Table via MPC Proxy

In this section, we present how to emulate oblivious table to enable privacy-preserving keyword search and document update efficiently in the multi-user setting.

MPC proxy. In MAPLE, we employ ℓ servers that together form a so-called *MPC proxy* to process keyword search (or document update) requests from the users. Specifically, the servers will jointly execute MPC protocols to emulate OTAB access on the search index IDX and the access control AC for search/update operations without leaking the search/update/volume patterns. As we will show that emulating OTAB only incurs boolean operations, it suffices for the MPC proxy to employ boolean-friendly MPC protocols. Therefore, we opt to employ Authenticated Garbled Circuit in [95] as it offers security with dishonest majority with fast online processing and privacy degree scalability (w.r.t the number of servers).

5.3.1 *Authenticated I/O.*

In MAPLE, the user sends a search (or update) query to the MPC proxy and receives a corresponding output. Given that the user does not participate in the secure computation, we need to ensure the user shares correct input to the individual servers in the MPC proxy, while receiving trustworthy output. We employ the authenticated input and output protocols in [27] with the following interfaces (see §D.1 for details).

- $\underline{(\langle v\rangle_1, \ldots, \langle v\rangle_n) \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Input}(v_1, \ldots, v_n)}$: Given n user secrets, it outputs the corresponding authenticated shares to MPC proxy.
- $\underline{(v_1, \ldots, v_n) \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Output}(\langle v_1\rangle, \ldots, \langle v_n\rangle)}$: Given n shares from MPC proxy, it outputs the corresponding secrets to the user.

5.3.2 *OTAB Emulation.*

We present how the OTAB access is emulated with the MPC proxy in MAPLE. For the sake of brevity, we show how to emulate oblivious column access in OTAB since the row access can be done analogously.

Suppose the OTAB structure contains the data of a table with M columns and N rows. Let $\mathbf{q} = (\mathsf{op}, \mathsf{cid}, \mathbf{v}')$ be the client query to the MPC proxy, where op is the operation type (read, write, update), cid is the column identifier to be accessed and $\mathbf{v}'$ is the column data to be updated. The client distributes the authenticated shares of $\mathbf{q}$ to the proxy using the above authenticated input protocol as

$$\langle \mathbf{q} \rangle \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Input}(\mathbf{q})$$

The proxy then emulates the TreeORAM.Access procedure in OTAB (Figure 2, line 18) with boolean MPC. In MAPLE, we employ Circuit-ORAM [92] as the backend ORAM protocol due to its optimal circuit complexity. Recall that the access procedure in Circuit-ORAM invokes two main subroutines: Read and Evict, which can be emulated using the oblivious comparison $\Pi_{\mathsf{mpc}}.\mathsf{cmp}(\cdot)$ and selection $\Pi_{\mathsf{mpc}}.\mathsf{sel}(\cdot)$ protocols as follows.

In Read subroutine, the requested column cid is retrieved by accessing $O(\log m)$ columns that reside on the same logical path with cid in OTAB structure. Let $(\langle \mathbf{v}_1 \rangle, \ldots, \langle \mathbf{v}_m \rangle)$ be the authenticated shares of such columns and $(\langle x_1 \rangle, \ldots, \langle x_m \rangle)$ be the shares of their corresponding identifiers. The MPC proxy computes the share of a selection vector $\langle \mathbf{s} \rangle = (\langle s_1 \rangle, \ldots, \langle s_m \rangle)$ via oblivious comparison as

$$\langle s_i \rangle \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{cmp}(\langle \mathsf{cid} \rangle, \langle x_i \rangle) \tag{1}$$

for $i \in [m]$. Let $\langle \tilde{\mathbf{v}} \rangle$ be the share of the column cid initialized with 0^N. The proxy obtains $\langle \tilde{\mathbf{v}} \rangle$ by iterating the oblivious selection protocol between $\langle \tilde{\mathbf{v}} \rangle$ and m columns on the logical path as

$$\langle \tilde{\mathbf{v}} \rangle \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{sel}(\langle s_i \rangle, \langle \tilde{\mathbf{v}} \rangle, \langle \mathbf{v}_i \rangle) \tag{2}$$

for $i \in [m]$. In Evict subroutine, the proxy first computes the column to be updated via oblivious selection as

$$\langle \hat{\mathbf{v}} \rangle \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{sel}(\langle \mathsf{op} \rangle, \langle \tilde{\mathbf{v}} \rangle, \langle \mathbf{v}' \rangle)$$

The proxy then privately writes $\langle \hat{\mathbf{v}} \rangle$ back to OTAB structure and shuffle the columns in a logical path chosen deterministically. The shuffle is realized with boolean operations by computing an *eviction plan* that determines which columns to be moved in the logical path according to their assigned positions. As shown in Equation 1 and Equation 2, the proxy determines what columns to be picked by their identifiers using the oblivious comparison protocol, and then relocates the column data using the oblivious selection protocol.

Position map. As discussed, the MPC proxy needs to know the logical path of the requested column in OTAB structure for OTAB access emulation. In MAPLE, we store this path directly on the MPC proxy using the recursive technique in [83]. When emulating OTAB, the MPC proxy retrieves/updates the path of the requested column/row by accessing the recursive ORAM structure using the oblivious comparison and selection protocols as shown above.

Consistency in mixed row/column access sequence. As presented in §4, OTAB requires a permutation to convert the column/row data of the original table to the corresponding column/row in OTAB structure during write/update operations (i.e., lines 15–17 in Figure 2). In the multi-user setting, the user is untrusted and stateless and, therefore, we delegate such permutation to the MPC proxy using a permutation matrix. Specifically, we create a $m \times zm$ matrix $\mathbf{P}$, where $\mathbf{P}[i, j] = 1$ indicates the i-th original column is relocated at the j-th column in OTAB. Whenever the columns in OTAB are shuffled due to the previous column accesses, we perform the same computation on $\mathbf{P}$ so that $\mathbf{P}$ will keep track of the latest positions of the original columns in OTAB. Once the user updates a row $\mathbf{u}'$ in the original table, the MPC proxy first computes $\hat{\mathbf{u}} \leftarrow \mathbf{u}' \times \mathbf{P}$ to obtain the appropriate row in OTAB before emulating the ORAM access.

The other components that must be kept consistent are the column and row headers ($\mathbf{h}_{\mathrm{c}}$ and $\mathbf{h}_{\mathrm{r}}$). Similar to the permutation matrix, whenever we shuffle the columns (*resp.* rows) in OTAB during a column (*resp.* row) access using boolean MPC, we apply the same computation on $\mathbf{h}_{\mathrm{c}}$ (*resp.* $\mathbf{h}_{\mathrm{r}}$) to shuffle the original column identifiers accordingly.

Additional notation. We use $\langle \cdot \rangle$ to denote the emulation of OTAB access with MPC as $\langle \tilde{\mathbf{v}} \rangle \leftarrow \langle \mathsf{OTAB.Access} \rangle(\langle \mathsf{op} \rangle, \langle \mathsf{id} \rangle, \langle \mathbf{v}' \rangle)$, which returns the authenticated shares of the requested column/row.

5.4 Protocol Details

We present the protocol of our proposed MAPLE platform in Figure 3 and Figure 4 with the following details.

```
MAPLE.Setup($1^\lambda$, DB, $\mathcal{L}, m, N, t$):
 1: Initialize $\mathbf{I} \leftarrow 0^{N\times m}$, $\mathbf{A} \leftarrow 0^{N\times t}$, and $\mathbf{P} \leftarrow 0^{m\times z\cdot m}$
 2: for each document $d_i \in$ DB do
 3:     $\mathcal{W}_i \leftarrow$ Extract unique keywords in $d_i$
 4:     $\mathbf{u}_i \leftarrow$ BF.Gen($\mathcal{W}_i$)
 5:     $\mathbf{I}[i, *] \leftarrow \mathbf{u}_i$
 6:     $\mathbf{A}[i, j] \leftarrow \mathcal{L}(i, j)$ for $1 \leq j \leq t$
 7: $(\mathsf{pm}_d, \mathsf{pm}_w, \mathbf{h}_d, \mathbf{h}_w, \mathsf{IDX}) \leftarrow \mathsf{OTAB.Setup}(1^\lambda, \mathbf{I}, N, m)$
 8: $(\mathsf{pm}_d, \mathsf{pm}_u, \mathbf{h}_d, \mathbf{h}_u, \mathsf{AC}) \leftarrow \mathsf{OTAB.Setup}(1^\lambda, \mathbf{A}, N, t)$
 9: for $i = 1$ to $m$ do
10:     $\mathbf{P}[i, x_i] \leftarrow 1$, where $x_i \leftarrow \mathsf{pm}_w[i].\mathsf{id}$
11: $(\langle \mathsf{IDX} \rangle, \langle \mathsf{AC} \rangle, \langle \mathbf{h}_d \rangle, \langle \mathbf{h}_u \rangle) \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Input}(\mathsf{IDX}, \mathsf{AC}, \mathbf{h}_d, \mathbf{h}_u)$    ▹ $\mathcal{U} \rightarrow \mathcal{S}$
12: $(\langle \mathsf{pm}_d \rangle, \langle \mathsf{pm}_w \rangle, \langle \mathsf{pm}_u \rangle, \langle \mathbf{P} \rangle) \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Input}(\mathsf{pm}_d, \mathsf{pm}_w, \mathsf{pm}_u, \mathbf{P})$    ▹ $\mathcal{U} \rightarrow \mathcal{S}$
13: return $\mathcal{M} \leftarrow (\langle \mathsf{IDX} \rangle, \langle \mathsf{AC} \rangle, \langle \mathbf{h}_d \rangle, \langle \mathbf{h}_u \rangle, \langle \mathbf{P} \rangle, \langle \mathsf{pm}_d \rangle, \langle \mathsf{pm}_w \rangle, \langle \mathsf{pm}_u \rangle)$

MAPLE.KSearch($w$, uid; $\mathcal{M}$):
14: for $i = 1$ to $K$ do
15:     $\mathsf{cid}_i \leftarrow H_i(w)$
16: $(\langle \mathsf{cid}_1 \rangle, \ldots, \langle \mathsf{cid}_K \rangle, \langle \mathsf{uid} \rangle) \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Input}(\mathsf{cid}_1, \ldots, \mathsf{cid}_K, \mathsf{uid})$    ▹ $\mathcal{U} \rightarrow \mathcal{S}$
17: for $i = 1$ to $K$ do
18:     $\langle \hat{\mathbf{v}}_i \rangle \leftarrow \langle \mathsf{OTAB.Access} \rangle(\mathsf{read}, \langle \mathsf{cid}_i \rangle, \perp, \langle \mathsf{pm}_w \rangle, \langle \mathsf{IDX} \rangle)$
19: $\langle \mathbf{v}' \rangle \leftarrow \langle \mathsf{OTAB.Access} \rangle(\mathsf{read}, \langle \mathsf{uid} \rangle, \perp, \langle \mathsf{pm}_u \rangle, \langle \mathsf{AC} \rangle)$
20: $\langle \mathbf{v} \rangle \leftarrow \langle \hat{\mathbf{v}}_1 \rangle \boxtimes \langle \hat{\mathbf{v}}_2 \rangle \boxtimes \ldots \boxtimes \langle \hat{\mathbf{v}}_K \rangle \boxtimes \langle \mathbf{v}' \rangle$
21: $\langle \mathbf{v}'' \rangle \leftarrow \langle \mathbf{v} \rangle \boxtimes \langle \mathbf{h}_d \rangle$
22: $\mathbf{v}'' \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Output}(\langle \mathbf{v}'' \rangle)$    ▹ $\mathcal{S} \rightarrow \mathcal{U}$
23: return $\mathcal{J} \leftarrow \{\mathsf{fid} : \mathsf{fid} \in \mathbf{v}'' \wedge \mathsf{fid} \neq 0\}$
```

Figure 3: MAPLE Protocols.

5.4.1 Setup Phase. Given a document collection, the data owner initializes two empty tables $\mathbf{I}$ and $\mathbf{A}$ for search index and access control, respectively (Figure 3, line 1). The owner extracts keywords in each document, and creates a BF for keyword representation (lines 2–5). The owner also sets the user permission for each document on the access control (line 6). The owner then creates the oblivious tables IDX and AC, as well as their components for the search index and the access control, respectively (lines 7, 8). For simplicity, we assume the columns represent the keywords, while the rows represent the documents. Notice that it is necessary to apply the same random row permutation (line 3 in Figure 2) in both IDX and AC to ensure the positions of documents in these tables are identical for correct search.

Since the user is expected to receive a list of document identifiers from the search, we maintain a document header $\mathbf{h}_d$ indicating the document identifiers at each row of IDX. We also maintain a user header $\mathbf{h}_u$ indicating the user identifier at each column of IDX for permission update operations. As discussed in §5.3.2, once the oblivious table index is setup, we also create a permutation matrix $\mathbf{P}$ that maps keyword positions in the original index to the correct positions in IDX for update consistency (lines 9– 10) . Note that our scheme does not maintain permutation matrix for column operations and the keyword header $\mathbf{h}_w$ because update only happens row-wise and search only happens column-wise, respectively. Finally, once all necessary components are created, the data owner executes $\Pi_{\mathsf{mpc}}.\mathsf{Input}(\cdot)$ protocol to distribute them to the MPC proxy (line 11). A final remark is that in MAPLE, all the position maps ($\mathsf{pm}_d, \mathsf{pm}_w, \mathsf{pm}_u$) are stored at the proxy in the recursive ORAM structures under the form of authenticated shares.

5.4.2 Search Protocol. To search for a keyword w, the user computes its column identifiers cid_i in $\mathbf{I}$ with BF (line 15) and distributes them along with her identifier to the MPC proxy using the authenticated input protocol (line 16). The proxy then emulates OTAB access on IDX to retrieve the corresponding columns $\hat{\mathbf{v}}_{\mathsf{cid}_i}$ (line 18), and then emulate OTAB access on AC to retrieve the column $\mathbf{v}'$ storing the permission of uid on all documents (line 19). Due to BF, w appears in fid_j (with high probability) if $\hat{\mathbf{v}}_{\mathsf{cid}_i}[\mathsf{fid}_j] = 1 \; \forall \; \mathsf{cid}_i$. Thus, the MPC proxy performs the secure bitwise AND operation between all columns as $\mathbf{v} \leftarrow \hat{\mathbf{v}}_{\mathsf{cid}_1} \wedge \hat{\mathbf{v}}_{\mathsf{cid}_2} \wedge \cdots \wedge \hat{\mathbf{v}}_{\mathsf{cid}_K}$ to obtain the search result. Since $\mathbf{v}_{\mathsf{cid}_j}$ are all binary vectors, the bitwise AND is equivalent to multiplication over $\mathbb{F}_2$ and thus, it can be realized with the MPC multiplication protocol. To filter out documents that uid is not authorized, the proxy performs secure bitwise AND operation between $\mathbf{v}$ and $\mathbf{v}'$ (line 20).

The proxy then computes the list of document identifiers $\mathbf{v}''$ that match with the user query by performing an entry-wise multiplication $\mathbf{v}'' \leftarrow \mathbf{v} \times \mathbf{h}$ under secure computation (line 21). Finally, the proxy outputs the list of document identifiers to the user using the authenticated output protocol (lines 22–23).

```
MAPLE.DUpdate(fid, uid, $\mathbf{w}$, $\mathcal{M}$):
1: $\mathbf{u} \leftarrow \{0\}^m$
2: for $i = 1$ to $n$ and $j = 1$ to $K$ do
3:     $\mathsf{cid}_{i,j} \leftarrow H_j(w_i)$                         ▷ $\mathbf{w} = (w_1, \ldots, w_n)$
4:     $\mathbf{u}[\mathsf{cid}_{i,j}] \leftarrow 1$
5: $(\langle \mathbf{u} \rangle, \langle \mathsf{uid} \rangle, \langle \mathsf{fid} \rangle) \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Input}(\mathbf{u}, \mathsf{uid}, \mathsf{fid})$         ▷ $\mathcal{U} \rightarrow \mathcal{S}$
6: $\langle \hat{\mathbf{u}}' \rangle \leftarrow \langle \mathsf{OTAB.Access} \rangle(\mathsf{read}, \langle \mathsf{fid} \rangle, \perp, \perp, \langle \mathsf{AC} \rangle)$
7: for $i = 1$ to $z \cdot t$ do
8:     $\langle b \rangle' \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{cmp}(\langle \mathsf{uid} \rangle, \langle \mathbf{h}_u[i] \rangle)$
9:     $\langle b \rangle'' \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{cmp}(\langle \hat{\mathbf{u}}'[i] \rangle, \text{‘w’})$
10:    $\langle b \rangle \leftarrow \langle b' \rangle \boxtimes \langle b'' \rangle$
11: $\langle \mathsf{op} \rangle \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{sel}(\langle b \rangle, \mathsf{read}, \mathsf{write})$
12: $\langle \mathbf{u} \rangle \leftarrow \langle \mathbf{u} \rangle \boxtimes \langle \mathbf{P} \rangle$
13: $\langle \mathsf{OTAB.Access} \rangle(\langle \mathsf{op} \rangle, \langle \mathsf{fid} \rangle, \langle \mathbf{u} \rangle, \perp, \langle \mathsf{IDX} \rangle)$

MAPLE.PUpdate(fid, uid, $p$, $\mathcal{M}$):
14: $(\langle \mathsf{fid} \rangle, \langle \mathsf{uid} \rangle, \langle p \rangle) \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Input}(\mathsf{fid}, \mathsf{uid}, p)$         ▷ $\mathcal{U} \rightarrow \mathcal{S}$
15: $\langle \mathbf{u} \rangle \leftarrow 0^t$, $\langle \mathbf{u}[\mathsf{uid}] \rangle \leftarrow \langle p \rangle$
16: $\langle \mathsf{OTAB.Access} \rangle(\mathsf{update}, \langle \mathsf{fid} \rangle, \langle \mathbf{u} \rangle, \perp, \langle \mathsf{AC} \rangle)$
17: $\langle \mathsf{OTAB.Access} \rangle(\mathsf{read}, \langle \mathsf{fid} \rangle, \perp, \perp, \langle \mathsf{IDX} \rangle)$
```

Figure 4: MAPLE Protocols (cont.).

5.4.3 Update Protocols.

Document update. Given a document (with identifier fid) with a list of updated keywords $\mathbf{w}$, the user uid creates a BF $\mathbf{u}$ that represents the keyword appearance in the updated document (Figure 4, line 1). Specifically, $\mathbf{u}[x_{i,j}] = 1$, where $x_{i,j} = H_j(w_i)$ for each w_i and $j \in [K]$, while other elements are set to 0 (lines 2–4). The user then uses the authenticated input protocol to secret-share the BF representation $\mathbf{u}$, the user identifier uid, and the document identifier fid with the MPC proxy (line 5). The proxy then multiplies $\mathbf{u}$ with the permutation matrix $\mathbf{P}$ to permute the elements in $\mathbf{u}$ to their correct positions in IDX (line 12).

Finally, the proxy emulates OTAB read on AC to check if uid is permitted to update the document fid (lines 6–10). If so, the proxy emulates OTAB write on IDX over the row dimension with fid to update the document accordingly (line 13). Otherwise, it simply performs a dummy read on fid = 0, which happens when $b = 0$.

Remark that the positions of document identifiers in AC and IDX must be kept identical for correct permission check in search. Therefore, when implementing this protocol, we use the same selection query and eviction plan during OTAB accesses on AC and IDX to permute their rows consistently.

Permission update. MAPLE permits the access control to be updated with a restriction that only the data owner can perform this operation. Thus, the owner authenticates herself with the MPC proxy using a standard authentication mechanism (e.g., username/password).

Once authenticated, the data owner secret-shares the user identifier uid, the document identifier fid, and the new permission $p \in \{\mathsf{r}, \mathsf{w}, \perp\}$ to be updated to the MPC proxy using the authenticated input protocol (line 14). Next, the MPC proxy creates an update vector and emulates OTAB update on AC over the row dimension with fid to update the user permission accordingly (lines 15-16). Similar to the document update, it is necessary to perform a dummy access on IDX with the same selection query and eviction plan used in AC access to ensure the document positions in IDX and AC consistent for correct search (line 17).

5.5 Cost Analysis

We analyze the complexity of MAPLE. Let m, k, N, t, ℓ, z be the size of keyword representation per document (i.e., BF size), the BF parameter, the number of documents, and the number of users, the number of servers, and the back-end ORAM expansion, respectively. We assume ℓ, k, t, z are small constants (e.g., $k = 7, z = 2, \ell \in [6]$).

For keyword search, the client secret-shares a search query of size $O(k)$ to the MPC proxy, and receives a column of size $O(N \log N)$ as the search result. Given that $O(\log N)$ bits is needed for each document identifier representation, the total client bandwidth cost is $O(N \log N)$ and the computation time is $O(N)$. At the MPC proxy, the servers emulate k OTAB column accesses on the search index, in which each incurs $O(N \log m)$ inter-server bandwidth overhead and $O(N \log m)$ computation complexity due to Circuit-ORAM. For each access, MPC proxy also shuffles $O(\log m)$ columns (via multiplication) of the permutation matrix $\mathbf{P}$ to keep track of the keyword positions in OTAB, which takes $O(m \log m)$ time. The MPC proxy emulates an OTAB column access on the access control structure to check the user permission, which costs $O(N \log t)$ inter-server bandwidth and computation overhead. Finally, the MPC proxy performs multiplication between $(k + 1)$ columns, which costs $O(k \cdot N)$ bandwidth/processing overhead. Therefore, the total server cost is $O(k(N \log m + m \cdot \log m) + N \log t) = O(N \log m)$ given that $m \ll N$ and k, t are constants.

For document update, the user creates a row for keyword representation of the updated document with BF, which incurs $O(m)$ processing time. The user secret shares the row to the MPC proxy, which costs $O(m)$ bandwidth. At the proxy, the servers first permute the row using the permutation matrix, which incurs $O(m^2)$ bandwidth/processing overhead. The servers then emulate an OTAB row access to the access control structure for permission verification, which incurs $O(t \log N)$ bandwidth/processing overhead, and then emulate an OTAB row access to the search index for document update, which incurs $O(m \log N)$ bandwidth/processing overhead. In total, the server cost is $O(m^2 + t \log N + m \log N) = O(m^2 + m \log N)$ as t is fixed.

The cost of permission update is similar to that of document update, except that permutation matrix multiplication is not needed, i.e., $O(t \log N + m \log N) = O(m \log N)$.

For storage overhead, the user is stateless, meaning they do not need to store any information. For the MPC proxy, each server stores a share of the oblivious tables for the search index and the access control. These tables are of size $O(z^2 \cdot N \cdot m)$ and $O(z^2 \cdot N \cdot t)$. The position map components for these tables are stored in recursive ORAM and thus do not add the extra storage overhead asymptotically. The servers also maintain the permutation matrix of size $O(m^2)$ and the headers for oblivious tables, whose total size is $O(N + m)$. In total, since $z = 2$, t is constant, the overall storage cost per server is $O(z^2 \cdot N \cdot (m + t) + N + m + m^2) = O(N \cdot m + m^2)$.

6 SECURITY ANALYSIS

We analyze the security of OTAB and MAPLE schemes.

Lemma 1. *OTAB is an oblivious data structure by Definition 2 that securely realizes the ideal functionality for oblivious table in Figure 15.*

Theorem 1. *MAPLE achieves $\mathcal{L}$-simulation security by Definition 1, where $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}}\}$, $\mathcal{L}^{\mathsf{Stp}} = \{m, N, t\}$, $\mathcal{L}^{\mathsf{Srch}} = \{\emptyset\}$, and $\mathcal{L}^{\mathsf{Updt}} = \{\emptyset\}$.*

We present the proof in Appendix.

7 EXPERIMENTAL EVALUATION

7.1 Implementation and Configuration

We fully implemented our proposed platform in C++ consisting of ≈ 5,000 lines of code. We used `EMP-toolkit` library [93] to realize the MPC proxy with AGC technique [95] and `libzeromq` [6] library for network communication. Our implementation is available at

https://github.com/vt-asaplab/MAPLE.

Two-level parallelism. To optimize the performance of our scheme, we implemented parallelism at two levels including thread-level and server-level. For thread-level parallelism, we modified the implementation of the online phase in `EMP-toolkit` library to utilize multi-core CPUs, where the evaluation step can be performed in parallel by multi-threading. For server-level parallelism, our platform makes use of each server in the MPC proxy as both the garbler and evaluator, where each server in ℓ servers evaluates a data-independent circuit of size T/ℓ for the entire circuit of size T.

Hardware and network. We deployed the MPC proxy with Amazon EC2 `r5n.16xlarge` instances each equipped with 32-core Intel Xeon Platinum 8375C CPU @ 2.90GHz and 512 GB memory. We used a 2021 Macbook Pro 14 as the user. The network bandwidth between servers is 37 Gbps and the network bandwidth between servers and the user is 20 Mbps with a 10ms round-trip latency.

Dataset. We used a subset of enwiki-latest-pages-articles dataset [8] to evaluate the performance of our technique. We used WikiExtractor [10] to extract 16,331,564 distinct text-only articles from the dataset. We removed empty articles that contain no keyword and randomly selected 1M articles as the testing set. We extract unique keywords using the standard tokenization method. The size of keyword universe is $M = 4{,}080{,}169$.

7.2 Parameters and Counterparts Selection

We compare MAPLE with other encrypted search platforms that offer similar functionalities or security properties (e.g., metadata hiding) to our technique. Specifically, we choose ODSE [56], DURASIFT [39] and DORY [35] as the main counterparts. We selected standard parameters to achieve 128-bit security in all schemes as follows.

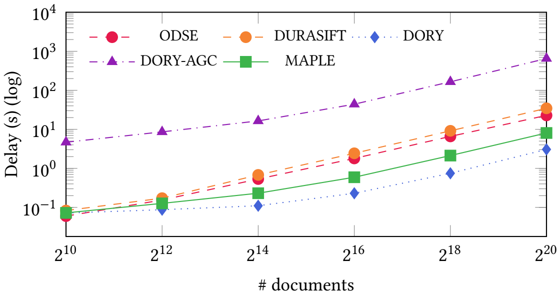


Figure 5: Search delay of MAPLE and its counterparts.

- **MAPLE**: We used standard parameters for AGC including 128-bit AES/MAC keys. For BF, we selected parameters such that the FP rate is small. Specifically, ≈ 99.54% number of documents have ≤ 300 keywords in our testing set (see Figure 20 in Appendix E). So, we set BF size $m = 2^{14}$ and the number of hashing $k = 7$ to achieve $< 10^{-6}$ FP rate (§5.2) to adapt for the case of 1 million documents. For OTAB, we selected standard Circuit-ORAM parameters including bucket size $z = 2$ and stash sizes $S_w = \log_2(m)$, $S_f = \log_2(N)$, $S_u = \log_2(t)$, where m is BF size, N is the number of documents, and t is the number of users.
- **ODSE** [56]: We used standard parameters as suggested in the original work. We compare our scheme with $\text{ODSE}^{\text{wo}}_{\text{xor}}$, which employs XOR-PIR and Write-Only ORAM for oblivious search and update, respectively, along with the computational MAC technique with 128-bit key.
- **DURASIFT** [39]: We used standard parameters including 128-bit key for IND-CPA encryption, 128-bit seed for pseudorandom generator (PRG), DPF in [43] and GMW protocol in [46] for oblivious search. The size of the search index is $N \cdot M$, where M is the size of the keyword universe.
- **DORY** [35]: We selected the BF parameters similar to our scheme including $m = 2^{14}$ and $k = 7$. Note that the BF size is larger than the one chosen in the original work since we test on much larger dataset (i.e., up to 300 keywords compared with 73 keywords per document in Enron dataset originally used in [35]). We used 128-bit keys for IND-CPA encryption, MAC, and PRG seed. DORY uses DPF in [43] for oblivious search. Additionally, we create a variant of DORY (denoted as DORY-AGC) that has the same trust assumption as ours (i.e., trustless user) by substituting its DPF-based PIR with AGC-based PIR. We compare MAPLE with this variant to demonstrate our performance advantage under the same security assumption.

Evaluation metrics. We evaluate our scheme and its counterparts according to the overall latency, the client/server processing delay, and the network communication overhead. We further evaluate the scalability of our scheme with various numbers of servers to improve the privacy degree.

7.3 Overall Results

7.3.1 Keyword Search. Figure 5 illustrates the search latency of our scheme and its counterparts with different numbers of documents ranging from 2^{10} to 2^{20}. We can see that our scheme is up to 4.3× faster than DURASIFT, 1.2×–3.1× faster than ODSE but 1.3×–2.8× slower than DORY. With 1K users and 64K documents, our scheme takes approximately 0.6s to accomplish a search, and increases

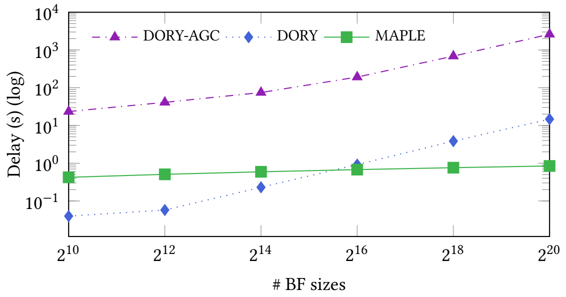


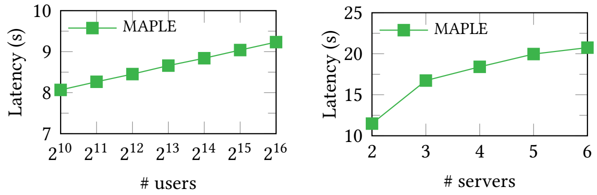
Figure 6: Search delay with varied BF sizes.

to about 8.1s for 1M documents. The latency of all the schemes grows almost linearly to the number of documents since the search complexity of all the schemes is linear to the number of documents.

MAPLE is slightly slower than DORY mostly due to the communication overhead incurred by AGC protocol, which does not happen in DORY as it uses DPF. However, it is worth noting that our scheme can offer security for both malicious users and malicious servers plus access control, while DORY can only offer security for a malicious server (no access control, trusted client). On the other hand, MAPLE outperforms both ODSE and DURASIFT with 3×–4× faster search time, since their complexity grows linearly to the size of *keyword universe*, while it only grows logarithmically to the size of keyword representation in MAPLE. With 1M documents, the keyword universe can contain up to 2.7M keywords, while the keyword representation is fixed at 16K (i.e., ≈ 160× smaller) to cover 99.54% of the entire dataset. When comparing DORY under the same trust assumption (DORY-AGC), MAPLE is 64.7×–80.9× faster. This is because MAPLE incurs a quasilinear complexity (w.r.t the table index size) due to OTAB while DORY-AGC incurs a quadratic overhead due to MPC evaluation for PIR.

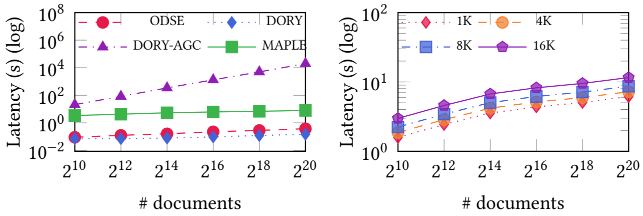
In summary, the cost difference between MAPLE/DORY over DURASIFT/ODSE is mostly due to the use of probabilistic data structure (i.e., BF) vs. plain bitmap for search index. When applying BF on DURASIFT/ODSE, we estimated that their cost will be similar to DORY. Meanwhile, the cost difference between MAPLE vs. DORY (or DURASIFT/ODSE) is mainly due to the ORAM emulation by MPC vs. efficient PIR evaluation thanks to trusted user(s). When putting all the schemes under the same trust assumption (trustless users), MAPLE outperforms other platforms by almost two orders of magnitude due to the complexity difference between OTAB and PIR. We further analyze the performance of OTAB in §7.5.

Varying BF sizes. Figure 6 demonstrates the impact of varying BF sizes on the performance of MAPLE and DORY under $N = 2^{16}$ documents. MAPLE is 2.6×–10.7× slower than DORY when $m \leq 2^{14}$, and start to outperform DORY when $m \geq 2^{16}$ with one order of magnitude faster. It is due to the fact that the search complexity in MAPLE is only logarithmic to the BF size, compared with linear growth in DORY. Compared with DORY-AGC, MAPLE is *two to three orders* of magnitude faster especially when BF size is large (2^{20}). Note that larger BF size permits each document to contain more unique keywords or reduce the false positive rate, and thus our scheme will be more beneficial for large database settings that require high precision search. For example, there are many documents in the enwiki-latest-pages-articles dataset that contain more than 3,500 keywords, which requires the BF size to be set to 2^{18} to achieve less than one false positive document.



(a) Varying # users **(b) Varying # servers**

Figure 7: Search delay of MAPLE by varying # users and # servers.



(a) Document update **(b) Permission update**

Figure 8: Update delay of MAPLE and its counterparts.

Varying number of users. Figure 7a presents the end-to-end search delay of our scheme with varied numbers of users. Increasing the number of users does not significantly impact the search delay of our scheme. Specifically, increasing the number of users 64 times (from 1K to 64K users) only adds 1.2 seconds to the end-to-end search delay (≈ 1.1 times). This is due to the use of oblivious table to privately verify the user permission, whose cost only grows logarithmically to the number of users.

Varying number of servers. We further evaluate the impact of increasing the number of servers to improve privacy degree on the search delay. Figure 7b shows that the search latency increases non-linearly w.r.t the number of servers. This is because having more servers reduces the size of circuits evaluated by each server, but increases the overhead to evaluate AND and XOR gates. With ℓ servers and T-sized circuit, the circuit size that each server evaluates is T/ℓ, while each XOR gate evaluation incurs $\ell - 1$ XOR operations and each AND gate evaluation incurs $(\ell-1)(3\ell+5)$ XOR operations plus $(\ell - 1)\ell$ AES invocations. As a result, adding one more server increases the network overhead to 16.7%, 8.3%, 5% and 3.3% over the time starting from $\ell = 2$ to 6 servers. The increasing number for XOR operations are 61.9%, 32.4%, 22.7%, and 17.8%, and the number of AES operations increases 100%, 50%, 33.3%, and 25%.

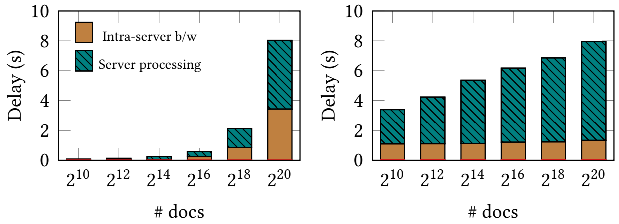
7.3.2 Update.

Document update. Figure 8a presents the update delay of MAPLE and its counterparts. MAPLE takes 3.4s and 6.2s to update a document in 1K documents and 64K documents, respectively, and goes up to 7.9s for 1M documents. The majority of update overhead stems from the nonconsecutive memory access of row data given that the oblivious table is stored column-wise for efficient search. Another factor is due to the permutation to rearrange the keyword representation, which takes a fixed amount of 1.8s for the BF of size $m = 2^{14}$ regardless of number of documents in the dataset.

MAPLE is 3.3s–7.8s slower than DORY and ODSE, which take 70–150ms, and 30–373ms, respectively, to finish a document update.

Table 2: Monetary cost per search/update operation in MAPLE.

	# Docs	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Srch	**Time (s)**	0.0723	0.1267	0.2308	0.5923	2.1276	8.0671
	Cost ($)	0.0002	0.0003	0.0006	0.002	0.006	0.021
Updt	**Time (s)**	3.3844	4.2317	5.3618	6.1727	6.8564	7.9426
	Cost ($)	0.009	0.011	0.014	0.016	0.018	0.021



(a) Search **(b) Update**

Figure 9: Detailed cost of keyword search and document update.

Note that we do not compare with DURASIFT as it does not offer document update functionality. Although MAPLE is slower than DORY and ODSE, it is worth noting that MAPLE guarantees update pattern privacy (not achieved by DORY), and document sharing with access control under untrusted user assumption (not offered by DORY and ODSE). When comparing with the oblivious update version of DORY (i.e., DORY-AGC), we can see that MAPLE is 6.2×–2491.1× faster. It is because MAPLE utilizes OTAB for oblivious row access with $O(m \log N)$ cost, while DORY-AGC costs $O(mN)$ due to PIR. Thus, we envision our MAPLE platform is highly desirable for applications where the document update is infrequent but its metadata privacy must be guaranteed.

Permission update. We present the delay to update user permission in MAPLE in Figure 8b. Since we perform permission update on the row dimension, its cost grows logarithmically to the number of documents, where in the case of 1K users it takes 1.6s with 1K documents, and goes up to 4.4s for 64K documents and 6.1s for 1M documents. Permission update is slightly faster than document update since it does not require permutation that costs a fixed amount of 1.8s. We also experiment the impact of user set on the permission update delay. Specifically, the delay to update a user permission among 1K, 4K, 8K, and 16K in the case of 1M documents is approximately 6.1s, 7.2s, 8.7s and 11.6s, respectively. This matches with the asymptotic complexity of permission update, where the cost grows logarithmically to the number of documents, but linearly to the number of users due to the oblivious table.

7.3.3 *Monetary cost.*

We report the monetary cost for each keyword search and document update operation by our MAPLE scheme in Table 2. The cost is computed based on two Amazon EC2 servers (`r5n.16xlarge`) pricing at $4.77/hr per instance.

7.4 Cost Breakdown

Figure 9 presents the detailed cost of keyword search and document update in MAPLE. There are three factors that impact the MAPLE delay including the user processing, the computation cost at the proxy and the network communication.

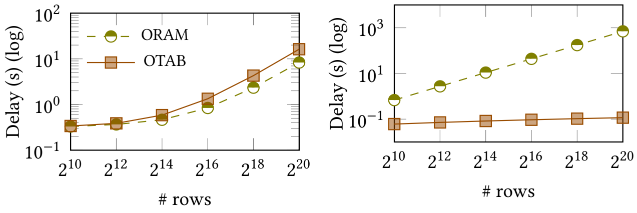
Client processing. In MAPLE, the client cost is negligible because she only needs to execute the MPC input and output protocols with the MPC proxy, which invokes cheap arithmetic operations. For keyword search, the client additionally invokes $k = 7$ calls to the hash function to compute the column identifiers of the searched keyword in the BF. For document update, since the user only generates a small updated keyword representation, its overhead is also minimal. In our benchmark, all these operations took less than 20ms, which is hardly visible in Figure 9 where the total delay is dominated by the overhead at the proxy.

Network communication. In MAPLE, there are two types of communication: (*i*) between the user and the proxy, (*ii*) between the servers in the proxy during MPC execution. The user network is negligible since the user only shares the search/update query to the proxy and receives the result, whose total cost is $O(\ell N \log N)$-bit for search (i.e., ≈ 16 MB for 1M documents with $\ell = 2$ servers), and $O(\ell m)$ for update (i.e., ≈ 4 KB for the BF size $m = 2^{14}$).

The majority overhead of network delay stems from the inter-server communication to emulate OTAB accesses and other secure computations with MPC, which contributes up to 43% to the total search delay and 32% to the total update delay as shown in Figure 9. Due to AGC [95], there is a step in the online phase where the "garbler" server and the other "evaluator" server exchange masked inputs and masked labels to each other. For each wire, its masked value is of 1 B and its masked label is of 16 B. For each search, the proxy emulates 8 OTAB accesses (7 for search index and 1 for access control). As we use Circuit-ORAM [92] as the backend ORAM, the evaluation circuit for a column access in the search index and access control has $\approx 3N(z \log m + S_w)$ and $\approx 3N(z \log t + S_u)$ input wires, respectively. Concretely, the total evaluation circuit size per keyword search on the search index is ≈ 15.5 MB, 992.3 MB, and 15.5 GB, for 1K, 64K, and 1M documents, respectively. The total evaluation circuit size per user permission check among 1K users is ≈ 1.6 MB, 101.3 MB and 1.6 GB, for 1K, 64K, and 1M documents, respectively. As a result, in the search procedure, network delay for search is about 3.3ms, 0.23s, and 3.4s for 1K, 64K, and 1M documents, respectively, which attributes 4.6% - 42.5% to the total delay.

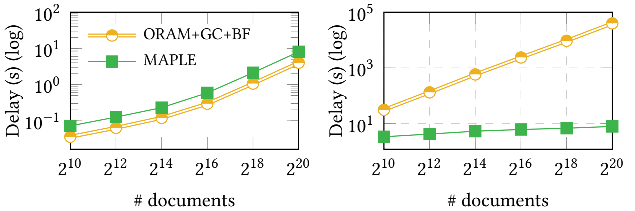
In document update, the majority of network overhead is due to the permutation, which requires 2^{28} input wires resulting in the bandwidth of 4.5 GB. The evaluation circuit for row access in the search index has $\approx 3m(z \log N + S_f)$ and ≈ 1.5M, 2.4M, and 2.9M input wires for 1K, 64K, and 1M documents, respectively. Also, for 1K users, the evaluation circuit for a column access in the access control has $\approx 3N(z \log t + S_u)$ and ≈ 92.2K, 5.9M, and 94.4M input wires for 1K, 64K, and 1M documents, respectively. Concretely, the total bandwidth cost per document update on the search index is ≈ 4.5 GB, 4.6 GB, and 6.1 GB, for 1K, 64K, and 1M documents, respectively. As a result, in the update procedure, network delay is about 1.1s-1.3s, which attributes 14.2% - 24.1% to the total delay.

Proxy processing. As presented in §5.3.2, search and update operations in our scheme are realized with comparison and selection circuits containing XOR and AND gates. Since comparison is only executed on a very small data portion (i.e., at most 32-bit for column/row identifiers), the majority of proxy computation stems from the execution of selection circuits on the entire rows/columns of the search index and access control. Each selection evaluation circuit of T input wires contains $3T$ XOR gates and T AND gates. Evaluating an XOR gate is cheap because it incurs XOR operations



(a) Column access **(b) Row access**

Figure 10: Performance of OTAB vs. plain ORAM for table access.



(a) Keyword search **(b) Document update**

Figure 11: Performance of MAPLE vs. plain ORAM.

including 1 operation between two input wire values and $(\ell-1)$ operations between the corresponding label values. Evaluating AND gates is more expensive, where each gate incurs 1 XOR between two input wire values, $(\ell-1)(3\ell+5)$ XOR between 128-bit values, and $(\ell-1)\ell$ AES invocations. For keyword search, the servers evaluate $\approx (63N(z\log m+S_w)+9N(z\log t+S_u))$ (concretely from 3.0M to 3.1B) XOR gates and $\approx (21N(z\log m+S_w)+3N(z\log t+S_u)+7N+N\log N)$ (from 1.0M to 1.0B) AND gates for 1K to 1M documents with 1K users. Given that two-level parallelism is implemented, evaluating all these XOR and AND gates takes 70ms–4.6s, which attributes 57.4%–90.9% to the total delay. For document update, the servers evaluate a total of $\approx (m(m-1)+9m(z\log N+S_f)+9N(z\log t+S_u))$ (from 273.1M to 560.4M) XOR gates and $\approx (m^2+3m(z\log N+S_f)+m+3N(z\log t+S_u))$ (from 270.0M to 365.8M) AND gates. The amount of overhead from evaluating $O(m^2)$ XOR and AND gates for permutation is about 0.9s. Most latency stems from random memory access when accessing rows in the oblivious table. This is because we store the oblivious table for search-friendly, where the table is stored column-wise in the memory. Each document update takes 2.3s–6.6s to evaluate all XOR and AND gates, as well as read/write the row data from the memory, which accounts for 51.1%–71.2% of the total delay.

7.5 Comparison with ORAM

We perform experiments to compare the performance of MAPLE with directly applying plain ORAM on table structure (without using OTAB). We first benchmark the performance of OTAB compared with directly using ORAM to access row/column in a table structure. In this experiment, we use Circuit-ORAM with the same parameter (bucket size $z=2$, stash size $S_c=14$) for both plain ORAM and OTAB approach as the underlying ORAM scheme to perform column access, and bucket size $z=2$, stash size $S_r=\log(N)$, where N is the number of rows for oblivious row access in OTAB. In the plain ORAM approach, we package each column of the original table as a Circuit-ORAM block, while OTAB doubles the number of rows and columns in the original table.

Figure 10a and Figure 10b present the end-to-end delay of accessing a single column and row in a table with 2×2^{14} columns and varied number of rows, respectively. We can see that accessing a column with plain ORAM is 1.1×–1.9× faster than OTAB. It is because in the plain ORAM approach, each column is packaged directly as an ORAM block, and the size of each column is not doubled as in OTAB to assist oblivious update. Meanwhile, row access in plain ORAM takes two to three orders of magnitude more time than OTAB. It is because plain ORAM can only support efficient column access, but not row access. To hide the row access patterns, it requires processing the entire table.

We now report the performance of MAPLE and the plain ORAM approach when both use MPC to enable oblivious search and multi-user functionalities. Figure 11a and Figure 11b present the end-to-end search and update delay of MAPLE and plain ORAM on BF table with MPC, respectively. In the plain ORAM approach, since the column size is not expanded to support oblivious update, its search latency is about 2× smaller than our MAPLE since it invokes MPC evaluation around 2 times less than MAPLE. However, in document update, plain ORAM approach takes 9.3×–4982.1× more time than MAPLE. It is due to the fact that plain ORAM incurs MPC evaluation (e.g., secure AND/XOR operations) on $O(N)$ rows compared with only $O(\log N)$ rows per document access in MAPLE.

8 CONCLUSION

In this paper, we introduced MAPLE, a new metadata-hiding platform that enables encrypted search and update on shared data across multiple users maintaining privacy-preserving fine-grained policy controls. MAPLE is the first to compose various cryptographic techniques including secure computation, oblivious probabilistic data structures, and ORAM for high security/privacy guarantees. Experimental results demonstrated that MAPLE is efficient for both keyword search and document update operations.

From a real-world use-case perspective, MAPLE is highly desirable for applications featuring a balanced frequency of keyword search and document update operations, in which the privacy of each operation is highly critical (i.e., no metadata leakage in any type of operations), and a small error in the search result is acceptable. MAPLE is less favorable for applications that are primarily focused on either search or update operations. Additionally, if the perfect search accuracy is paramount and the update privacy is not a primary concern, alternative techniques that are solely based on ORAM, PIR, or standard SSE may be more appropriate.

ACKNOWLEDGMENTS

We would like to thank our shepherd and the anonymous reviewers in PETS 2023 for their insightful comments and suggestions to improve the quality of our work. This research is supported by an unrestricted gift from Robert Bosch, 4-VA, and the Commonwealth Cyber Initiative (CCI) — an investment in the advancement of cyber R&D, innovation, and workforce development. For more information about CCI, visit www.cyberinitiative.org.

REFERENCES

[1] 2008. What can Dropbox help you do? https://www.dropbox.com.
[2] 2011. Sync: Secure cloud storage, privacy guaranteed. https://www.sync.com/.
[3] 2012. Google Drive: Easy and secure access to your content. https://www.google.com/drive/.

[4] 2013. Boxcryptor: No. 1 cloud encryption made in Germany. https://www.boxcryptor.com/en/.

[5] 2013. Tresorit: Secure file sharing & content collaboration with encryption. https://tresorit.com/secure-file-sharing.

[6] 2021. ZeroMQ: An open-source universal messaging library. https://github.com/zeromq/libzmq

[7] 2022. IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/oya

[8] 2022. The Wikipedia Corpus. https://dumps.wikimedia.org/.

[9] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. 2014. Verifiable oblivious storage. In *International Workshop on Public Key Cryptography*. Springer, 131–148.

[10] Giusepppe Attardi. 2015. WikiExtractor. https://github.com/attardi/wikiextractor.

[11] Adam J Aviv, Seung Geol Choi, Travis Mayberry, and Daniel S Roche. 2016. Oblivisync: Practical oblivious file backup and synchronization. *arXiv preprint arXiv:1605.09779* (2016).

[12] Adam J. Aviv, Seung Geol Choi, Travis Mayberry, and Daniel S. Roche. 2017. ObliviSync: Practical Oblivious File Backup and Synchronization. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/oblivisync-practical-oblivious-file-backup-and-synchronization/

[13] Michael Backes, Amir Herzberg, Aniket Kate, and Ivan Pryvalov. 2016. Anonymous ram. In *European Symposium on Research in Computer Security*. Springer, 344–362.

[14] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. 2011. Semi-homomorphic encryption and multiparty computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 169–188.

[15] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2019. Revisiting leakage abuse attacks. *Cryptology ePrint Archive* (2019).

[16] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. 2014. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 203–214.

[17] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[18] Christoph Bösch, Pieter Hartel, Willem Jonker, and Andreas Peter. 2014. A survey of provably secure searchable encryption. *ACM Computing Surveys (CSUR)* 47, 2 (2014), 1–51.

[19] Christoph Bösch, Andreas Peter, Bram Leenders, Hoon Wei Lim, Qiang Tang, Huaxiong Wang, Pieter Hartel, and Willem Jonker. 2014. Distributed searchable symmetric encryption. In *2014 Twelfth Annual International Conference on Privacy, Security and Trust*. IEEE, 330–337.

[20] Raphael Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1465–1482.

[21] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1465–1482.

[22] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. 2020. Efficient 3-party distributed ORAM. In *International Conference on Security and Cryptography for Networks*. Springer, 215–232.

[23] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. *IACR Cryptol. ePrint Arch.* 2014 (2014), 853.

[24] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2013. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Annual cryptology conference*. Springer, 353–373.

[25] T-H Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. 2018. 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*. Springer, 158–188.

[26] Binyi Chen, Huijia Lin, and Stefano Tessaro. 2016. Oblivious parallel RAM: improved efficiency and generic constructions. In *Theory of Cryptography Conference*. Springer, 205–234.

[27] Weikeng Chen, Thang Hoang, Jorge Guajardo, and Attila A. Yavuz. 2022. Titanium: A Metadata-Hiding File-Sharing System with Malicious Security. Cryptology ePrint Archive, Paper 2022/051. https://doi.org/10.14722/ndss.2022.24161 https://eprint.iacr.org/2022/051.

[28] Weikeng Chen and Raluca Ada Popa. 2020. Metal: a metadata-hiding file-sharing system. In *NDSS Symposium 2020*.

[29] Benny Chor, Niv Gilboa, and Moni Naor. 1997. *Private information retrieval by keywords*. Citeseer.

[30] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private information retrieval. *Journal of the ACM (JACM)* 45, 6 (1998), 965–981.

[31] Sherman SM Chow, Katharina Fech, Russell WF Lai, and Giulio Malavolta. 2020. Multi-client oblivious RAM with poly-logarithmic communication. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 160–190.

[32] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[33] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 655–671.

[34] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 655–671.

[35] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. 2020. DORY: An Encrypted Search System with Distributed Trust. Cryptology ePrint Archive, Report 2020/1280. https://ia.cr/2020/1280.

[36] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. {SEAL}: Attack mitigation for encrypted databases via adjustable leakage. In *29th USENIX Security Symposium (USENIX Security 20)*. 2433–2450.

[37] Saba Eskandarian and Matei Zaharia. 2017. Oblidb: Oblivious query processing for secure databases. *arXiv preprint arXiv:1710.00458* (2017).

[38] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proceedings of the VLDB Endowment* 13, 2 (2019).

[39] Brett Hemenway Falk, Steve Lu, and Rafail Ostrovsky. 2019. DURASIFT: A Robust, Decentralized, Encrypted Database Supporting Private Searches with Complex Policy Controls. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society* (London, United Kingdom) *(WPES'19)*. Association for Computing Machinery, New York, NY, USA, 26–36. https://doi.org/10.1145/3338498.3358651

[40] Amos Fiat and Adi Shamir. 1986. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*. Springer, 186–194.

[41] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. 2017. HardIDX: Practical and secure index with SGX. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 386–408.

[42] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New constructions for forward and backward private symmetric searchable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1038–1055.

[43] Niv Gilboa and Yuval Ishai. 2014. Distributed Point Functions and Their Applications. In *EUROCRYPT*.

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

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

[46] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to play any mental game. In *Annual ACM Symposium on Theory of Computing*.

[47] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[48] S Dov Gordon, Jonathan Katz, and Xiao Wang. 2018. Simple and efficient two-server ORAM. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 141–157.

[49] Zichen Gui, Kenneth G Paterson, and Sikhar Patranabis. 2023. Rethinking searchable symmetric encryption. In *2023 IEEE Symposium on Security and Privacy (SP)*.

[50] Ariel Hamlin, Rafail Ostrovsky, Mor Weiss, and Daniel Wichs. 2019. Private anonymous data access. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 244–273.

[51] Thang Hoang, Rouzbeh Behnia, Yeongjin Jang, and Attila A Yavuz. 2020. MOSE: Practical multi-user oblivious storage via secure enclaves. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. 17–28.

[52] Thang Hoang, Jorge Guajardo, and Attila A Yavuz. 2020. MACAO: A maliciously-secure and client-efficient active ORAM framework. *NDSS 2020* (2020).

[53] Thang Hoang, Ceyhun D Ozkaptan, Gabriel Hackebeil, and Attila Altay Yavuz. 2018. Efficient oblivious data structures for database services on the cloud. *IEEE Transactions on Cloud Computing* 9, 2 (2018), 598–609.

[54] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. 2017. 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. 491–505.

[55] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. 2019. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *Proceedings on Privacy Enhancing Technologies* 2019, 1 (2019).

[56] Thang Hoang, Attila Yavuz, F. Durak, and Jorge Guajardo. 2019. A multi-server oblivious dynamic searchable encryption framework. *Journal of Computer Security* 27 (09 2019), 1–28. https://doi.org/10.3233/JCS-191300

[57] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. 2020. Ghostor: Toward a Secure {Data-Sharing} System from Decentralized Trust. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 851–877.

[58] T-H Hubert Chan and Elaine Shi. 2017. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In *Theory of Cryptography Conference*. Springer, 72–107.

[59] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS*.

[60] Seny Kamara, Tarik Moataz, Andrew Park, and Lucy Qin. 2021. A Decentralized and Encrypted National Gun Registry. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1520–1537. https://doi.org/10.1109/SP40001.2021.00072

[61] Seny Kamara and Charalampos Papamanthou. 2013. Parallel and dynamic searchable symmetric encryption. In *International conference on financial cryptography and data security*. Springer, 258–274.

[62] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 965–976.

[63] Jonathan Katz and Andrew Y. Lindell. 2008. Aggregate Message Authentication Codes. In *Topics in Cryptology – CT-RSA 2008*, Tal Malkin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–169.

[64] Aggelos Kiayias, Ozgur Oksuz, Alexander Russell, Qiang Tang, and Bing Wang. 2016. Efficient encrypted keyword search for multi-user data sharing. In *European symposium on research in computer security*. Springer, 173–195.

[65] Mehmet Kuzu, Mohammad Saiful Islam, and Murat Kantarcioglu. 2012. Efficient similarity search over encrypted data. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1156–1167.

[66] Steven Lambregts, Huanhuan Chen, Jianting Ning, and Kaitai Liang. 2022. VAL: Volume and Access Pattern Leakage-Abuse Attack with Leaked Documents. In *European Symposium on Research in Computer Security*. Springer, 653–676.

[67] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-An Tan. 2014. Search Pattern Leakage in Searchable Encryption: Attacks and New Construction. *Inf. Sci.* 265 (may 2014), 176–188. https://doi.org/10.1016/j.ins.2013.11.021

[68] Jacob R Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. 2013. Shroud: Ensuring Private Access to {Large-Scale} Data in the Data Center. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. 199–213.

[69] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. 2015. Privacy and access control for outsourced personal records. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 341–358.

[70] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. 2017. Maliciously secure multi-client ORAM. In *International Conference on Applied Cryptography and Network Security*. Springer, 645–664.

[71] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. 2017. Maliciously secure multi-client ORAM. In *International Conference on Applied Cryptography and Network Security*. Springer, 645–664.

[72] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. 2013. Efficient private file retrieval by combining ORAM and PIR. *NDSS 2013* (2013).

[73] Jonathan Mayer, Patrick Mutchler, and John C Mitchell. 2016. Evaluating the privacy properties of telephone metadata. *Proceedings of the National Academy of Sciences* 113, 20 (2016), 5536–5541.

[74] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 279–296.

[75] Simon Oya and Florian Kerschbaum. 2021. Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 127–142. https://www.usenix.org/conference/usenixsecurity21/presentation/oya

[76] Vasilis Pappas, Mariana Raykova, Binh Vo, Steven M Bellovin, and Tal Malkin. 2011. Private search in the real world. In *Proceedings of the 27th Annual Computer Security Applications Conference*. 83–92.

[77] Geong Sen Poh, Ji-Jian Chin, Wei-Chuen Yau, Kim-Kwang Raymond Choo, and Moesfa Soeheila Mohamad. 2017. Searchable symmetric encryption: designs and challenges. *ACM Computing Surveys (CSUR)* 50, 3 (2017), 1–37.

[78] Daniel S Roche, Adam Aviv, Seung Geol Choi, and Travis Mayberry. 2017. Deterministic, stash-free write-only oram. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 507–521.

[79] Alan Rusbridger. 2013. The Snowden leaks and the public. , 31–34 pages.

[80] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. 2016. Taostore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 198–217.

[81] Jacob T Schwartz. 1980. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)* 27, 4 (1980), 701–717.

[82] Zhiwei Shang, Simon Oya, Andreas Peter, and Florian Kerschbaum. 2021. Obfuscated access and search patterns in searchable encryption. *arXiv preprint arXiv:2102.09651* (2021).

[83] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O((logN) 3) worst-case cost. In *International Conference on The Theory and Application of Cryptology and Information Security*. Springer, 197–214.

[84] Dawn Xiaoding Song, D. Wagner, and A. Perrig. 2000. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*. 44–55. https://doi.org/10.1109/SECPRI.2000.848445

[85] Emil Stefanov and Elaine Shi. 2013. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 247–258.

[86] Emil Stefanov and Elaine Shi. 2013. Multi-cloud oblivious storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 247–258.

[87] Shi-Feng Sun, Ron Steinfeld, Shangqi Lai, Xingliang Yuan, Amin Sakzad, Joseph K Liu, Surya Nepal, and Dawu Gu. 2021. Practical Non-Interactive Searchable Encryption with Forward and Backward Privacy.. In *NDSS*.

[88] Shi-Feng Sun, Xingliang Yuan, Joseph K Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. 2018. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 763–780.

[89] Afonso Tinoco, Sixiang Gao, and Elaine Shi. 2022. EnigMap: Signal Should Use Oblivious Algorithms for Private Contact Discovery. *Cryptology ePrint Archive* (2022).

[90] Shruti Tople, Yaoqi Jia, and Prateek Saxena. 2019. {PRO-ORAM}: Practical {Read-Only} Oblivious {RAM}. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 197–211.

[91] Jiafan Wang and Sherman SM Chow. 2022. Omnes pro uno: Practical multi-writer encrypted database. In *USENIX Security*.

[92] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 850–861.

[93] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[94] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 21–37.

[95] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-scale secure multiparty computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 39–56.

[96] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 215–226.

[97] Peter Williams, Radu Sion, and Alin Tomescu. 2012. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 977–988.

[98] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. 162–167. https://doi.org/10.1109/SFCS.1986.25

[99] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 707–720. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang

[100] Richard Zippel. 1979. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*. Springer, 216–226.

A OBLIVIOUS TABLE VISUALIZATION

We present a toy example of how column and row accesses in OTAB are performed in Figure 12. Given a table $\mathbf{M}$ of size 3×2, the oblivious table (OTAB) of $\mathbf{M}$ is a (bigger) table of size 7×3, where each dimension of $\mathbf{M}$ is expanded to $O(Z)$ times (where Z is the bucket size of the underlying Tree-based ORAM scheme), and the row and column data of $\mathbf{M}$ is assigned to random row and column locations, respectively, in OTAB. OTAB is augmented by a row header $\mathbf{h}_\mathrm{r}$ and a column header $\mathbf{h}_\mathrm{c}$ that keep track of the locations of the original rows and columns of $\mathbf{M}$ in OTAB.

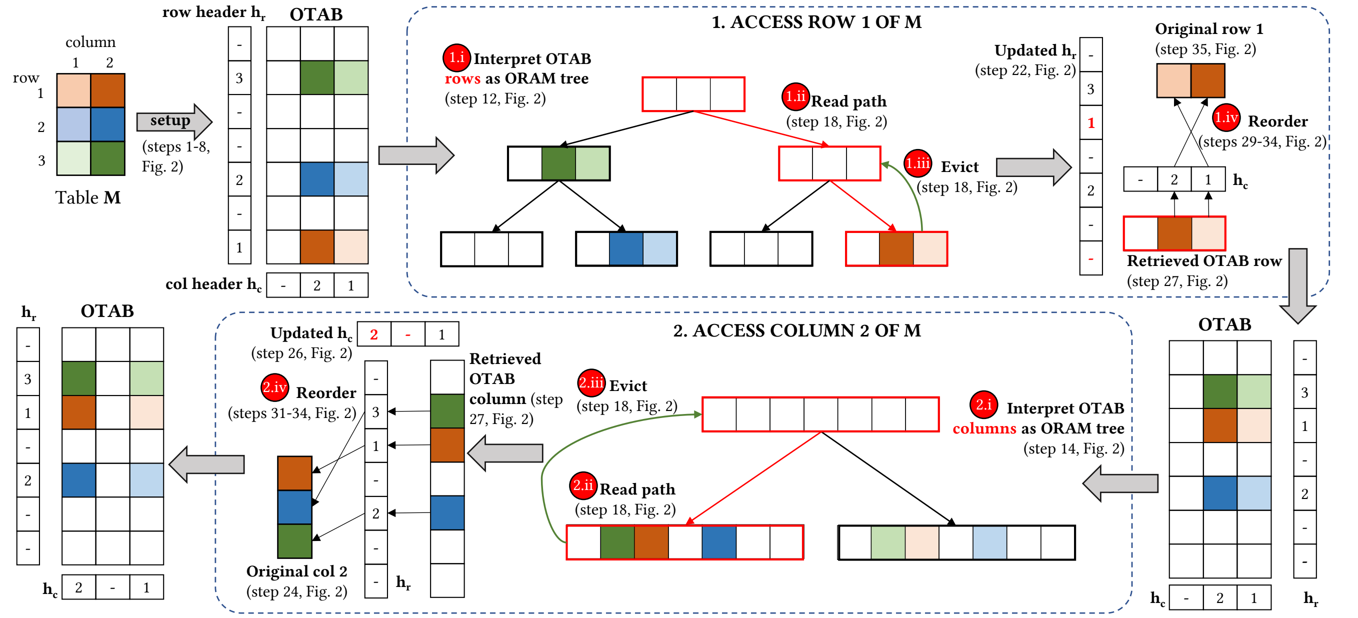

column
1 2
row
1
2
3
Table M
setup
(steps 1-8, Fig. 2)
row header h_r
OTAB
col header h_c
1. ACCESS ROW 1 OF M
1.i Interpret OTAB rows as ORAM tree (step 12, Fig. 2)
1.ii Read path (step 18, Fig. 2)
1.iii Evict (step 18, Fig. 2)
Updated h_r (step 22, Fig. 2)
Original row 1 (step 35, Fig. 2)
1.iv Reorder (steps 29-34, Fig. 2)
h_c
Retrieved OTAB row (step 27, Fig. 2)
2. ACCESS COLUMN 2 OF M
Updated h_c (step 26, Fig. 2)
2.iv Reorder (steps 31-34, Fig. 2)
Retrieved OTAB column (step 27, Fig. 2)
2.iii Evict (step 18, Fig. 2)
2.i Interpret OTAB columns as ORAM tree (step 14, Fig. 2)
2.ii Read path (step 18, Fig. 2)
Original col 2 (step 24, Fig. 2)
h_r
OTAB
h_r
h_c

Figure 12: Toy example of row/column access in OTAB.

To access the first row of the original table **M**, we first interpret the rows in OTAB as the logical ORAM tree (step 1.i). We then execute the Tree-based ORAM access procedure on the interpreted tree, which incurs the path retrieval (step 1.ii) and eviction subroutines (step 1.iii). After executing the Tree-based ORAM access, we obtain a row in OTAB containing the elements of the first row of **M** in a permuted order. In this case, the column header $\mathbf{h}_\mathbf{c}$ is needed to rearrange the permuted elements to obtain the original row (step 1.iv). As some rows in OTAB will be shuffled during the Tree-based ORAM access, the corresponding values in the row header also needs to be updated accordingly. We also show in Figure 12 how a column in **M** is accessed (steps 2.i – 2.iv) following the same principle as row access.

B DEFINITIONS

Forward privacy and backward privacy are the standard security notions in encrypted search schemes. We present the forward and backward privacy definitions in the context of multi-user as follows.

Let $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}}\}$ be the set of stateful leakage functions containing the leakage during setup, search, and update operation, respectively. Let Q be a sequence of all operations and t be the timestamp when an operation occurs. Q records (t, uid, w) for a search on keyword w at time t by user uid, and $(t, \mathsf{uid}, \mathsf{fid}, \{(w_i, \mathsf{op}_i)\}_{i=1}^n)$ for an update on document with identifier fid and a set of updated keywords $\{w_i\}$ at time t with corresponding operations $\{\mathsf{op}_i\}$, where $\mathsf{op}_i \in \{add, del\}$, by user uid. In this setting, we consider each update operation contains all keywords being added or deleted in the updated document, instead of individual keyword-document pairs as prior works.

Let $\mathbf{sp}(w)$ be the search pattern over keyword w consisting of the timestamps of all search queries on w performed by some user.

$$\mathbf{sp}(w) = \{(t, \mathsf{uid}) : (t, \mathsf{uid}, w) \in Q\} \tag{3}$$

Definition 3 (Forward-privacy). *An $\mathcal{L}$-adaptively-secure encrypted search is* forward-private *if the update leakage $\mathcal{L}^{\mathsf{Updt}}$ can be written as*

$$\mathcal{L}^{\mathsf{Updt}}(\mathsf{uid}, \mathsf{fid}, \{w_i, \mathsf{op}_i\}_{i=1}^n) = \mathcal{L}'(\mathsf{uid}, \mathsf{fid}, \{\mathsf{op}_i\}_{i=1}^n)$$

where $\mathcal{L}'$ is stateless.

Definition 4 (Backward-privacy). *Let* $\mathbf{TimeDB}(w)$ *be a function that returns the list of all timestamp/document-identifier pairs of keyword w that have been added to the database by some user* uid *and have not been subsequently deleted by some user* uid′ *as*

$$\mathbf{TimeDB}(w) = \{(t, \mathsf{uid}, \mathsf{fid}) \mid (t, \mathsf{uid}, \mathsf{fid}, (w, add)) \in Q \wedge \\ \forall (t', \mathsf{uid}') : (t', \mathsf{uid}', \mathsf{fid}, (w, del)) \notin Q\}$$

Let $\mathbf{Updates}(w)$ *be a function that returns the timestamp of all insertion and deletion operations for w by some user* uid *in Q.*

$$\mathbf{Updates}(w) = \{t \mid (t, \mathsf{uid}, \mathsf{fid}, (w, add)) \in Q \vee (t, \mathsf{uid}, \mathsf{fid}, (w, del)) \in Q$$

Let $\mathbf{DelHist}(w)$ *be a function that returns the history of deleted entries including which deletion corresponding to which addition.*

$$\mathbf{DelHist}(w) = \{(t^{add}, t^{del}) \mid \exists \mathsf{fid} : (t^{add}, \mathsf{uid}, \mathsf{fid}, (w, add)) \in Q \\ \wedge (u^{del}, \mathsf{uid}, \mathsf{fid}, (w, del)) \in Q\}$$

An $\mathcal{L}$-adaptively-secure encrypted search scheme is backward-private *with different types if the update and search leakage functions $\mathcal{L}^{\mathsf{Udpt}}$ and $\mathcal{L}^{\mathsf{Srch}}$ can be written as one of the following cases:*

- ***(Type-I)**: $\mathcal{L}^{\mathsf{Udpt}}(\mathsf{uid}, \mathsf{fid}, \{w_i, \mathsf{op}_i\}_{i=1}^n) = \mathcal{L}'(\mathsf{uid}, \{op_i\}_{i=1}^n)$ and $\mathcal{L}^{\mathsf{Srch}}(\mathsf{uid}, w_i) = \mathcal{L}''(\mathbf{sp}(w_i), \mathbf{TimeDB}(w_i), a_{w_i})\forall i \in [n]$.*
- ***(Type-II)**: $\mathcal{L}^{\mathsf{Udpt}}(\mathsf{uid}, \mathsf{fid}, \{w_i, op_i\}_{i=1}^n) = \mathcal{L}'(\mathsf{uid}, \{w_i, \mathsf{op}_i\}_{i=1}^n)$ and $\mathcal{L}^{\mathsf{Srch}}(\mathsf{uid}, w_i) = \mathcal{L}''(\mathbf{sp}(w_i), \mathbf{TimeDB}(w_i), \mathbf{Updates}(w_i))$ $\forall i \in [n]$.*
- ***(Type-III)**: $\mathcal{L}^{\mathsf{Udpt}}(\mathsf{uid}, \mathsf{fid}, \{w_i, \mathsf{op}_i\}_{i=1}^n) = \mathcal{L}'(\mathsf{uid}, \{w_i, \mathsf{op}_i\}_{i=1}^n)$ and $\mathcal{L}^{\mathsf{Srch}}(\mathsf{uid}, w) = \mathcal{L}''(\mathbf{sp}(w), \mathbf{TimeDB}(w), \mathbf{DelHist}(w))$.*

Setup. On command $(\mathit{setup}, \mathsf{DB}, N)$, it stores DB.

Data access. On command $(\mathit{access}, \mathsf{op}, \mathsf{id}, B')$, it gets $B \leftarrow \mathsf{DB}[\mathsf{id}]$. If $\mathsf{op} = \mathsf{update}$, it updates the i-th bit in block $\mathsf{DB}[\mathsf{id}]$ with b'_i, where $b'_i \in B'$ is the i-th bit in B' such that $b'_i \neq \perp$. Else if $\mathsf{op} = \mathsf{write}$, it replaces $\mathsf{DB}[\mathsf{id}] \leftarrow B'$. Finally, it returns B.

Figure 13: Ideal functionality $\mathcal{F}_{\mathsf{TreeORAM}}$ of Tree-ORAM.

Setup. The simulator executes the real TreeORAM.Setup protocol honestly with a dummy database DB' (e.g., $\mathsf{DB}' = \{0\}^N$) and sends *setup* command with $\mathcal{Z}$'s chosen database to the ideal functionality $\mathcal{F}_{\mathsf{TreeORAM}}$.

Data access. The simulator executes the real TreeORAM.Access protocol honestly on the dummy block identifier id' (e.g., $\mathsf{id}' = 0$), dummy block data B' (e.g., $B' = \{0\}$), and dummy operation op'. The simulator sends *access* command with $\mathcal{Z}$'s request to the ideal functionality $\mathcal{F}_{\mathsf{TreeORAM}}$ and returns the output of $\mathcal{F}_{\mathsf{TreeORAM}}$ to $\mathcal{Z}$.

Figure 14: Simulator $\mathcal{S}_{\mathsf{TreeORAM}}$ for ideal functionality $\mathcal{F}_{\mathsf{TreeORAM}}$.

Setup. On command $(\mathit{setup}, \mathbf{T})$, where $\mathbf{T} \in \mathbb{F}^{N\times M}$, it stores $\mathbf{T}$

Row access. On command $(\mathit{row\_access}, \mathsf{op}, \mathsf{rid}, \mathbf{u})$, it gets $\hat{\mathbf{u}} \leftarrow \mathbf{T}[\mathsf{rid}, *]$. If $\mathsf{op} = \mathsf{update}$, it updates $\mathbf{T}[\mathsf{rid}, j] \leftarrow \mathbf{u}[j]$ if $\mathbf{u}[j] \neq \perp$ for $1 \leq j \leq N$. Else if $\mathsf{op} = \mathsf{write}$, it replaces $\mathbf{T}[\mathsf{rid}, *] \leftarrow \mathbf{u}$. Finally, it returns $\hat{\mathbf{u}}$.

Column access. On command $(\mathit{col\_access}, \mathsf{op}, \mathsf{cid}, \mathbf{v})$, it gets $\hat{\mathbf{v}} \leftarrow \mathbf{T}[*, \mathsf{cid}]$. If $\mathsf{op} = \mathsf{update}$, it updates $\mathbf{T}[j, \mathsf{cid}] \leftarrow \mathbf{v}[j]$ if $\mathbf{v}[j] \neq \perp$ for $1 \leq j \leq M$. Else if $\mathsf{op} = \mathsf{write}$, it replaces $\mathbf{T}[*, \mathsf{cid}] \leftarrow \mathbf{v}$. Finally, it returns $\hat{\mathbf{v}}$.

Figure 15: Ideal functionality $\mathcal{F}_{\mathsf{OTAB}}$ of oblivious table.

Setup. The simulator executes the real setup protocol honestly with the dummy table $\mathbf{T}'$ and sends *setup* command to ideal functionality $\mathcal{F}_{\mathsf{OTAB}}$.

Row access. The simulator executes the protocol OTAB.Access honestly on the dummy row identifier rid', dummy row data $\mathbf{u}'$, and dummy operation op'. To simulate the inner ORAM access operation, the simulator invokes the Tree-ORAM simulator $\mathcal{S}_{\mathsf{TreeORAM}}$with dummy inputs. The simulator sends *row_access* command with $\mathcal{Z}$'s request to the ideal functionality $\mathcal{F}_{\mathsf{OTAB}}$ and returns the output of $\mathcal{F}_{\mathsf{OTAB}}$ to $\mathcal{Z}$.

Column access. The simulator executes the protocol OTAB.Access honestly on the dummy column identifier cid', dummy column data $\mathbf{v}'$, and dummy operations op'. To simulate the inner ORAM access operation, the simulator invokes the Tree-ORAM simulator $\mathcal{S}_{\mathsf{TreeORAM}}$with dummy inputs. The simulator sends *column_access* command with $\mathcal{Z}$'s request to the ideal functionality $\mathcal{F}_{\mathsf{OTAB}}$ and returns the output of $\mathcal{F}_{\mathsf{OTAB}}$ to $\mathcal{Z}$.

Figure 16: Simulator $\mathcal{S}_{\mathsf{OTAB}}$ for ideal functionality $\mathcal{F}_{\mathsf{OTAB}}$.

Setup. On command $(\mathit{setup}, \mathsf{IDX}, \mathcal{L})$, it stores $(\mathsf{IDX}, \mathcal{L})$.

Keyword Search. On command $(\mathit{search}, w, \mathsf{uid})$, it returns $\mathcal{J} = \{\mathsf{fid} : \mathsf{fid} \in \mathsf{IDX}(w) \wedge \mathcal{L}(\mathsf{uid}, \mathsf{fid}) \in \{\mathsf{r}, \mathsf{rw}\}\}$.

Document Update. On command $(\mathit{document\_update}, \mathsf{fid}, \mathsf{uid}, \mathbf{w})$, it checks if $\mathcal{L}(\mathsf{uid}, \mathsf{fid}) \overset{?}{=} \mathsf{w}$. If this holds, it sets $\mathsf{IDX}(w_i) \leftarrow \mathsf{IDX}(w_i) \cup \{\mathsf{fid}\}$ if w_i is added keyword, otherwise $\mathsf{IDX}(w_i) \leftarrow \mathsf{IDX}(w_i) \setminus \{\mathsf{fid}\}$ if w_i is deleted keyword for each $w_i \in \mathbf{w}$.

Permission Update. On command $(\mathit{permission\_update}, \mathsf{fid}, \mathsf{uid}, p)$, it sets $\mathcal{L}(\mathsf{uid}, \mathsf{fid}) \leftarrow p$.

Figure 17: Ideal functionality $\mathcal{F}_{\mathsf{MSrch}}$ of our MAPLE platform.

where a_{w_i} is the total number of updates on w_i, and $\mathcal{L}', \mathcal{L}''$ are stateless.

Definition 5 (Search and Update Obliviousness). *An $\mathcal{L}$-adaptively-secure encrypted search scheme is search and update oblivious if the search and update leakage functions $\mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}}$ can be written as*

$$\mathcal{L}^{\mathsf{Srch}}(\mathsf{uid}, w) = \mathcal{L}'(\mathsf{uid})$$

$$\mathcal{L}^{\mathsf{Updt}}(\mathsf{uid}, \mathsf{fid}, \{w_i, \mathsf{op}_i\}_{i=1}^n) = \mathcal{L}''(\mathsf{uid})$$

where $\mathcal{L}', \mathcal{L}''$ are stateless.

C PROOFS

C.1 Proof of Lemma 1

Proof. We construct a simulator $\mathcal{S}_{\mathsf{OTAB}}$ such that a PPT environment $\mathcal{Z}$ cannot distinguish between the real and ideal protocol execution. The environment $\mathcal{Z}$ selects inputs for the OTAB client and obtains the corresponding outputs. The environment $\mathcal{Z}$ can corrupt the server storing the oblivious table and takes control over its action. When $\mathcal{Z}$ stops, it outputs a bit indicating whether $\mathcal{Z}$ is interacting with real protocol or simulated protocol. In the simulator $\mathcal{S}_{\mathsf{OTAB}}$, it runs a copy of OTAB protocol online and invokes the Tree-ORAM simulator $\mathcal{S}_{\mathsf{TreeORAM}}$ to simulate the ORAM access. Figure 13 and Figure 14 present the ideal functionality $\mathcal{F}_{\mathsf{oram}}$ and the simulator $\mathcal{S}_{\mathsf{TreeORAM}}$ of Tree-ORAM, respectively. $\mathcal{S}_{\mathsf{OTAB}}$ relays the messages between the client, the server, and the environment $\mathcal{Z}$ such that $\mathcal{Z}$ will see the same interface as when interacting with the real protocol. Figure 16 presents the specification of $\mathcal{S}_{\mathsf{OTAB}}$.

To demonstrate that the simulated and real process are indistinguishable, we show that the view of $\mathcal{Z}$ in the ideal world is indistinguishable from its view in the real process. Note that the view contains the transcript during the protocol execution of the corrupted server and the input/output of the honest client.

We argue that the view for each operation on the oblivious table has the exactly same distribution in the real and in the simulated case: First, in the input phase, the oblivious table being sent to the server looks uniformly random when it is IND-CPA encrypted. Second, during each row/column read/write operation, since OTAB makes use of a secure Tree-ORAM protocol (e.g., [92]), it generates a set of uniformly random and independent values that are simulatable by the Tree-ORAM simulator $\mathcal{S}_{\mathsf{TreeORAM}}$ regardless of which row/column is being accessed. Specifically, these values include the uniformly random logical paths and the encrypted rows/columns.

Now, if the real or simulated protocol proceeds to the last step in each command, the only new data that $\mathcal{Z}$ sees is the output column/row $\mathbf{v}$ requested. In the simulation, $\mathbf{v}$ is obviously the correct column/row requested by $\mathcal{Z}$ since it was read directly from the ideal functionality. To finish the proof, we show that the same happens in the real process with overwhelming probability. In other words, the event that the real protocol terminates but the output is not correct occurs with negligible probability. As discussed, oblivious table employs Circuit-ORAM to access the requested row/column, and thus the integrity depends on the verifiability of the back-end ORAM. In Circuit-ORAM, verifiability is achieved with authenticated encryption, wherein the integrity of all rows/columns being transmitted to the honest client during protocol execution is checked. □

C.2 Proof of Theorem 1

Proof. We construct a simulator $\mathcal{S}_{\mathsf{MAPLE}}$ such that a PPT environment $\mathcal{Z}$ cannot distinguish between the real and ideal protocol execution. The environment $\mathcal{Z}$ selects inputs for the honest parties (which is the MAPLE user and some of the servers in the proxy) and obtains the corresponding outputs. The environment $\mathcal{Z}$ can

Input. The simulator executes the protocol $\mathsf{MAPLE.Setup}$ honestly over the dummy index IDX' of size $k \times N$ and dummy access control list $\mathcal{L}'$ of size $t \times N$ and sends *setup* command to the ideal functionality $\mathcal{F}_{MAPLE}$.

Keyword Search. The simulator executes the protocol $\mathsf{MAPLE.KSearch}$ honestly on the dummy keyword w' and the dummy user identifier uid'. The simulator then sends *keyword_search* command to the ideal functionality $\mathcal{F}_{MAPLE}$, and returns the output of $\mathcal{F}_{\mathsf{OTAB}}$.

Document Update. The simulator executes the protocol $\mathsf{MAPLE.DUpdate}$ honestly on the dummy document identifier fid', the dummy user identifier uid' and the dummy updated keywords $\mathbf{w}'$. The simulator sends *document_update* command to the ideal functionality $\mathcal{F}_{\mathsf{MSrch}}$.

Permission Update. The simulator executes the protocol $\mathsf{MAPLE.PUpdate}$ honestly on the dummy document identifier fid', the dummy user identifier uid' and the dummy permission p'. The simulator sends *permission_update* command to the ideal functionality $\mathcal{F}_{\mathsf{MSrch}}$.

Figure 18: Simulator $\mathcal{S}_{\mathsf{MAPLE}}$ for $\mathcal{F}_{\mathsf{MSrch}}$.

corrupt up to t servers in the proxy and takes control over their action. When $\mathcal{Z}$ stops, it outputs a bit indicating whether $\mathcal{Z}$ is interacting with real protocol or simulated protocol. The simulator runs a copy of MAPLE protocol online. It relays the messages between the parties (client and server) and $\mathcal{Z}$ such that $\mathcal{Z}$ will see the same interface as when interacting with the real protocol. Figure 18 presents the specification of $\mathcal{S}_{\mathsf{MAPLE}}$.

We show that the view of $\mathcal{Z}$ in the ideal world is indistinguishable from the view in the real process. The view contains the transcript of the protocol execution of the corrupted servers and the input/output of the honest user. We argue that the view for each operation has the exactly same distribution in the real and in the simulated case: First, in the setup phase, the search index and the access control as well as their auxiliary components (e.g., row/column header and permutation matrix) always look uniformly random due to secret sharing invoked in $\Pi_{\mathsf{mpc}}.\mathsf{Input}(\cdot)$ protocol. Second, during each keyword search or document update operation, due to the security of oblivious table, the honest parties will always send a set of uniformly random and independent values regardless of what keyword/document is being search/updated. Specifically, in the keyword search, the proxy emulates oblivious accesses over the column dimension of the oblivious tables representing the index and access control, and then performs exclusive-AND operations to filter unauthorized documents and obtain the final search results. As shown in §5.3 and §5.4.2, all these processings are realized with MPC boolean operation techniques in MAPLE, which were proven to yield random and independent values during protocol execution [95]. In the document update, the proxy performs matrix multiplication for keyword permutation followed by emulating oblivious table accesses on the search index and the access control, all of which are realized by MPC boolean protocols that yield uniformly and independent values during protocol execution.

Now, if the real or simulated protocol proceeds to the last step in each command, the only new data that $\mathcal{Z}$ sees is the output of the search result $\mathcal{D}$. In the simulation, $\mathcal{D}$ is obviously the correct column/row data requested by $\mathcal{Z}$ since it was read directly from the ideal functionality. We show that the same happens in the real process with overwhelming probability. In other words, the event that the real protocol terminates but the output is not correct occurs with negligible probability.

In our real protocol, the integrity of the search result depends on four factors: (*i*) the integrity of the MPC input protocol to distribute the search request to the proxy, (*iii*) the integrity of MPC computation during search operation at the proxy, (*ii*) the integrity of previous document update operations (if any), and (*iv*) the integrity of MPC output protocol. Since MAPLE harnesses authenticated MPC, the integrity of MPC computation during search and update is achieved. It remains to show that the MPC input and output protocols offer integrity for the values being transmitted to or received from the proxy. We can see that $\Pi_{\mathsf{mpc}}.\mathsf{Input}(\cdot)$ protocol indeed relies on $\Pi_{\mathsf{mpc}}.\mathsf{Output}(\cdot)$ protocol to broadcast a reliable randomness for the user to distribute the masked secrets to the proxy. Thus, as long as $\Pi_{\mathsf{mpc}}.\mathsf{Output}(\cdot)$ protocol offers integrity with overwhelming probability, the integrity of the search protocol in our scheme will be achieved. We can see that $\Pi_{\mathsf{mpc}}.\mathsf{Output}(\cdot)$ protocol relies on probabilistic polynomial identity testing technique to check the integrity of secret values being privately opened to the honest user from their shares. In this technique, the honest user requests the proxy to evaluate an n-degree random polynomial representing share of n secrets revealed to the user at a random evaluation point. On the other hand, the honest user herself evaluates another n-degree polynomial representing n secrets opened by the proxy at the same evaluation point. Due to Schwartz-Zippel lemma [81, 100], the probability these two polynomials are different but their evaluation is the same is $n/|\mathbb{F}|$. This implies the secrets revealed to the honest user are consistent with their authenticated shares at the proxy with overwhelming probability, thereby enabling integrity in our search protocol. □

D MPC PROTOCOLS

D.1 Secure Input/Output for MPC Proxy

We present the secure input and output protocols proposed in [27] for the user to interact with the MPC proxy. The main idea is to rely on the polynomial identity testing applied in zero-knowledge proofs as well as random masking techniques using correlated randomness. We denote $[\![x]\!]_i$ as the share of the secret x to party $\mathcal{P}_i$.

- $\underline{(v_1, \ldots, v_n) \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Output}(\langle v_1 \rangle, \ldots, \langle v_n \rangle)}$: Given n authenticated shares $\langle v_1 \rangle, \ldots, \langle v_n \rangle$ from the MPC proxy, each sever $\mathcal{S}_i$ in the proxy individually samples a random $[\![r]\!]_i \overset{\$}{\leftarrow} \mathbb{F}$ and sends it along with $([\![v_1]\!]_i, \ldots, [\![v_n]\!]_i)$ to the user. The user computes $v' \leftarrow \sum_i [\![v]\!]_i$ and $r' \leftarrow \sum_i [\![r]\!]_i$. The user forms a polynomial $f'(X) = r' + \sum_{i=1}^{n} v'_i \cdot X^i$, while the MPC proxy evaluates the polynomial $\langle f'(\alpha) \rangle = \langle r \rangle + \sum_{i=1}^{n} \langle v \rangle_i \cdot X^i$ at a random point $\alpha \overset{\$}{\leftarrow} \mathbb{F}$ chosen by the user. The MPC proxy publicly opens $f(\alpha)$ (with verifiability by the underlying MPC protocol). If $f(\alpha) = f'(\alpha)$, the user outputs and accepts $(v'_1, \ldots, v'_n)$ as the correct secrets corresponding to $\langle v_1 \rangle, \ldots, \langle v_n \rangle$. Otherwise, the user aborts.
- $\underline{(\langle v \rangle_1, \ldots, \langle v \rangle_n) \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Input}(v_1, \ldots, v_n)}$: Given n secrets $(v_1, \ldots, v_n) \in \mathbb{F}^n$ from the user, the user first receives n random values from the MPC proxy by executing $(r_1, \ldots, r_n) \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{Output}(\langle r_1 \rangle, \ldots, \langle r_n \rangle)$, where $\langle r_i \rangle$ is the authenticated share of the random r_i generated during preprocessing. The user broadcasts $v'_i \leftarrow v_i - r_i$ to the MPC proxy, which, in turn, obtains the authenticated shares by computing $\langle v_i \rangle \leftarrow v'_i + \langle r_i \rangle$ for $i \in [n]$.

```
$\langle b \rangle \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{cmp}(\langle x \rangle, \langle y \rangle)$
Parse $\langle x \rangle$ as $\langle x_0 x_1 \dots x_{n-1} \rangle$, $\langle y \rangle$ as $\langle y_0 y_1 \dots y_{n-1} \rangle$   ▷ binary representation
for $i = 0$ to $n-1$ do
    $\langle c_i \rangle \leftarrow \langle x_i \rangle \boxplus \langle y_i \rangle$
$\langle b \rangle \leftarrow (1 \boxplus \langle c_0 \rangle) \boxtimes (1 \boxplus \langle c_1 \rangle) \boxtimes \dots (1 \boxplus \langle c_{n-1} \rangle)$
return $\langle b \rangle$

$\langle c \rangle \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{sel}(\langle b \rangle, \langle x \rangle, \langle y \rangle)$ :
Parse $\langle x \rangle$ as $\langle x_0 x_1 \dots x_{n-1} \rangle$, $\langle y \rangle$ as $\langle y_0 y_1 \dots y_{n-1} \rangle$   ▷ binary representation
for $i = 0$ to $n-1$ do
    $\langle z_i \rangle \leftarrow \big( (\langle x_i \rangle \boxplus \langle y_i \rangle) \boxtimes \langle b \rangle \big) \boxplus \langle x_i \rangle$
return $\langle z \rangle \leftarrow \langle z_0 z_1 \dots z_{n-1} \rangle$

$\langle b \rangle \leftarrow \Pi_{\mathsf{mpc}}.\mathsf{gt}(\langle x \rangle, \langle y \rangle)$
Parse $\langle x \rangle$ as $\langle x_0 x_1 \dots x_{n-1} \rangle$, $\langle y \rangle$ as $\langle y_0 y_1 \dots y_{n-1} \rangle$   ▷ binary representation
for $i = 0$ to $n-1$ do
    $\langle c_i \rangle \leftarrow \langle x_i \rangle \boxtimes (1 \boxplus \langle y_i \rangle)$
$\langle b \rangle \leftarrow 1 \boxplus (1 \boxplus \langle c_0 \rangle) \boxtimes (1 \boxplus \langle c_1 \rangle) \boxtimes \cdots \boxtimes (1 \boxplus \langle c_{n-1} \rangle)$
return $\langle b \rangle$
```

Figure 19: Secure boolean operations.

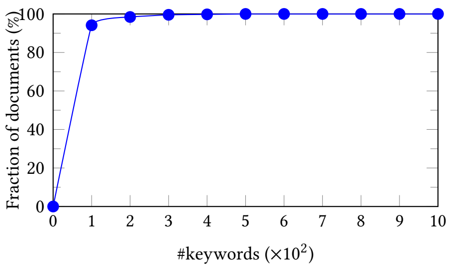


Figure 20: Keyword distribution of our enwiki subdataset.

Note that the above interactive input/output protocols can be made non-interactive via Fiat-Shamir transformation [40], in which the user's random challenge can be derived directly by hashing the current protocol transcripts.

D.2 Secure Boolean Computation

We present the algorithmic details of some secure boolean operations that are used in our platform in Figure 19. Note that in binary field, ⊠ and ⊞ are equivalent to secure bit-wise AND and XOR operations, respectively.

E ADDITIONAL EXPERIMENTS

Figure 20 presents the keyword distribution of our testing set, where the y-axis shows the proportion of articles that have less than a number of keywords on the x-axis.

E.1 Storage Cost

We report the storage cost of MAPLE as follows. OTAB incurs 4× expansion to the original tables (search index, access control) to support both oblivious search and update efficiently. Moreover, for malicious security, each bit of OTAB is attached with a MAC component consisting of the share of the MAC key and MAC value. This results in the total size of 18.2 GB to 2.2 TB to store all the necessary data structures in MAPLE. Since this size is large, we applied k-top caching strategy to partially cache them on memory for efficient processing.

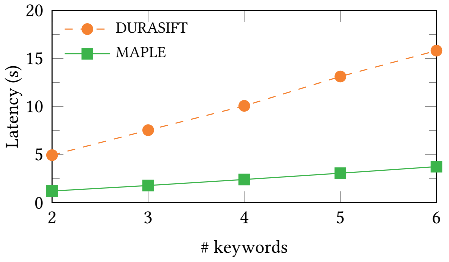


Figure 21: Multi-keyword search delay (64K documents).

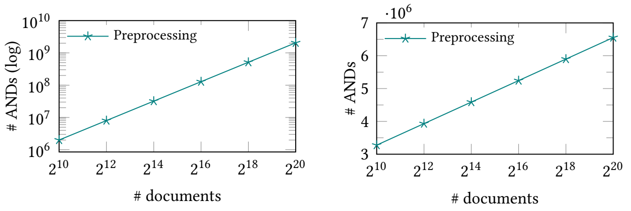
MAC components are the most dominant factor in the total storage overhead. Therefore, in our future work, we will explore several strategies to reduce the size of the MAC components such as aggregate MAC [63]. We will investigate how to evaluate such aggregate MAC with MPC (similar to running encryption/decryption circuits in 2PC in [60]). We expect this strategy will decrease the storage overhead down to 9 MB-8.5 GB (259×-2063× smaller) at the cost of incurring 33.7%–47.9% more overhead in the search. We leave this investigation as our future work.

E.2 Multi-keyword Search

We compared the performance of our scheme in handling multi-keyword search with DURASIFT [39]. Figure 21 presents the end-to-end search delay of our scheme and DURASIFT for n-keyword queries where $n \in [6]$. In this experiment, we repeat the single-keyword search for each term in the multi-keyword query and then perform corresponding secure boolean operations to obtain final result. MAPLE takes about 1.2s to process a boolean query with two keywords compared with 4.9s by DURASIFT. The search delay of both schemes grows almost linearly with the number of keywords in the search query but by varying degrees. With six terms in a query, MAPLE takes roughly 3.7s while DURASIFT takes around 15.8s. We can see that MAPLE is more efficient than DURASIFT (i.e., 3.8×–4.3× faster) in all cases mostly due to the use of Bloom filter for compact keyword representation. Moreover, by using oblivious table to instantiate the search index, our search time is only *logarithmic* to the small size of keyword representation (i.e., 16K), whereas DURASIFT is *linear* to the large size of the keyword universe (i.e., 1.9M).

E.3 Preprocessing Cost

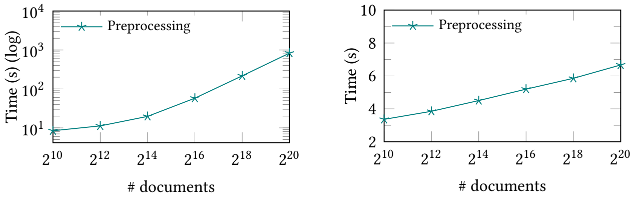
We present the preprocessing costs to generate necessary materials in the offline phase that will be used in the online keyword search and document update evaluation in Figure 22 and Figure 23. Specifically, Figure 22 shows the number of AND gates needed for each search and update, which dominates most of the preprocessing cost. Figure 23 illustrates the corresponding latency to prepare circuits for AGC evaluation in both search and update in the setting of two servers. For the number of documents ranging from 2^{10} to 2^{20}, the number of AND gates for each search grows from 2M to 2B, thereby the latency of preprocessing for each search operation increases from 8.4s to 832.3s, respectively. Because AGC circuits



(a) Keyword search

(b) Document update

Figure 22: The number of AND gates.



(a) Keyword search

(b) Document update

Figure 23: Preprocessing time.

in update are smaller than circuits in search, and each update just requires one OTAB access operation instead of $k = 7$ OTAB access operations on the search index as keyword search, the offline phase for an update prepares a significantly smaller number of AND gates than for a search, which grows from 3.3M to 6.6M, corresponding to the latency rising from 3.4s to 6.7s for the number of documents increasing from 1K to 1M, respectively. As the preprocessing phase is independent with search and update queries, the precomputed triples of AGC can be stored on storage device (*e.g.*, SSD), and later reloaded to the RAM memory for evaluating in the online phase.