Tokenised Multi-client Provisioning of Searchable Encryption with Forward and Backward Privacy

Blinded for review

No Institute Given

Abstract. Searchable Symmetric Encryption (SSE) has opened up an attractive avenue for privacy-preserved processing of outsourced data on the untrusted cloud infrastructure. SSE aims to support efficient Boolean query processing with optimal storage and search overhead over large real databases. However, current constructions in the literature lack the support for multi-client search and dynamic updates to the encrypted databases, which are essential requirements for the widespread deployment of SSE on real cloud infrastructures. Trivially extending a state-of-the-art single client dynamic construction, such as ODXT (Patranabis et al., NDSS'21), incurs significant leakage that renders such extension insecure in practice. Currently, no SSE construction in the literature offers multi-client query processing and search with dynamic updates over large real databases while maintaining a benign leakage profile.

This work presents the first dynamic multi-client SSE scheme Nomos supporting efficient multi-client conjunctive Boolean queries over an encrypted database. Precisely, Nomos is multi-reader-single-writer construction that allows only the gate-keeper (or data-owner) - a trusted entity in the Nomos framework, to update the encrypted database stored on the adversarial server. Nomos achieves forward and backward privacy of dynamic SSE constructions while incurring lesser leakage than the trivial extension of ODXT to a multi-client setting. Furthermore, our construction is practically efficient and scalable - attaining linear encrypted storage and sublinear search overhead for conjunctive Boolean queries. We provide an experimental evaluation of software implementation over an extensive real dataset containing millions of records. The results show that Nomos performance is comparable to state-of-the-art static conjunctive SSE schemes in practice.

1 Introduction

Recent advancements in cloud computing have fuelled the development of privacy-preserved processing of sensitive data on third-party cloud servers. Outsourced processing and storing of users' data are becoming standard practices for individuals and organisations. Presently, cloud infrastructures are responsible for handling users' private data obtained from devices/systems used by ordinary citizens, government and industrial establishments. For extended functionalities, the cloud service providers often delegate access to users' data to third-party entities. The involvement of the cloud service providers and other third-party entities - all of whom are considered untrusted, raises deep concern about users' data confidentiality and information privacy. Furthermore, modern cloud applications serve multiple clients, and the data stored on the cloud is frequently updated. In this context, straightforward encryption that provides high confidentiality of data trivially loses the ability to process information in the encrypted form, defeating the advantage of using the cloud.

State-of-the-art cryptographic paradigms such as Fully Homomorphic Encryption (FHE) [17,18], Functional Encryption (FE) [1], Oblivious RAM (ORAM) [21], Searchable Symmetric Encryption (SSE) [13,31] and Multi-party Computation (MPC) [20] allow implementing diverse functionalities over encrypted data, including search, analytics, and fine-grained access control. However, apart from SSE, all of the aforementioned approaches either incur prohibitively heavy computation overhead or require extremely high communication bandwidth for real applications. In contrast, SSE offers theoretically robust and implementation-efficient constructions for encrypted data processing, especially searching, while leaking only benign information to the untrusted parties. The benign leakage in SSE is formalised and quantified using precise leakage functions that capture the information leaked. We elaborate more on the general SSE setting and construction below.

1.1 Searchable Symmetric Encryption

Searchable Symmetric Encryption (SSE) [2,4,7–10,12–14,16,19,23,24,26,28,30,31] allows querying an encrypted database privately without decryption. Fundamentally, an SSE scheme offers the following capabilities.

- Allow an (or many) entity (potentially untrusted client) to efficiently search encrypted queries over encrypted database stored on the cloud (untrusted) without revealing the result to the server.
- Minimise the leakage during query (or update) such that the untrusted entities receive only benign information.

Typical benign leakages include crude statistical information related to the database elements, the query, or the result of the query - but do not include actual associated (encrypted) data. The database size, *query pattern* (the set of queries corresponding to the same keyword), the *access pattern* (the set of database records matching a given query) are few such leakages typically studied in the context of SSE. We present formal syntax of SSE with elaborate details in Section 2, and study these leakages in Section 6.

Dynamic SSE. A dynamic SSE construction [3,5,6,11,15,25,27] allows dynamic updates (adding or deleting data elements) to the encrypted database offloaded to the cloud by the client. In contrast, static constructions do not allow updates to the database once it is encrypted. The update capability of dynamic constructions implies two security notions - *forward* privacy and *backward* privacy. Informally, the forward privacy notion states that a current search operation can not be linked to a future update operation, and backward privacy dictates that an insertion operation followed by a deletion does not reveal any information in a future search operation. These two notions are essential for dynamic schemes to prevent a certain class of attacks, specifically, the file injection attack [32].

Multi-client SSE. A multi-client SSE scheme allows multiple clients to search (or update) the encrypted database. SSE schemes can be classified in the following way according to different entities involved in the setting.

Single-Reader–Single-Writer (SRSW): Single-reader–single-writer or SRSW setting has a single client and an untrusted cloud server. The single client also acts as the data owner who has permission to update the encrypted database on the server.

Multi-Reader–Single-Writer (MRSW): In the multi-reader–single-writer or MRSW setting, multiple clients interact with the untrusted server to search (with individual trapdoors), and a single data owner can generate or update the encrypted database on the untrusted server.

Multi-Reader–Multi-Writer (MRMW): Multi-reader-multi-reader or MRMW is the most generic setting allowing multiple clients to search and update the encrypted database on the cloud server.

SRSW constructions [7, 8, 10, 12, 13, 16, 19, 23, 24, 26, 28, 31] have been extensively studied in the literature, especially in the static setting. A number of dynamic schemes [3, 5, 6, 11, 15, 25, 27] have been proposed in recent years and ODXT by Patranabis et al. [30] is the only state-of-the-art dynamic scheme with conjunctive query support. Essentially, cloud applications require multi-client access along with dynamic update capability to cater a large number of users (or clients), which prompts us to raise the following question.

Is there an existing SSE scheme that supports multi-client search, supports dynamic updates to the encrypted database, has strong forward and backward privacy, and has linear encrypted storage and sublinear search overhead – all collectively?

As it turns out, the answer is *no.* ODXT supports dynamic updates and conjunctive queries with sublinear search overhead and linear encrypted storage. However, it is an SRSW construction without support for multiple clients. Furthermore, ODXT is vulnerable to a particular leakage originating from the cross-terms in a conjunctive query (discussed later) that can lead to complete query recovery. Trivially extending ODXT to the multi-client setting by delegating the search token generation phase from multiple clients (can be malicious) to the data-owner (a trusted party) retains this leakage and we outline an attack process based on this leakage in Section 3 that leads to complete recovery of the cross-terms. In brief, the existing SSE literature lacks dynamic SSE schemes in MRSW and MRMW settings, which hinders widespread adoption of SSE in encrypted processing tasks on the cloud.

In this work, we aim to bridge this gap between secure and practically efficient SSE constructions and real multi-client cloud applications. We summarise our goal by asking the following question.

Can we design an efficient dynamic SSE scheme with forward and backward privacy in the multi-reader-single-writer setting?

We show in this paper that it is possible to design such a scheme and we present Nomos[1] construction that achieves the aforementioned practical design and security goals. The following subsection lists the primary contributions of this work. We emphasise that Nomos is a dynamic multi-keyword construction in the MRSW setting, which is a stepping stone towards building "ideal" MRMW SSE constructions. Extending Nomos to the MRMW setting is of independent interest requiring separate in-depth exploration and we leave this as a future work.

1.2 Our Contributions

We summarise our main contributions of this work with brief overview below.

❶ **Multi-client SSE**. We present the first multi-client dynamically updatable SSE construction Nomos for outsourced encrypted databases. Nomos supports efficient multi-keyword conjunctive Boolean queries in the MRSW setting, which is essential for practical cloud applications. To the best of our knowledge, this is the first scheme in the literature that can process conjunctive queries from multiple clients with dynamic updates. Clients in Nomos obtains search tokens from a trusted entity called gate-keeper, who is allowed to update the encrypted database and holds the keys for token generation. The clients use the search tokens obtained from gate-keeper to query over the encrypted database on the cloud server. We use Oblivious Pseudo-random Function (OPRF)-based mechanism to delegate the search token generation process to the gate-keeper; thus bypassing the need to share the keys for token generation among multiple potentially malicious clients.

❷ **Leakage Mitigation**. The Nomos construction mitigates a particular leakage originating from the cross-terms of a conjunctive query. This leakage is inherently present in state-of-the-art ODXT scheme and we discuss an attack flow that show that trivial extensions of ODXT to the multi-client setting remains vulnerable to this leakage which can potentially break the scheme completely. Nomos avoids this specific cross-term–based leakage exploitable from XSet accesses by introducing decorrelated XSet access pattern. We use a variant of Bloom Filter (BF), denoted as the Redundant Bloom Filter (RBF), for decorrelating repeated memory accesses to eliminate cross-term leakage. Using RBF has minimal impact on the storage, communication and computation overhead of Nomos compared to non-BF and plain BF versions of our construction.

❸ **Tokenised search**. Nomos allows each client to obtain search tokens from gate-keeper (the data owner holding the keys for token generation) individually and engage in the search protocol with the server to retrieve the query result.

[1] In Greek mythology, Nomos is the spirit of law. In a way, Nomos (SSE) aims to maintain lawful conduct of multiple clients and other entities involved in processing sensitive data.

The token generation process and search phase work asynchronously, although the search phase requires the tokens to be generated first through the token generation protocol. This tokenised search process for multiple clients allows to avoid a three-party search protocol involving a client, the gate-keeper and the server, without blocking other clients from invoking the token generation or the search protocol (whichever is available). This is a desirable capability in multi-client cloud applications where requests arrive asynchronously and the gate-keeper/cloud needs to serve as early as possible (provide example) reducing waiting time.

❹ **Security analysis and implementation**. We provide elaborate security proofs for NOMOS using hybrid arguments of indistinguishability framework. NOMOS setting assumes that gate-keeper (or the data owner) is a trusted entity, and the cloud server is an honest-but-curious polynomial-time adversarial entity. The clients are assumed to be malicious entities individually, that is, a client can try to obtain additional information from the actual execution by providing modified tokens. We provide an overview of NOMOS security in Section 6 and elaborate security analysis in Appendix A.

We implemented the NOMOS framework using C++ (natively multi-threaded) with Redis[2] as the database back-end. We used the Enron dataset[3] to evaluate NOMOS performance and we report the results in Section 7. The experimental results show that NOMOS achieves linear storage overhead and sublinear search overhead, comparable to other state-of-the-art conjunctive SSE constructions.

We provide preliminary notations and syntax in Section 2 which we follow throughout the manuscript. We present an attack in Section 3 on the trivial extension of ODXT to multi-client setting to demonstrate the devastating effect of cross-term leakage on a multi-client SSE construction. We outline required security notions and challenges of designing multi-client SSE in Section 4. We present our main NOMOS construction in Section 5 and security analysis (informal) of NOMOS in Section 6. Finally, the experimental results and related discussion are available in Section 7, and we end with a concluding remark on our work.

2 Basic Notations and Syntax

We provide the basic notations and syntax of SSE below, which we follow throughout this paper. For ease of exposition, we assume the database to be a document collection indexed by keywords and unique document identifiers. More precisely, we assume an inverted index of keywords and corresponding document identifiers for a document collection is available as the plain database.

[2] https://redis.io/

[3] https://www.cs.cmu.edu/ enron/

2.1 Basic Notations

Data. We use w to represent a keyword and id to represent a document identifier in a database. We use Δ to denote the dictionary of all ws in a database. We represent the plain database as $\mathbf{DB}$, and $\mathbf{DB}(q)$ represents the set of ids returned upon querying a conjunctive query q over $\mathbf{DB}$. Similarly, for a single w, $\mathbf{DB}(\mathsf{w})$ is the set of ids where w appears. The number of ids in $\mathbf{DB}(q)$ is represented as $|\mathbf{DB}(q)|$. For two values (or strings) v_1 and v_2, $v_1||v_2$ represents the concatenation of v_1 and v_2. The cardinality of a set S is denoted by $|S|$, and for a string s (or vector), $|s|$ represents the length of s. We represent the sequence $m, m+1, \ldots, n-1, n$ using $[m, n]$ and $1, 2, \ldots, n$ using $[n]$. Sampling a value v from a distribution $\mathcal{D}$ is expressed as $v \xleftarrow{\$} \mathcal{D}$. We denote a negligible function as $\mathsf{negl}()$. We denote the attribute of a w using $I(\mathsf{w})$. We represent a set of valid keyword attribute combinations using $\mathcal{P}$ from d unique keyword attributes for ws in Δ. We also assume that during an update a complete document (containing multiple keywords) is updated and for that existing records are deleted and added again with the modified content. In that case, update operations are usually done in batches of deletions followed by additions of multiple (w,id) pairs.

Entities. We use $\mathcal{C}$ to represent a client and $\mathbf{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_t\}$ to represent a set of t clients. We denote the server using $\mathcal{S}$ and the gate-keeper using $\mathcal{G}$ (explained in Section 5). We denote a polynomial time adversary using $\mathcal{A}$ and a simulator using SIM.

SSE Algorithms. We denote an SSE algorithm using SSE and the associated setup, update and search using Setup, Update, and Search, respectively. We use $\mathbf{EDB}$ to denote the encrypted database and $\mathcal{R}_q$ to denote result set obtained through Search routine upon querying a conjunctive query q, compactly expressed as $\mathcal{R}_q = \mathbf{EDB}(q)$. We denote a conjunctive query as $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$ where $\mathsf{w}_i \in \Delta$. Without loss of generality, we assume w_1 in q has the least frequency of updates or occurrence. We call w_1 in q as the special-term or s-term and $\mathsf{w}_2, \ldots, \mathsf{w}_n$ are denoted as the cross-terms or x-terms.

The Setup routine in SSE is a probabilistic polynomial-time (PPT) algorithm that initialises $\mathbf{EDB}$ and generates the master secret key sk. The data-owner $\mathcal{D}$ invokes the PPT algorithm Update with a $(\mathsf{w}, \mathsf{id})$ pair and an update type $\mathsf{op} = \{\mathsf{ADD},\mathsf{DEL}\}$ and interacts with $\mathcal{S}$ to update $\mathbf{EDB}$. A client $\mathcal{C}$ needs to invoke the PPT algorithm GenToken and interact with $\mathcal{D}$ to obtain the search tokens. Search is a deterministic algorithm where $\mathcal{C}$ and $\mathcal{S}$ engage in a two-party protocol where $\mathcal{C}$ provides the search tokens obtained using GenToken, and $\mathcal{S}$ looks-up the corresponding ids from $\mathbf{EDB}$ that are returned as the result.

Cryptographic Primitives. We denote a pseudo-random function by PRF, and a CPA-secure symmetric-key encryption scheme as SKE. We assume that SKE has $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ as its subroutines for secret key generation, encryption and decryption, respectively.

We represent a collision resistant hash function as CRHF or with the symbol H which we assume can be modelled as a random oracle. Additionally, we use an authenticated encryption (AE) [**?**,**?**] scheme with the routines $\{\mathsf{AuthEnc},\mathsf{AuthDec}\}$ that is IND-CPA and and strongly UF-CMA-secure (unforgablility guarantee) [**?**].

Decisional Diffie-Hellman Assumption. Let $\mathbb{G}$ be a cyclic group of prime order q, and let g be any uniformly sampled generator for $\mathbb{G}$. The decisional Diffie-Hellman (DDH) assumption is that for all PPT algorithms $\mathcal{A}$, we have,

$$\left| Pr[\mathcal{A}(g,g^{\alpha},g^{\beta},g^{\alpha\cdot\beta})=1]-Pr[\mathcal{A}(g,g^{\alpha},g^{\beta},g^{\gamma})=1] \right| \leq \mathsf{negl}(\lambda),$$

where $\alpha,\beta,\gamma \xleftarrow{\$} \mathbb{Z}_p^*$.

Oblivious Pseudo-random Function. Oblivious Pseudo-random Function (OPRF) is public-key primitive that allows two parties to jointly evaluate a PRF where party A provides the input plaintext x and party B inputs the key $K \in \mathbb{Z}_p^*$. At the end of protocol, A receives the output t which is indistinguishable from a regular PRF evaluation with the same x and K, and party B receives nothing (or error/nothing symbol $\perp$).

Hashed Diffie-Hellman OPRF. We use a specific instance of OPRF called hashed Diffie-Hellman (DH) OPRF that works as follows - party A provides input x and a randomly sampled value r. A uses a hash function H which hashes an input to a group element of $\mathbb{G}$. A uses H to obtain $H(x) \in \mathbb{G}$ and raises $H(x)$ to generate $s = H(x)^r$. A sends s to B, and B raises s by power K to obtain $t = s^K$. B sends back t to A and A outputs t. DH-OPRF is used as a core primitive in our construction to allow multi-client search. Please refer [29] for more details on DH-OPRF.

SSE Data Structures. SSE constructions heavily rely on the underlying data structures to store and efficiently search over encrypted data. In this work, we consider two widely used SSE-specific data structures, namely TSet and XSet. Throughout the manuscript we assume **EDB** to contain both TSet and XSet (as required by the construction discussed later in Section 5).

TSet. TSet is an encrypted version of a multi-map data structure that stores the encrypted database in a structured form. Fundamentally, TSet stores and accesses data elements in a uniformly indistinguishable manner that hides the association of an w with respective ids. At a high-level, TSet follows the typical syntax of a multi-map.

- Insertion: TSet[key] = val
- Retrieval: val = TSet[key]

The key and corresponding val are generated through PRFs (and SKE) such that the probability of an $\mathcal{A}$ distinguishing two different ws from randomly accessed (key,val) pairs is negligible.

XSet. XSet is a data structure typically used in multi-keyword SSE schemes supporting conjunctive query search (especially in the cross-tag family of constructions [8, 28, 30]). XSet stores the cross-term–related information which is used during conjunctive query. Note that XSet does not store any encrypted information about individual ws or ids; rather, it stores flags or bits associated with cross-term generated using CRHF or PRFs. At a high level, an XSet has the following syntax,

- Insertion: $\mathsf{XSet}[\mathsf{index}] = b,\ \ b \in \{0,1\}$
- Retrieval: $b = \mathsf{XSet}[\mathsf{index}]$

where the index is typically generated from a combined input of w and id to a CRHF and $b = 1$ indicates that (w,id) is valid pair (that is w appears in document id). For detailed properties and analysis of TSet and XSet, please refer [8]. We use a slightly different variant of TSet as adopted in ODXT [30].

We choose ODXT as our base construction for developing the multi-client solution as ODXT is the state-of-the-art conjunctive dynamic scheme with efficient update and search. Unfortunately, ODXT itself does not support multi-client search. We first transform ODXT into a multi-client construction following the approach of [22] (presented in Appendix D). However, we show that MC-ODXT is vulnerable to cross-term based leakage, and the following attack exploits this leakage to break the scheme.

3 Attack on Multi-client SSE Exploiting Cross-term Leakage

We outline an attack on the trivial extension of ODXT to the multi-client setting or MC-ODXT following the multi-client provisioning in [22]. This attack demonstrates that presence of the same cross-terms in different queries can lead to a severe leakage exploitable from XSet access pattern that completely breaks the security of the scheme. More specifically, we show that in a setting where a malicious client colluding with adversarial server can exactly recover the queries issued by a legitimate client. Given enough query instances by the malicious client, it can recover the complete keyword dictionary. Clearly, this glaring vulnerability puts MC-ODXT at risk of serious data breach. Our main construction Nomos prevents this leakage by adopting a "light-weight" redundancy-based database access mechanism without affecting the performance or storage overhead of the base construction in practice.

We briefly summarise the workflow of MC-ODXT following the MC-ODXT algorithms presented in Appendix D to identify the source of the leakage and after that discuss the attack exploiting this leakage. MC-ODXT workflow below involves a client $\mathcal{C}$ obtaining search token for a conjunctive query $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$ from the data owner $\mathcal{D}$ and querying the server $\mathcal{S}$ using those search tokens.

MC-ODXT Workflow. MC-ODXT follows the ODXT structure with the Setup, Update, Search routines and an additional routine GenToken for query token generation. The Setup routine sets up and initialises the parameters, data structures and generates the secret keys. $\mathcal{D}$ repeatedly invokes Update for **DB** entries to update **EDB** on $\mathcal{S}$. The Update algorithm generates a unique TSet address addr by appending a counter value with w and evaluating the resulting value through a PRF. Since this counter value is incremented with each update, an addr is never repeated (in other words, a addr is unique to an update invocation). Furthermore, the update routine treats an ADD or DEL op identically as there is no conditional execution based on ADD or DEL. The id is encrypted, and a w-specific deblinding token (α) is generated to use in DH-based blinded oblivious computation later during search. $\mathcal{D}$ sends the encrypted id with the deblinding token to store in the XSet at address addr. Additionally, $\mathcal{D}$ generates an xtag by combining w with id (concatenated with op) through PRF and raising to the power of g (such that during search, xtag can be recomputed obliviously using the query tokens and α). $\mathcal{D}$ sends xtag to the server and the server sets a bit 1 at address xtag in XSet.

Prior to interacting with $\mathcal{S}$, $\mathcal{C}$ obtains a blinded search token for a conjunctive query q that has two components - the stokens corresponding to the s-term w_1 (bstags in this version), and the xtokens corresponding to the x-terms $\mathsf{w}_2, \ldots, \mathsf{w}_n$. $\mathcal{C}$ sends these tokens to $\mathcal{S}$ to look up **EDB**. In this process, $\mathcal{S}$ retrieves the encrypted ids and associated deblinding tokens and computes the deblinded xtags for each cross-term and retrieved encrypted id (concatenated with op) pair. $\mathcal{S}$ checks whether the xtags for *all* cross terms and a particular id is set to 1 in XSet. If all xtag locations are set, it returns the encrypted id (concatenated with op) to $\mathcal{C}$. $\mathcal{C}$ locally checks if the id has been added for *all* ws in q, and not deleted even from one w in q. If it is present for all ws, it keeps the id in the final result set, otherwise discards it.

Note that, the xtag computation process is deterministic as it refers to a physical location of a value in the memory (or storage). For validating a $(\mathsf{w}, \mathsf{id})$ pair, the same xtag address needs to be generated each time SEARCH protocol encounters the same $(\mathsf{w}, \mathsf{id})$ pair. This association is revealed even across different queries having a same keyword issued by multiple clients and leads to the leakage across multiple clients. The following example expounds on this observation for a clearer understanding.

Client	Time	Operation	Query/Data
$\mathcal{C}_1$	T1	Search	$\mathsf{w}_1 \wedge \mathsf{w}_2 \wedge \mathsf{w}_3$
$\mathcal{C}_2$	T2	Update	$(\mathsf{w}_3, \mathsf{id})$
$\mathcal{C}_1$	T3	Search	$\mathsf{w}_1 \wedge \mathsf{w}_3$
$\mathcal{C}_3$	T4	Search	$\mathsf{w}_2 \wedge \mathsf{w}_3$

Table 1: SSE execution sequence to illustrate cross-term leakage

Consider the sequence of MC-ODXT events shown in Table 1. Assume that w_3 was not present in **EDB** during T1. It is inserted into **EDB** at T2 by $\mathcal{C}_2$, and queried again (as an x-term) at T3 by $\mathcal{C}_1$ followed by $\mathcal{C}_3$. Observe that these three instances of queries and update involve w_3. By construction of MC-ODXT, the second and third instances generate the same xtag for $(\mathsf{w}_3, \mathsf{id})$ pair. Since $\mathcal{S}$ is assumed to be semi-honest, it can "see" that the same xtag(s) are inserted into the XSet and accessed again later. $\mathcal{S}$'s ability to observe these distinct accesses for xtags is the base of the attack that we outline below.

This two-phase attack assumes a malicious $\mathcal{C}_i$ that colludes with $\mathcal{S}$ in the attack process. In the building phase, $\mathcal{C}_i$ legitimately obtains search tokens for its own conjunctive queries and sends those to $\mathcal{S}$ for searching. $\mathcal{S}$ honestly executes the search routine but at the same time records the xtag access from XSet (as an honest-but-curious entity, it executes the search according to the protocol). Since $\mathcal{C}_i$ colludes with the $\mathcal{S}$, $\mathcal{C}_i$ provides the server with the exact query ws it sent the search tokens for (without shuffling). $\mathcal{S}$ associates the recorded xtags with received query ws and stored locally for later references. $\mathcal{C}_i$ can repeat this process multiple times to obtain multiple $(\mathsf{w}, \mathsf{xtag})$ mappings, and $\mathcal{S}$ can grow the recorded information covering more ws.

While launching the attack on a benign $\mathcal{C}_j$, $\mathcal{S}$ compares the xtags generated for the search tokens of $\mathcal{C}_j$. If the xtags match, $\mathcal{S}$ can infer the corresponding w from the recorded database with high probability. Observe that if $\mathcal{C}_i$ and $\mathcal{S}$ together can cover complete the δ, $\mathcal{S}$ would be able to recover all query keywords of $\mathcal{C}_j$ with complete certainty. We formalise this attack below.

3.1 Formalising MC-ODXT Attack Process

We denote the malicious client colluding with $\mathcal{S}$ using $\mathcal{C}_i$, and a benign client using $\mathcal{C}_j$. We assume that $\mathcal{C}_i$ can send its queries (individual ws) with corresponding tokens tk (without shuffling) to $\mathcal{S}$, and $\mathcal{S}$ builds a local database XDB that stores records of the form $(\mathsf{w}, \mathsf{xtag})$. We assume that $\mathcal{C}_i$ makes t queries during XDB building phase. We summarise the attack process (titled CrossAttack) formally in Algorithm 1 below.

The CrossAttack attack in Algorithm 1 has two phases - the building phase, where $\mathcal{C}_i$ colludes with $\mathcal{S}$ to build the XDB. In the attack phase, $\mathcal{S}$ obtains the xtags corresponding to $\mathcal{C}_j$'s query q, and looks-up XDB to recover the ws in q. Provided the building phase covers a large fraction of ws from Δ, the attack phase would recover the cross-terms with higher probability. Therefore, increasing the number of query iterations t in the building phase would lead to higher successful keyword recovery as more keywords would be covered by XDB. The attack perfectly recovers all cross-terms with probability 1 for the ideal case when XDB contains all ws of Δ.

4 Challenges in Designing Multi-client SSE

Developing a multi-client SSE construction (MRSW or MRMW) poses several challenges as a multi-client-specific workflow fundamentally differs from an SRSW construction. As illustrated by the attack above, trivial extensions like MC-ODXT suffer from multi-client specific attack(s), which need to be suitably addressed without compromising functionality or efficiency. At a high level, the following multi-client-specific privacy notions are necessary for a multi-client construction.

Algorithm 1 Query recovery attack MC-ODXT in presence of colluding malicious client and adversarial server

Input: Query tokens and query ws from malicious $\mathcal{C}_i$, query tokens of benign $\mathcal{C}_j$
Output: $\mathcal{W}$: the set of cross-term ws present in $\mathcal{C}_j$'s query

function CROSSATTACK

- **Building Phase**
- Server
- Initialise empty database XDB
- Server + Malicious Client
- **for** $i = 1$ to t **do**
  - Malicious Client
  - Generate a random query $q_i \xleftarrow{\$} \Delta^*$
  - Obtain search tokens tk_{q_i} for q_i from $\mathcal{D}$
  - Send q_i and tk_{q_i} (without shuffling) to $\mathcal{S}$
  - Server
  - Recover xtags using tk_{q_i} for $\mathsf{w} \in q_i$ available from q_i sent by $\mathcal{C}_i$
  - Insert $(\mathsf{xtag}_j, \mathsf{w}_j)$ into XDB: $\mathsf{XDB}[\mathsf{xtag}_j] = \mathsf{w}_j, \forall \mathsf{w}_j \in q_i$ received from $\mathcal{C}_i$
- **Attack Phase**
- Benign Client
- Obtain search token tk_q for $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$ from $\mathcal{D}$
- Send tk_q to $\mathcal{S}$
- Server
- Compute the xtags from tk_q
- Look-up XDB using the computed xtags: $\mathsf{w}_i = \mathsf{XDB}[\mathsf{xtag}_i]$
- Repeat this for all xtags to recover $\mathcal{W} = \{\mathsf{w}_2, \ldots, \mathsf{w}_n\}$
- Return $\mathcal{W}$

Query privacy. A legitimate client needs to share query information with the data owner, but the data owner must not be able to figure out which keywords are being queried.

Preventing token forgery. A malicious client must not be able to modify or reuse received query tokens with previously obtained (or future) query tokens.

Token validation. The server should be able to validate that it received a genuine query token from a client generated by the data owner and not by a third party.

The above privacy requirements are provisioned in MC-ODXT setting through a two-party oblivious computation-based mechanism, more precisely through OPRF. In that case, $\mathcal{C}$ does not have to share the exact q with $\mathcal{D}$. Instead, $\mathcal{C}$ shares hashed values mapped to group elements with $\mathcal{D}$. However, in dynamic construction following the ODXT structure, it needs to generate several stags and $\mathsf{xtokens}$, which requires modifications of the approach of [22].

Our final construction NOMOS adopts a similar approach to provision multi-client search capability, and an AE-based authentication method is incorporated into the token generation process to validate query tokens on the server side.

However, the leakage from cross-terms in the multi-client settings poses further challenges that need to be addressed without compromising efficiency and security.

Observe that the leakage mentioned in Section 3 appears due to a $(\mathsf{w}, \mathsf{id})$ pair validation requiring a valid physical location look-up in the XSet storage, which is deterministic across multiple queries from different clients. Therefore, $\mathcal{S}$ can record this information and exploits later as demonstrated in CrossAttack of Algorithm 1. Therefore, to prevent this leakage, the XSet look-up access pattern needs to be hidden from $\mathcal{S}$. However, a typical search requires a sublinear number (in total database size) of XSet look-ups and can vary from hundreds to millions for an extensive database. Oblivious RAM (ORAM) [?] constructions are functionally ideal for such private look-up tasks. However, current theoretical constructions of ORAM are impractical for real use at such a scale. On the other hand, Private Information Retrieval (PIR) [?] solutions allow retrieving data privately without revealing accessed locations, but high computation and communication overheads render these practically ineffective. In summary, a multi-client solution faces the following challenges over the traditional XSet look-up method.

Decorrelated access pattern. The XSet locations accessed should vary each time the same $(\mathsf{w}, \mathsf{id})$ pair is encountered such that the adversary is unable to associate a keyword with the access pattern.

Fast look-up. XSet look-up needs to be fast for each $(\mathsf{w}, \mathsf{id})$ pair to reduce search time.

Linear storage. The storage overhead for XSet should remain linear (in total database size) without growing excessively large.

We opt for a redundancy based mechanism for XSet look-up that produces different access patterns for the same $(\mathsf{w}, \mathsf{id})$ pair. We incorporate and elaborate this method in our construction presented in Section 5.

5 Nomos - Dynamic Multi-client SSE Construction

We start by outlining the setting of our main construction with brief details of each entity and how each interacts with other entities. We discuss Nomos construction in two phases - the first phase describes the multi-client provisioning, and the second phase discusses the cross-term leakage mitigation technique incorporated into Nomos. The core structure of the basic construction follows from ODXT construction, and we encourage the readers to refer [30] for more details.

Clients. We assume there are t clients $\{\mathcal{C}_1, \ldots, \mathcal{C}_t\}$ who are allowed to obtain search tokens and search over the database. Each $\mathcal{C}_i$ can request a search token following the GenToken routine and engage in the Search protocol with the server to retrieve query results.

Gate-keeper. We denote the data owner who can update the database and issue search tokens to a client using gate-keeper and the symbol $\mathcal{G}$ (the gate-keeper name is assigned to signify additional responsibility to generate search tokens for clients). Gate-keeper holds the secret key (sk) to generate the search tokens and update the database. Since Nomos is an MRSW construction, gate-keeper is the only entity allowed to update and considered a trusted party.

Server. The server (denoted by $\mathcal{S}$) stores the encrypted database **EDB** comprised of TSet and XSet, and performs update or search as requested. $\mathcal{S}$ engages in the Update protocol with $\mathcal{D}$ to update **EDB**. During search, $\mathcal{S}$ interacts with $\mathcal{C}$ to receive the search tokens and performs the database look-up. At the end of Search protocol, it returns the retrieved encrypted ids to $\mathcal{C}$ matching the actual query.

Nomos setting assumes that $\mathcal{S}$ is an honest-but-curious adversarial entity and a $\mathcal{C}$ in a sense that it can collude with the adversarial server to recover other $\mathcal{C}$'s information. $\mathcal{G}$ is a trusted entity that can update **EDB** stored on $\mathcal{S}$ and also holds the secret keys for generating search tokens. The multi-client provisioning follow the approach of [22] with modifications to support dynamic updates and is similar to MC-ODXT presented in Appendix D.

5.1 Enabling Multi-client Search

Multi-client provisioning in Nomos segregates the search token generation routine and the search process. In contrast, an SRSW construction typically combines both as the client (who is also the data owner) holds the secret key for generating the search tokens. In this way, in Nomos multiple $\mathcal{C}$s can obtain search tokens from $\mathcal{G}$, and later share the tokens with $\mathcal{S}$ to retrieve the resulting ids. We assume that $\mathcal{G}$ initialises the system by invoking Setup (presented in Algorithm 2) and populates **EDB** by invoking Update (presented in Algorithm 4) routine repeatedly on data elements from **DB**.

Algorithm 2 Nomos Setup

1: **function** Nomos.Setup
  Gate-keeper
2: Sample a uniformly random ley K_S from $\mathbb{Z}_p^*$ for OPRF
3: Sample two sets of uniformly random keys $K_T = \{K_T^1, \ldots, K_T^d\}$ and $K_X = \{K_X^1, \ldots, K_X^d\}$ from $(\mathbb{Z}_p^*)^d$ for OPRF
4: Sample uniformly random key K_Y from $\{0,1\}^\lambda$ for PRF F_p
5: Sample shared uniformly random key K_M from $\{0,1\}^\lambda$ for AE
6: Initialise UpdateCnt, TSet, XSet to empty maps
7: Gate-keeper keeps $\mathsf{sk} = (K_S, K_T, K_X, K_Y)$; UpdateCnt is disclosed to clients when required, and K_M is shared between gate-keeper and the server
8: Set **EDB** $= (\mathsf{TSet}, \mathsf{XSet})$
9: Send **EDB** to server

The multi-client search process starts with the token generation process outlined in Algorithm 3 - a two-party protocol between a $\mathcal{C}$ and $\mathcal{G}$. The discussion below briefly summarises the workflow of the GenToken method of Algorithm 3 that generates the search tokens while maintaining query privacy of legitimate clients and preventing token forgery by a malicious client.

Token Generation Phase. The Nomos Search algorithm follows the ODXT [30] Search process which generates two types search tokens – the stags and the xtraps. The stags are generated from the s-term, and are used to obtain the encrypted ids following TSet look-up. Whereas xtraps are generated from x-terms which are used to check validity of a $(\mathsf{w}, \mathsf{id})$ pair through XSet look-up. The GenToken routine in Nomos is responsible for generating these tokens for multiple clients. Since the $\mathcal{G}$ holds the keys $(K_T,\, K_X)$ as a part of the secret key sk to generate the tokens, $\mathcal{C}_j$ needs to send the query to $\mathcal{G}$ to generate tokens without revealing the actual ws. We resort to an OPRF-based computation, allowing $\mathcal{C}_i$ to send query ws in blinded form. The major difference from [22] is that ODXT generates an stag for each update count for the s-term, where as OSPIR-OXT generates a single stag. This is a direct consequence of dynamic update capability of ODXT and GenToken routine computes the blinded exponentiations for each stag. Similarly, $\mathcal{C}_j$ also computes the blinded xtraps and the set of keyword attributes $\mathsf{av} = \{I_1, \ldots, I_n\}$ where $I_i = I(\mathsf{w}_i)$ for $i \in [n]$, which are sent to $\mathcal{G}$.

Blinded Tokens and Query Validation. Upon receiving the search tokens, $\mathcal{G}$ first verifies whether av is a valid set of attributes which $\mathcal{C}_i$ is allowed to query by checking $\mathsf{av} \in \mathcal{P}$. If not valid, $\mathcal{G}$ aborts the process. Otherwise, $\mathcal{G}$ computes it own part of the OPRF computation (party B's computation as discussed in Section 2) by processing bstrap, bstag and bxtrap. The blinded strap (bstrap') generation is done through OPRF evaluation using K_S, and blinded stag (bstag') and xtrap (bxtrap') generation are done through OPRF evaluation using K_T and K_X combined with $\mathcal{G}$s own blinding factors $\{\rho_1, \ldots, \rho_m\}$ and $\{\gamma_1, \ldots, \gamma_m\}$. $\mathcal{G}$'s blinding factors ρ_is and γ_is are necessary to prevent a malicious client from modifying the query by replacing the search tokens. Since the blinding factors are randomly generated for each request, a polynomially bound malicious party can not replicate the blinding factors. $\mathcal{S}$ can verify the tokens as $\mathcal{G}$ encrypts $\{\rho_1, \ldots, \rho_m\}$ and $\{\gamma_1, \ldots, \gamma_m\}$ using AE that $\mathcal{S}$ can authenticate prior to search (in Search routine).

In the final phase of GenToken routine, $\mathcal{C}_i$ deblinds the doubly-blinded bstag's, δs and bxtrap's using its own blinding factors $(r_1, \ldots, r_n)$ and $(s_1, \ldots, s_m)$ to obtain the $\mathcal{G}$-blinded tokens (strap, bstag and bxtrap), which $\mathcal{C}$ subsequently uses as the search token. Note that, $\mathcal{S}$ receives the AE-encrypted blinding factors from $\mathcal{G}$ as a part of the search token which are used for token validation and deblinding during Search execution. However, $\mathcal{S}$ itself is not involved in the GenToken protocol. The AE decryption key K_M is generated by $\mathcal{G}$ as Setup and shared with $\mathcal{S}$.

Search Phase. The Nomos Search of Algorithm 5 is jointly executed by $\mathcal{C}$ and $\mathcal{S}$ without any involvement of $\mathcal{G}$. However, $\mathcal{C}$ must have obtained the search tokens from $\mathcal{G}$ prior to invoking the Search routine. In this phase, the client sends the blinded search tokens and encrypted blinding factors to $\mathcal{S}$ that it received from $\mathcal{G}$ at the end of GenToken (blinded with $\mathcal{G}$'s blinding factors).

Algorithm 3 NOMOS GENTOKEN

Input: $q = \{\mathsf{w}_1, \cdots, \mathsf{w}_n\}$, $\mathcal{P}$ is the set of allowable attribute sequences, ℓ and k (for query) is number of hash functions used in RBF

Output: strap, $\mathsf{bstag}_1, \cdots, \mathsf{bstag}_m$, $\delta_1, \cdots, \delta_m$, $\overline{\mathsf{bxtrap}}_1, \cdots, \overline{\mathsf{bxtrap}}_n$, env

function NOMOS.GENTOKEN

Client

Set $m = \mathsf{UpdateCnt}[\mathsf{w}_1]$

Sample $r_1, \cdots, r_n \xleftarrow{\$} \mathbb{Z}_p^*$

Sample $s_1, \cdots, s_m \xleftarrow{\$} \mathbb{Z}_p^*$

Set $a_j \leftarrow (H(\mathsf{w}_j))^{r_j}$, for $j = 1, \cdots, n$

Set $b_j \leftarrow (H(\mathsf{w}_1||j||0))^{s_j}$, for $j = 1, \cdots, m$

Set $c_j \leftarrow (H(\mathsf{w}_1||j||1))^{s_j}$, for $j = 1, \cdots, m$

Set $\mathsf{av} = (I(\mathsf{w}_1), \cdots, I(\mathsf{w}_n)) = (I_1, \ldots, I_n)$

Gate-keeper

Abort if $\mathsf{av} \notin \mathcal{P}$

Sample $\rho_1, \cdots, \rho_n \xleftarrow{\$} \mathbb{Z}_p^*$

Sample $\gamma_1, \cdots, \gamma_m \xleftarrow{\$} \mathbb{Z}_p^*$

Set $\mathsf{strap}' \leftarrow (a_1)^{K_S}$

Set $\mathsf{bstag}'_j \leftarrow (b_j)^{K_T[I_1]\cdot\gamma_j}$, for $j = 1, \cdots, m$

Set $\delta'_j \leftarrow (c_j)^{K_T[I_1]}$, for $j = 1, \cdots, m$

Set $\mathsf{bxtrap}'_j \leftarrow (a_j)^{K_X[I_j]\cdot\rho_j}$ for $j = 2, \cdots, n$

Set $\mathsf{env} = \mathbf{AuthEnc}_{K_M}(\rho_1, \cdots, \rho_n, \gamma_1, \cdots, \gamma_m)$

Send (strap', $\mathsf{bstag}'_1, \cdots, \mathsf{bstag}'_m$, $\delta'_1, \cdots, \delta'_m$, $\mathsf{bxtrap}'_2, \cdots, \mathsf{bxtrap}'_n$, env) to Client

Client

Set $\mathsf{strap} \leftarrow (\mathsf{strap}')^{r_1^{-1}}$

Set $\mathsf{bstag}_j \leftarrow (\mathsf{bstag}'_j)^{s_j^{-1}}$, for $j = 1, \cdots, m$

Set $\delta_j \leftarrow (\delta'_j)^{s_j^{-1}}$, for $j = 1, \cdots, m$

Set $\mathsf{bxtrap}_j \leftarrow (\mathsf{bxtrap}'_j)^{r_j^{-1}}$, for $j = 2, \cdots, n$

Sample random indices for RBF $\beta_i \xleftarrow{\$} [\ell], i \in [k]$

for $j = 2$ to n **do**

  $\overline{\mathsf{bxtrap}}_j \leftarrow \{\}$

  $\overline{\mathsf{bxtrap}}_j \leftarrow \overline{\mathsf{bxtrap}}_j \cup \mathsf{bxtrap}_j^{\beta_t}$, for $\beta_i \in \{\beta_1, \ldots, \beta_k\}$

Output (strap, $\mathsf{bstag}_1, \cdots, \mathsf{bstag}_m$, $\delta_1, \cdots, \delta_m$, $\overline{\mathsf{bxtrap}}_1, \cdots, \overline{\mathsf{bxtrap}}_n$, env) as search token

At a high level, the SEARCH protocol proceeds in two stages - first, $\mathcal{C}$ computes the final xtraps from the received bxtraps. Note that the resulting xtokens are still blinded as $\mathcal{C}$ does not have $\mathcal{G}$'s blinding factors. $\mathcal{S}$ receives the bstags and computed xtokens along with $\mathcal{G}$'s AE-encrypted blinding factors. $\mathcal{S}$ validates the AE ciphertext using key K_M and proceeds for decryption if the validation is successful. $\mathcal{S}$ deblinds the received bstags to recover the actual stags, and after that, it follows the usual ODXT search routine to retrieve the matching ids. During xtag computation, $\mathcal{S}$ deblinds xtokens using the decrypted $\mathcal{G}$'s blinding factors, and follows the usual ODXT search process.

Update Process. The UPDATE algorithm is invoked by $\mathcal{G}$ with $(\mathsf{w}, \mathsf{id})$ and op as inputs and the encrypted values and tags generated by $\mathcal{G}$ are transferred to $\mathcal{S}$ who updates the TSet and XSet. The UPDATE process of Algorithm 4 adopts the update routine of ODXT with modifications to support multi-client search. The modifications include the way the TSet and XSet addresses (stags and xtags) are generated, such that in the SEARCH routine the same addresses can be recomputed from search tokens obtained via OPRF evaluations.

5.2 Mitigating Cross-term Leakage

Recall from Section 3 that the cross-term leakage arises from repeated xtag accesses (translated to memory location accesses) by $\mathcal{S}$ for the same $(\mathsf{w}, \mathsf{id})$ combinations from different queries (and updates). Intuitively, to mitigate this leakage, these memory accesses (to the same address for a particular $(\mathsf{w}, \mathsf{id})$ pair) need to be different for each access without affecting the look-up performance severely. We adopt a simple yet effective way to achieve this through redundant location accesses, where we keep multiple "copies" of the XSet bit value at multiple addresses. A different address is looked up in each subsequent access for the same $(\mathsf{w}, \mathsf{id})$ pair during search.

Randomising XSet access. We opt for a Bloom filter (BF) based solution to achieve this redundant look-up. At a high level, a BF uses k different hash functions to generate k distinct addresses for an element lookup. However straightforwardly plugging in BF into MC-ODXT does not hide the repeated access pattern as k addresses for a particular $(\mathsf{w}, \mathsf{id})$ pair are still generated from the same xtag. We modify the BF structure slightly in the following way. Instead of using k hash functions to generate the BF addresses (that stores the XSet), we use ℓ hash functions to generate the BF addresses for an input element, where $\ell > k$. During a search, instead of using all ℓ hash functions to generate the BF addresses, a subset of k hash functions out of the ℓ are chosen randomly to generate the BF addresses. Observe that, with this modification, the $\mathcal{S}$ receives a different set of BF addresses for each repeated access of a particular $(\mathsf{w}, \mathsf{id})$ pair and can not correlate among previously accessed elements. We call this version Redundant Bloom Filter (RBF), and we present elaborate details and analysis of RBF in Appendix C.

Avoiding two rounds. Note that incorporating RBF as a module into NOMOS would incur a two-round solution as the RBF addresses needs to be generated from xtag. The xtags must not be revealed to $\mathcal{S}$, and hence need to be generated on the $\mathcal{C}$'s side. This is undesirable in a multi-client setting due to communication/computation overhead and increased leakage from additional token exchanges. We avoid this by embedding the RBF address generation phase into the UPDATE and GENTOKEN algorithms in the following way.

$$\mathcal{G}(\textsc{Update}) : \mathsf{xtag}_i = H(\mathsf{w})^{K_X[I(\mathsf{w})] \cdot F_p(K_Y, \mathsf{id}||\mathsf{op}) \cdot i}, i \in [\ell]$$
$$\mathcal{G}(\textsc{GenToken}) : \overline{\mathsf{bxtrap}}_j \leftarrow \overline{\mathsf{bxtrap}}_j \cup \mathsf{bxtrap}_j^{\beta_t}$$

Algorithm 4 Nomos Update

Input: $K_T = \{K_T^1, \ldots, K_T^d\}$, $K_X = \{K_X^1, \ldots, K_X^d\}$, accessed as $K_T[I(\mathsf{w})]$ and $K_X[I(\mathsf{w})]$ for attribute $I(\mathsf{w})$ of w, ℓ number of hash functions for RBF, $(\mathsf{w}, \mathsf{id})$ to be updated, update operation op

Output: Updated **EDB**

function Nomos.Update

Gate-keeper

Parse $\mathsf{sk} = (K_T, K_X, K_Y)$ and $\mathsf{UpdateCnt}$

Set $K_Z \leftarrow F((H(\mathsf{w}))^{K_S}, 1)$

If $\mathsf{UpdateCnt}[\mathsf{w}]$ is `NULL` then set $\mathsf{UpdateCnt}[\mathsf{w}] = 0$

Set $\mathsf{UpdateCnt}[\mathsf{w}] = \mathsf{UpdateCnt}[\mathsf{w}] + 1$

Set $\mathsf{addr} = (H(\mathsf{w}||\mathsf{UpdateCnt}[\mathsf{w}]||0))^{K_T[I(\mathsf{w})]}$

Set $\mathsf{val} = (\mathsf{id}||\mathsf{op}) \oplus (H(\mathsf{w}||\mathsf{UpdateCnt}[\mathsf{w}]||1))^{K_T[I(\mathsf{w})]}$

Set $\alpha = F_p(K_Y, \mathsf{id}||\mathsf{op}) \cdot (F_p(K_Z, \mathsf{w}||\mathsf{UpdateCnt}[\mathsf{w}])^{-1})$

Set $\mathsf{xtag}_i = H(\mathsf{w})^{K_X[I(\mathsf{w})] \cdot F_p(K_Y, \mathsf{id}||\mathsf{op}) \cdot i}$, where $i \in [\ell]$

Send $(\mathsf{addr}, \mathsf{val}, \alpha, \{\mathsf{xtag}_i\}_{i \in [\ell]})$ to server

Server

Parse $\mathbf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$

Set $\mathsf{TSet}[\mathsf{addr}] = (\mathsf{val}, \alpha)$

Set $\mathsf{XSet}[\mathsf{xtag}_i] = 1$, for $i \in [\ell]$

The revised final Update and GenToken algorithms are presented in Algorithm 4 and 3, respectively. The Search routine is modified to compute final k addresses for RBF and is presented in Algorithm 5.

Since Update protocol is executed in batches of multiple deletions and additions involving several (w,id) pairs (a realistic assumption stated in Section 2), several XSet addresses are generated for inserting multiple (w,id) into RBF-based XSet. The generated XSet addresses (for all (w,id) pairs) must be shuffled by gate-keeper prior to sending to $\mathcal{S}$ in batches.

5.3 Computation and Storage Overhead

The following storage overhead analysis assumes that a single data element in TSet or XSet requires a constant amount of storage, and the group operations and storage look-ups are the costliest operations in practice.

Computation overhead. The Update routine executes for a $(\mathsf{w}, \mathsf{id})$ pair in each invocation. The Update routine computes the TSet addresses along with w-bound deblinding factor, which requires a total of three hash computations, two group operations and field inversion. However, as we use RBF-based XSet, ℓ xtag computations require ℓ group operations that dominates the Update routine with $O(\ell)$ computation overhead. Since ℓ is a constant (which is significantly small compared to the number of updates) for a specific setting, this $O(\ell)$ can be asymptotically approximated to $O(1)$ per Update invocation for a series of updates.

The GenToken protocol requires $|q| + 2|\mathsf{UpdateCnt}[\mathsf{w}_1]|$ hash computations and group operations to generate the client-side values with blinding. The gate-keeper-side processing involves $|q| + 2|\mathsf{UpdateCnt}[\mathsf{w}_1]|$ group operations and $|q| + |\mathsf{UpdateCnt}[\mathsf{w}_1]|$ field multiplications. The client-side deblinding phase computes $|q| + 2|\mathsf{UpdateCnt}[\mathsf{w}_1]|$ group operations. As a result, GenToken incurs $O(|q| + 2|\mathsf{UpdateCnt}[\mathsf{w}_1]|)$ computation overhead asymptotically that is sublinear in the total database size $|\mathbf{DB}|$. The communication overhead is also $O(|q|+2|\mathsf{UpdateCnt}[\mathsf{w}_1]|)$ as $\mathcal{C}$ and $\mathcal{G}$ exchange $|q| + 2|\mathsf{UpdateCnt}[\mathsf{w}_1]|$ tokens in this process.

The Search protocol computes $k \cdot |q| \cdot |\mathsf{UpdateCnt}[\mathsf{w}_1]|$ group operations to compute the blinded $\mathsf{xtokens}$. $\mathcal{S}$ performs $|\mathsf{UpdateCnt}[\mathsf{w}_1]|$ TSet look-ups that require $|\mathsf{UpdateCnt}[\mathsf{w}_1]|$ group operations for deblinding. Additionally, $\mathcal{S}$ computes a total $k \cdot |q| \cdot |\mathsf{UpdateCnt}[\mathsf{w}_1]|$ XSet addresses for look-up. Therefore, Nomos Search incurs $O(k \cdot |q| \cdot |\mathsf{UpdateCnt}[\mathsf{w}_1]|)$ asymptotic computation overhead with all combined. Since k is a small constant, the Search overhead is sublinear in the total database size $|\mathbf{DB}|$. Furthermore, $\mathcal{C}$ needs to send $O(k \cdot |q| \cdot |\mathsf{UpdateCnt}[\mathsf{w}_1]|)$ tokens to $\mathcal{S}$, and it receives $O(|\mathsf{UpdateCnt}[\mathsf{w}_1]|)$ encrypted values back as the result. Hence, the asymptotic communication overhead of Nomos Search routine is $O(k \cdot |q| \cdot |\mathsf{UpdateCnt}[\mathsf{w}_1]|)$.

Storage overhead. We analyse the Nomos storage overhead for **EDB** with respect to the plain database **DB**. The storage overhead for **EDB** in Nomos is essentially the TSet and XSet overhead. The TSet overhead of Nomos is practically the same as of single client dynamic construction ODXT which is $O(|\mathbf{DB}|)$ (linear in terms of the number of records in the plain database **DB**) as the TSet stores one encrypted value for each entry in **DB**. The RBF-backed XSet requires $\ell \cdot O(|\mathbf{DB}|)$ storage. However, compared to TSet, XSet stores only 1/0 for each index and requires lesser storage than TSet that stores encryptions of $O(|\mathbf{DB}|)$ items. As a result, Nomos has linear $O(|\mathbf{DB}|)$ storage overhead asymptotically in practice.

6 Security Analysis of Nomos

We outline the leakage profile and security analysis (informal) of Nomos briefly in this section. Since Nomos adopts the ODXT structure, we closely follow the analysis of ODXT for Nomos and as our multi-client extension follows the approach of OSPIR-OXT we take a similar approach (and setting) to prove the security of Nomos in the multi-client setting. We reiterate on the Nomos setting summarising the roles and capabilities of all entities below.

In this analysis, we treat $\mathcal{S}$ as a polynomially bounded honest-but-curious adaptive adversarial entity, $\mathcal{G}$ as a trusted entity, and each $\mathcal{C} \in C$ as individually malicious who may collude with $\mathcal{S}$. For the ease of analysis, we first categorise the leakages of Nomos according to the leakage to the clients and to the server. Nomos incurs the following class of leakages for the respective entities as summarised below.

Leakage to clients. A client $\mathcal{C}$ receives the following information apart from the data/information trivially available from the execution of respective protocols GenToken and Search. We denote this ensemble of leakages as $\mathcal{L}_{\mathcal{C}}$.

Algorithm 5 NOMOS SEARCH

Input: q, strap, $\mathsf{bstag}_1, \cdots, \mathsf{bstag}_m$, $\delta_1, \cdots, \delta_m$, $\overline{\mathsf{bxtrap}}_1, \cdots, \overline{\mathsf{bxtrap}}_n$, $\beta_1, \ldots, \beta_n$, env, $\mathsf{UpdateCnt}$
Output: IdList

function NOMOS.SEARCH
  Client
  Set $K_Z \leftarrow F(\mathsf{strap}, 1)$
  $m = \mathsf{UpdateCnt}[\mathsf{w}_1]$
  Initialise $\mathsf{stokenList}$ to an empty list
  Initialise $\mathsf{xtokenList}_1, \cdots, \mathsf{xtokenList}_m$ to empty lists
  for $j = 1$ to m **do**
    $\mathsf{stokenList} = \mathsf{stokenList} \cup \{\mathsf{bstag}_j\}$
    for $i = 2$ to n **do**
      $\mathsf{xtokenSet}_{i,j} \leftarrow \{\}$
      for $t = 1$ to k **do**
        Set $\mathsf{xtoken}_{i,j}^t = \overline{\mathsf{bxtrap}}_i[t]^{F_p(K_Z, w_1||j)}$
        Set $\mathsf{xtokenSet}_{i,j} = \mathsf{xtokenSet}_{i,j} \cup \mathsf{xtoken}_{i,j}^t$
      Randomly permute the tuple-entries of $\mathsf{xtokenSet}_{i,j}$
      Set $\mathsf{xtokenList}_j = \mathsf{xtokenList}_j \cup \mathsf{xtokenSet}_{i,j}$
  Send $(\mathsf{stokenList}, \mathsf{xtokenList}_1, \cdots, \mathsf{xtokenList}_m)$
  Server
  Upon receiving env from client, verify env; if verification fails, return $\perp$; otherwise decrypt env
  Parse $\mathbf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$
  Initialise $\mathsf{sEOpList}$ to an empty list
  for $j = 1$ to $\mathsf{stokenList}.size$ **do**
    Set $\mathsf{cnt}_j = 1$
    Set $\mathsf{stag}_j \leftarrow (\mathsf{stokenList}[j])^{1/\gamma_j}$
    Set $(\mathsf{sval}_j, \alpha_j) = \mathsf{TSet}[\mathsf{stag}_j]$
    Initialise $flag = 1$
    for $i = 2$ to n **do**
      Set $\mathsf{xtokenSet}_{i,j} = \mathsf{xtokenList}_j[i]$
      for $t = 1$ to k **do**
        Compute $\mathsf{xtag}_{i,j} = (\mathsf{xtokenSet}_{i,j}[t])^{\alpha_j/\rho_i}$
        If $\mathsf{XSet}[\mathsf{xtag}_{i,j}] = 0$, then set $flag = 0$
      If $flag = 1$, then set $\mathsf{cnt}_j = \mathsf{cnt}_j + 1$
    Set $\mathsf{sEOpList} = \mathsf{sEOpList} \cup \{(j, \mathsf{sval}_j, \mathsf{cnt}_j)\}$
  Sent $\mathsf{sEOpList}$ to client
  Client
  Initialise IdList to an empty list
  for $\ell = 1$ to $\mathsf{sEOpList}.size$ **do**
    Let $(j, \mathsf{sval}_j, \mathsf{cnt}_j) = \mathsf{sEOpList}[\ell]$
    Recover $(\mathsf{id}_j||\mathsf{op}_j) = \mathsf{sval}_j \oplus \delta_\ell$
    If op_j is ADD and $\mathsf{cnt}_j = n$ then set $\mathsf{IdList} = \mathsf{IdList} \setminus \{\mathsf{id}_j\}$
  Output IdList

Token generation leakage. The GENTOKEN protocol of NOMOS involves both s-terms and x-terms in blinded from $\mathcal{C}$. However, **EDB** is not accessed in this phase and $\mathcal{C}$ receives only the following information.

s-term leakage. $\mathcal{C}$ receives back the doubly blinded strap, bstag and δ values that are obtained from the s-term. However, it does not learn anything about the actual blinding factors or the OPRF key. Hence, the s-term leakage to $\mathcal{C}$ in GenToken is $\mathcal{L}_{\mathcal{C}}^{\text{GenToken},s-term} = \perp$.
x-term Leakage. Similarly $\mathcal{C}$ does not learn anything about the x-term blinding factors or the OPRF evaluation keys. Hence, the x-term leakage to $\mathcal{C}$ from GenToken is $\mathcal{L}_{\mathcal{C}}^{\text{GenToken},s-term} = \perp$.

In GenToken protocol, $\mathcal{C}$ only gets to know whether a particular av is valid or not. However, this does not reveal any information about the actual query keywords or the encrypted database. Therefore, the GenToken leakage to $\mathcal{C}$ is $\mathcal{L}_{\mathcal{C}}^{\text{GenToken}} = \perp$.

In the Search protocol, $\mathcal{C}$ inputs the search tokens and receives the query result. It does not receive anything else beyond the query result in encrypted form which is part of the actual protocol execution. Therefore, in Search, $\mathcal{C}$ learns nothing about the s-term and x-terms and the Search leakage to $\mathcal{C}$ can be expressed as $\mathcal{L}_{\mathcal{C}}^{\text{Search},s-term} = |\mathbf{DB}(q)|$ (the volume of the result).

The complete leakage profile to a malicious $\mathcal{C}$ therefore can be expressed as the combined leakage of the above subcomponents.

$$\mathcal{L}_{\mathcal{C}} = \{|\mathbf{DB}(q)|\}$$

Leakage to server. $\mathcal{S}$ engages with $\mathcal{G}$ and a $\mathcal{C}$ in Update and Search respectively. In these routines, $\mathcal{S}$ learns the following information about the query (tokens) or the encrypted data in addition to trivially received data/information as the output of the interaction during Update and Search with the $\mathcal{G}$ and $\mathcal{C}$, respectively. We classify these leakages as below.

Update Leakage. The Update process practically incurs zero leakage to $\mathcal{S}$ as the server receives $(\mathsf{addr}, \mathsf{val})$ to insert into TSet and a set of XSet addresses to set to 1. The addr and val both are obtained using PRF/OPRF evaluation where none of the values are repeated. Similarly, the XSet addresses are also uniquely generated each time and never repeated. Moreover, ADD and DEL operations are treated identically during an update to **EDB**. Hence, $\mathcal{S}$ learns no information about the w or the id, or the op involved in the Update process and the leakage to $\mathcal{S}$ can be expressed as $\mathcal{L}_{\mathcal{S}}^{\text{Update}} = \perp$.

Search leakage. We define a few notations prior to analysing the Search leakage to $\mathcal{S}$. We state the forward and backward privacy notions informally here. Forward privacy dictates that an update involving a w does not reveal any information of a prior search involving w. Backward privacy states that if an update involving adding w and deleting it after, a subsequent search involving w does not reveal that w was involved in these updates. For this analysis, we assume a list $\mathcal{Q}$ that stores the following information.

1. (t, w): w searched at time t.
2. $(t, \mathsf{op}, (\mathsf{w}, \mathsf{id}))$: $(\mathsf{w}, \mathsf{id})$ pair was update with update type op at time t.

Let TimeDB be a function that takes a w as input and returns the respective ids along with the timestamp t. We express this as below.

$$\begin{aligned}\mathsf{TimeDB}(\mathsf{w}) = \{(t, \mathsf{id}) | &(t, \mathsf{ADD}, (\mathsf{w}, \mathsf{id})) \in \mathcal{Q} \\ &\text{and } \forall t' : (t, \mathsf{DEL}, (\mathsf{w}, \mathsf{id})) \notin \mathcal{Q}\}\end{aligned}$$

For conjunctive query $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$, the TimeDB notion is extended as follows.

$$\begin{aligned}\mathsf{TimeDB}(q) = \{(\{t_i\}_{i \in [n]}, \mathsf{id}) | &(t, \mathsf{ADD}, (\mathsf{w}_i, \mathsf{id})) \in \mathcal{Q} \\ &\text{and } \forall t' : (t, \mathsf{DEL}, (\mathsf{w}_i, \mathsf{id})) \notin \mathcal{Q}\}\end{aligned}$$

This essentially corresponds to the ids along with timestamps which satisfies q and have not been deleted from **EDB**. We denote the following s-term and x-term leakages as below.

s-term leakage. Let $\textsc{Upd}(\mathsf{w})$ for a w be defined as follows.

$$\textsc{Upd}(\mathsf{w}) = \{t | \exists(\mathsf{op}, \mathsf{id}) : (t, \mathsf{op}, (\mathsf{w}, \mathsf{id})) \in \mathcal{Q}\}$$

In summary, $\textsc{Upd}(\mathsf{w})$ captures the s-term (w without loss of generality) leakages of q.

x-term leakage. We modify the $\textsc{Upd}$ for a pair $(\mathsf{w}_1, \mathsf{w}_2)$ in the following way.

$$\begin{aligned}\textsc{Upd}(\mathsf{w}_1, \mathsf{w}_2) = \{(t_1, t_2) | \exists(\mathsf{op}, \mathsf{id}) : &(t_1, \mathsf{op}, (\mathsf{w}_1, \mathsf{id})) \in \mathcal{Q} \\ &\text{and } (t_2, \mathsf{op}, (\mathsf{w}_2, \mathsf{id})) \in \mathcal{Q}\}\end{aligned}$$

The above expression implies that $\textsc{Upd}(\mathsf{w}_1, \mathsf{w}_2)$ returns the timestamps of the update operations on w_1 and w_2 involving the same id. For a conjunctive query q, this essentially encapsulates the x-term leakage as $\{\textsc{Upd}(\mathsf{w}_1, \mathsf{w}_j)\}_{j \in [2,n]}$.

The s-term and x-term leakages are combined to obtain the total search leakage to the server as follows.

$$\mathcal{L}_{\mathcal{S}}^{\textsc{Search}} = \textsc{Upd}(q) = \textsc{Upd}(\mathsf{w}_1) \cup \Big(\bigcup_{j=2}^{n} \textsc{Upd}(\mathsf{w}_1, \mathsf{w}_j) \Big)$$

Combining $\mathcal{L}_{\mathcal{S}}^{\textsc{Update}}$ and $\mathcal{L}_{\mathcal{S}}^{\textsc{Update}}$, the leakage to $\mathcal{S}$ can be expressed as

$$\mathcal{L}_{\mathcal{S}} = \{\mathsf{TimeDB}(q), \textsc{Upd}(q)\}.$$

6.1 Security of Nomos

We follow the ideal/real framework of secure computation that are parameterised by leakage function $\mathcal{L}$ capturing the information leaked to an adversarial entity along with correct output. Our goal is to investigate whether an adaptive adversary can do whatever by running the real protocol on data and queries chosen adaptively by the adversary, a simulator can do the same purely from the leakage function.

We consider two separate security analyses - one considering adversarial clients and another considering adversarial server. We start with adpative security against adversarial clients. We follow the analysis approach from [22] for analysis security against adversarial clients.

Security against adversarial $\mathcal{C}$. We present the following definition to analyse security of Nomos against adversarial clients. Definition 2 compares the real execution to an emulated interaction of $\mathsf{SSE}_{\mathcal{L}_\mathcal{C}}$ that models the functionality of Nomos instantiated from $\mathcal{L}$. $\mathsf{SSE}_{\mathcal{L}_\mathcal{C}}$ takes $(\mathbf{DB}, \mathcal{P})$ as input and process queries q if $I(q) \in \mathcal{P}$. If $I(q) \in \mathcal{P}$, it replies with $\mathbf{DB}(q), \mathcal{L}_\mathcal{C}$; otherwise it returns error symbol $\perp$.

Definition 1 (Security against an adversarial client). *Let* $\Pi = \{\textsc{Setup}, \textsc{Update}, \textsc{GenToken}, \textsc{Search}\}$ *be a* Nomos *SSE scheme. Define the following* $\mathbf{Real}^{\Pi}_{\mathcal{A}}$ *and* $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}$ *experiments (algorithms with running time in* 1^λ*) as below, provided* $\mathcal{L}_\mathcal{C}$*,* $\mathcal{A}$*, and* $\mathrm{SIM} = (\mathrm{SIM}_0, \mathrm{SIM}_1, \mathrm{SIM}_2)$*.*

$\mathbf{Real}^{\Pi}_{\mathcal{A}}$: *$\mathcal{A}$ chooses* $\mathbf{DB}$ *and the experiment executes* Setup *to receive* sk *and the repeatedly runs* Update *to obtain* $\mathbf{EDB}$*. After that, $\mathcal{A}$ adaptively invokes* GenToken *and* Search *with input* sk*, where $\mathcal{A}$ interacts with $\mathcal{G}$ and $\mathcal{S}$, respectively. Let t be the number of* GenToken *instances invoked and* av_i *be the local output of $\mathcal{G}$ in i'th instance. As above, if at any point $\mathcal{A}$ halts and output a bit b, the game outputs* $(b, \mathsf{av}_1, \ldots, \mathsf{av}_t)$*.*

$\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}$: *$\mathcal{A}$ chooses* $\mathbf{DB}$ *and the experiment initialises* $\mathrm{SIM}_0, \mathrm{SIM}_1, \mathrm{SIM}_2$ *by executing* $\mathsf{st} \leftarrow \mathrm{SIM}_0(\lambda)$*. After that, each time $\mathcal{A}$ invokes* GenToken*, it interacts with* $\mathrm{SIM}_1(\mathsf{st})$ *and each time $\mathcal{A}$ invokes* Search*, it interacts with* $\mathrm{SIM}_2(\mathsf{st})$*. Both* SIM_1 *and* SIM_2 *are allowed to update global state* st *while interacting with $\mathcal{A}$. Both can issue queries to an ideal emulation of* Nomos*. Let t be the number of such queries and let* $\mathsf{av}_i = I(q)$*. As above, if at any point $\mathcal{A}$ halts and output a bit b, the game outputs* $(b, \mathsf{av}_1, \ldots, \mathsf{av}_t)$*.*

Π is called $\mathcal{L}_\mathcal{C}$-semantically-secure against malicious $\mathcal{C}$s if for any efficient $\mathcal{A}$, there is an efficient algorithm SIM*, such that statistical difference between* $(b, \mathsf{av}_1, \ldots, \mathsf{av}_t)$ *outputs from* $\mathbf{Real}^{\Pi}_{\mathcal{A}}$ *and* $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}$ *experiments is negligible in λ.*

Based of the above definition, we present the following theorem about Nomos security against adversarial clients.

Theorem 1. *A* Nomos *scheme instantiated with Hashed Diffie-Hellman OPRF is adaptively $\mathcal{L}_\mathcal{C}$-semantically-secure against malicious clients provided that the DH assumption holds in $\mathbb{G}$, F_p and F are secure PRFs, and (* $\mathsf{AuthEnc}$,$\mathsf{AuthDec}$*) is an IND-CPA and strongly UF-CMA-secure AE scheme, and all hash functions are modelled using the Random Oracle Model.*

Proof. The proof is given in the Appendix A

Security Against Adversarial $\mathcal{S}$. The adaptive security analysis against an adversarial $\mathcal{S}$ follows from ODXT which is $\mathcal{L}_{\mathrm{ODXT}}$-semantically-secure. Nomos is $\mathcal{L}_\mathcal{S}$-semantically-secure as $\mathcal{L}_\mathcal{S}$ is identical to $\mathcal{L}_{\mathrm{ODXT}}$.

Definition 2 (Security against an adversarial client). *Let* $\Pi = \{\textsc{Setup}, \textsc{Update}, \textsc{GenToken}, \textsc{Search}\}$ *be a* Nomos *SSE scheme. Define the following* $\mathbf{Real}^{\Pi}_{\mathcal{A}}$ *and* $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}$ *experiments (algorithms with running time in* 1^{λ}*) as below, provided* $\mathcal{L}_{\mathcal{S}}$*,* $\mathcal{A}$*, and* $\mathrm{SIM} = (\mathrm{SIM}_0, \mathrm{SIM}_1, \mathrm{SIM}_2)$*.*

$\mathbf{Real}^{\Pi}_{\mathcal{A}}$: *$\mathcal{A}$ chooses* **DB** *and the experiment executes* Setup *to receive* sk *and the repeatedly runs* Update *to obtain* **EDB***. After that, $\mathcal{A}$ adaptively invokes* Update *and* Search *with input* sk*, where $\mathcal{A}$ interacts with $\mathcal{G}$ and $\mathcal{C}$, respectively. Let t be the number of* Update+Search *instances invoked and τ_i be the local output of $\mathcal{G}$ or $\mathcal{C}$ in i'th instance. As above, if at any point $\mathcal{A}$ halts and output a bit b, the game outputs* $(b, \mathbf{EDB}, \tau_1, \ldots, \tau_t)$*.*

$\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}$: *$\mathcal{A}$ chooses* **DB** *and the experiment initialises* $\mathrm{SIM}_0, \mathrm{SIM}_1, \mathrm{SIM}_2$ *by executing* $\mathsf{st} \leftarrow \mathrm{SIM}_0(\lambda)$*. After that, each time $\mathcal{A}$ invokes* Update*, it interacts with* $\mathrm{SIM}_1(\mathsf{st})$ *and each time $\mathcal{A}$ invokes* Search*, it interacts with* $\mathrm{SIM}_2(\mathsf{st})$*. Both* SIM_1 *and* SIM_2 *are allowed to update global state* st *while interacting with $\mathcal{A}$. Both can issue queries to an ideal emulation of* Nomos*. Let t be the number of* Update+Search *queries and let τ_i be the local output* SIM_1 *or* SIM_2*. As above, if at any point $\mathcal{A}$ halts and output a bit b, the game outputs* $(b, \mathbf{EDB}, \tau_1, \ldots, \tau_t)$*.*

Π is called $\mathcal{L}_{\mathcal{S}}$-semantically-secure against adversarial $\mathcal{S}$ if for any efficient $\mathcal{A}$, there is an efficient algorithm SIM*, such that statistical difference between* $(b, \tau_1, \ldots, \tau_t)$ *outputs from* $\mathbf{Real}^{\Pi}_{\mathcal{A}}$ *and* $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}$ *experiments is negligible in λ.*

Specifically, the PRF instances in ODXT that are replaced by OPRF evaluations and $\mathcal{S}$'s view of Search protocol in Nomos can be generated from Search of ODXT with minor modifications due to the group exponentiations by ρ and γ values that are randomly sampled from $\mathbb{Z}_p^*$. We state the theorem below where $\mathcal{L}_{\mathcal{S}}$ is defined as earlier.

Theorem 2. *A* Nomos *scheme instantiated with DH OPRF is adaptively $\mathcal{L}_{\mathcal{S}}$-semantically-secure against adversarial server provided that the DH assumption holds in $\mathbb{G}$, F_p and F are secure PRFs, and (*$\mathsf{AuthEnc}$,$\mathsf{AuthDec}$*) is an IND-CPA and strongly UF-CMA-secure AE scheme, and all hash functions are modelled using the Random Oracle Model.*

Proof. The proof is given in the Appendix A

7 Implementation Details and Results

In this section, we describe a prototype implementation of Nomos and evaluate its performance over real-world databases. We present experimental results for the storage requirements and search performance of Nomos.

Data set and platform. We used the Enron email data set[45] for our experiments. The Enron email data set contained 517,401 documents (emails) and 20

[4] https://www.cs.cmu.edu/SIMenron
[5] https://www.kaggle.com/wcukierski/enron-email-dataset

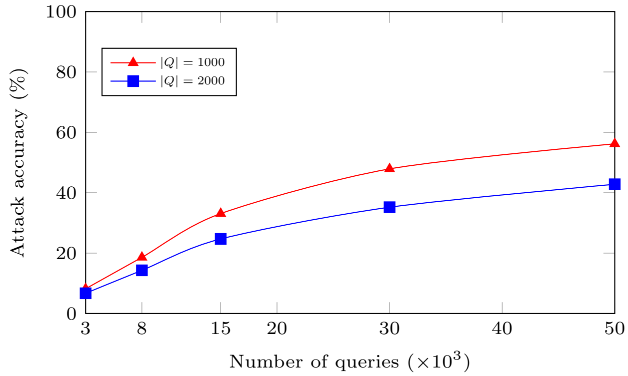

Fig. 1: Attack accuracy vs number of queries by the malicious client in building phase. The set of queries by the benign clients is represented as Q, and $|Q|$ denotes number of queries by benign clients.

million keyword-document pairs, with a total size 1.9 GB. The complete NOMOS implementation was done using C++ (with GCC9 compiler) with native multi-threading support, and we used Redis as the database backend. We ran the experiments on two 24-core Intel Xeon E5-2690 v4 2.6 GHz CPU with 128 GB RAM and 512 GB SSD storage.

Query processing. We evaluated performance for two different cases - two-keyword and multi-keyword queries. The two-keyword queries are of the form $q = \mathsf{w}_1 \wedge \mathsf{w}_2$, which we represent as $q = \mathsf{w}_a \wedge \mathsf{w}_v$ or $q = \mathsf{w}_v \wedge \mathsf{w}_a$. Here w_a is called the constant term whose frequency is kept fixed, and w_v is the variable term whose frequency is varied during experimentation. For the multi-keyword queries of the form $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n,\ n \in [3, 6]$, the first keyword w_1 is varied and maximum frequency of $\{\mathsf{w}_2, \ldots, \mathsf{w}_n\}$ is fixed in one set of experiments and in another set of experiments the frequency of w_1 is kept fixed and max frequency of $\{\mathsf{w}_2, \ldots, \mathsf{w}_n\}$ is varied.

7.1 Leakage Experiments on MC-ODXT

We executed the attack in Algorithm 1 on MC-ODXT to highlight the severity of the leakage discussed in Section 3. The experiment involves building the XDB database from recorded xtags and the associated query ws. Subsequently, we issued queries as a normal client in the attack phase and recorded the successful keyword recoveries from XDB. This result is plotted in Figure 1, where the attack accuracy is plotted against the number of iterations t in the building phase. The accuracy is defined as the ratio of the number of correct x-terms recovered to the total number of x-terms processed across different queries. The attack accuracy improves as the number of iterations increases, allowing to cover more keywords in XDB.

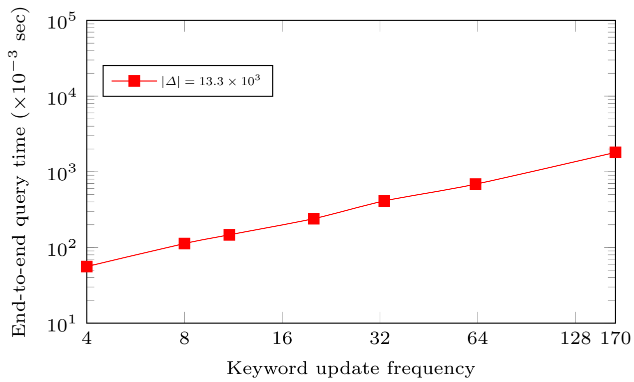

Fig. 2: End-to-end query latency for two keyword queries of the form $q = \mathsf{w}_a \wedge \mathsf{w}_v$ (s-term frequency is kept fixed at 100) (update).

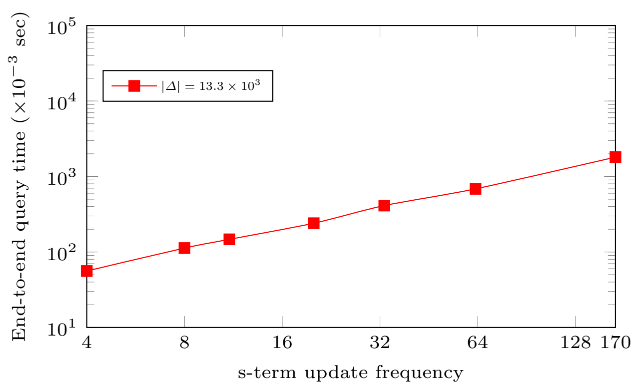

Fig. 3: End-to-end query latency for two keyword queries of the form $q = \mathsf{w}_v \wedge \mathsf{w}_a$ (s-term frequency is varied, x-term is fixed at 250).

7.2 Experiments on Search Latency

We considered two types of queries two evaluate the search performance of Nomos as stated earlier in this section. For two-keyword queries, we fix the frequency of the variable term w_v at 100 and vary the frequency of the constant term w_a from 10 to 5000. The end-to-end search latency for the queries of the form $q = \mathsf{w}_a \wedge \mathsf{w}_v$ in Figure 2 and queries of the form $q = \mathsf{w}_v \wedge \mathsf{w}_a$ in Figure 3. Observe that, in Figure 2, the end-to-end search latency remains almost constant. Whereas in Figure 3, it varies linearly with frequency of the variable term. This observation validates the sublinearity of the Nomos search algorithm where the search latency linearly depends on the frequency of the s-term of $|\mathbf{DB}(\mathsf{w}_1)|$ (where w_1 is the s-term).

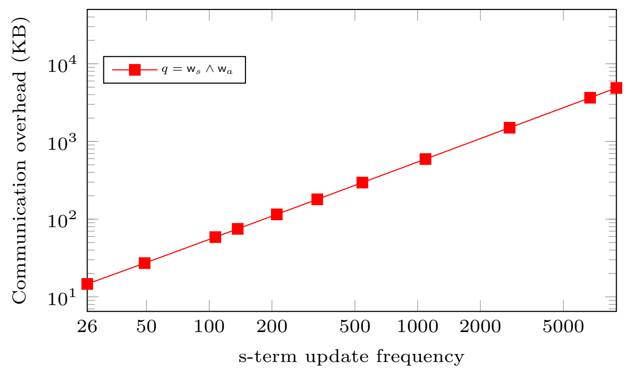

Fig. 4: End-to-end communication overhead for two-keyword queries of the form $q = \mathsf{w}_v \wedge \mathsf{w}_a$.

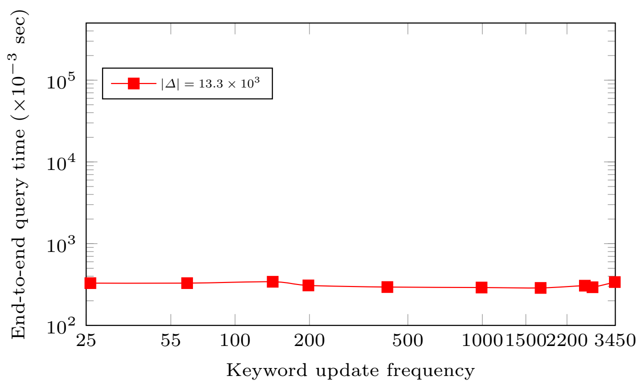

Fig. 5: End-to-end query latency for multi-keyword queries of the form $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$ (s-term frequency is fixed at 60).

The communication overhead of Nomos GenToken and Search are plotted in Figure 4 for two-keyword queries. Nomos incurs subliner communication overhead for GenToken and Search. However, in comparison with ODXT, Nomos has the additional *necessary* overhead of GenToken, whereas the Search communication overhead increases by a factor k due to RBF-based XSet.

We also report experimental results for multi-keyword queries of the form $q = \mathsf{w}_1 \wedge \ldots \mathsf{w}_n$, where $n \in [3, 6]$. The end-to-end search latency for two sets of experiments are plotted for fixed and variable s-term frequencies in Figure 5 and Figure 6, respectively. Observe that, in this case, Nomos performance overhead remains sublinear (proportional to the frequency of the s-term). Similarly, the communication overhead remains sublinear as plotted in Figure 7 in the total database size scaled by the number of cross-terms.

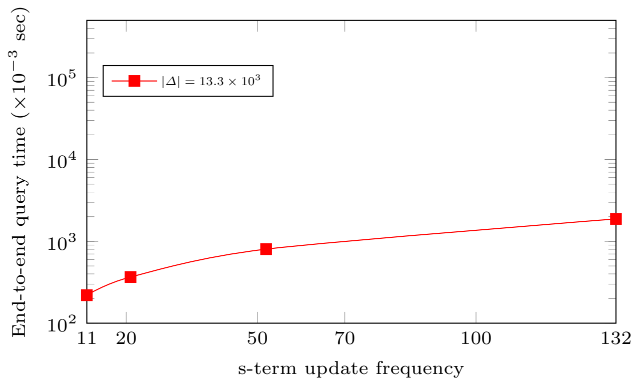

Fig. 6: End-to-end query latency for multi-keyword queries of the form $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$ (s-term frequency is variable).

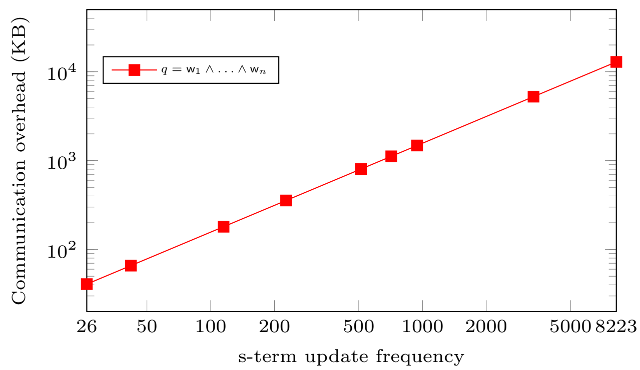

Fig. 7: Communication overhead for multi-keyword queries of the form $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$.

7.3 Evaluation of Storage Overhead

We varied the number of ws in **DB** and generated the corresponding **EDB** by executing Nomos. We compare the Nomos storage overhead with ODXT to illustrate the minor increase in storage overhead as a trade-off with lesser leakage. The **EDB** storage overhead for both Nomos and ODXT are plotted in Figure 8. Observe that the storage overhead profile for both Nomos and ODXT remains linear with increasing plain database size, and Nomos overhead is approximately 2.5 times than ODXT which can be accommodated on the cloud without blow-up in practice.

Client side storage overhead. In Nomos, the client-side storage overhead is essentially the same as of ODXT, where both requires $O(|\Delta|)$ space to store the UpdateCnt information. We plot the client-side storage overhead for both Nomos and ODXT in Figure 9. We include the secret-key storage of $\mathcal{G}$ in Nomos client-

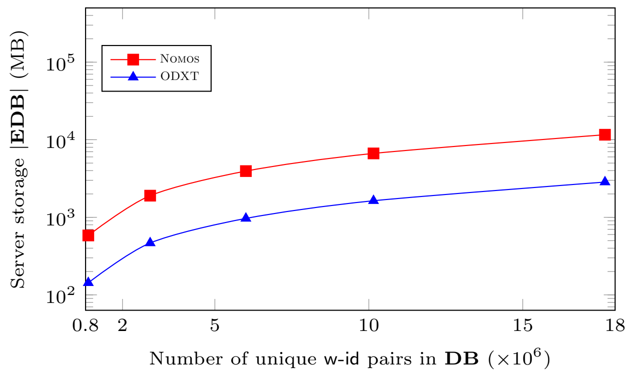

Fig. 8: Server storage overhead (**EDB** size) comparison for Nomos and ODXT.

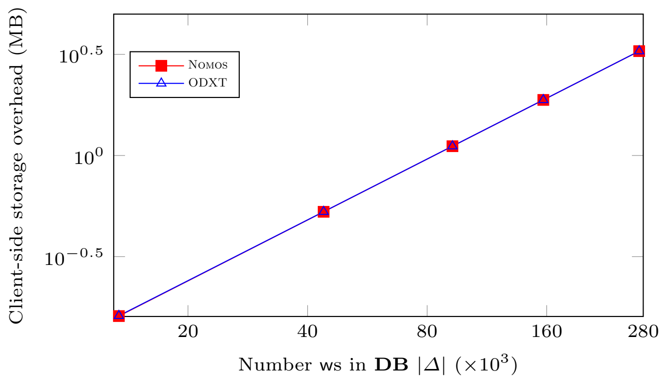

Fig. 9: Client storage overhead comparison for Nomos and ODXT.

side storage overhead to account for the non-server-side storage. In this case too, the overhead varies linearly with number of ws in **DB** (or $|\Delta|$).

8 Conclusion

We introduced the first forward and backward secure dynamic multi-client SSE scheme Nomos supporting conjunctive Boolean queries in this work. Nomos is a MRSW construction that builds upon the state-of-the-art SRSW dynamic construction ODXT [30]. We show that straight-forward extension of ODXT to multi-client setting is vulnerable to a cross-term based leakage that renders it completely insecure against colluding malicious client and adversarial server. Our Nomos construction mitigates this leakage by adopting a customised Bloom filter called redundant Bloom filter while supporting efficient single-round multi-client queries. We present extensive experimental results to demonstrate the performance of Nomos, which is comparable with the state-of-the-art SRSW constructions in the literature. Finally, we leave extending Nomos to the more generic MRMW setting as an interesting open problem for future work.

References

1. Boneh, D., Boyen, X., Goh, E.: Hierarchical identity based encryption with constant size ciphertext. In: EUROCRYPT 2005. pp. 440–456 (2005)
2. Bost, R.: $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In: ACM CCS 2016. pp. 1143–1154 (2016)
3. Bost, R.: $\sigma o\varphi o\varsigma$: Forward secure searchable encryption. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1143–1154 (2016)
4. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: ACM CCS 2017. pp. 1465–1482 (2017)
5. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1465–1482 (2017)
6. Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.C., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation. Cryptology ePrint Archive (2014)
7. Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Dynamic searchable encryption in very-large databases: Data structures and implementation. In: NDSS 2014 (2014)
8. Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: CRYPTO 2013. pp. 353–373 (2013)
9. Chamani, J.G., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: ACM CCS 2018. pp. 1038–1055 (2018)
10. Chang, Y., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: ACNS 2005. pp. 442–455 (2005)
11. Chang, Y.C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: International conference on applied cryptography and network security. pp. 442–455. Springer (2005)
12. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: ASIACRYPT 2010. pp. 577–594 (2010)
13. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: ACM CCS 2006. pp. 79–88 (2006)
14. Etemad, M., Küpçü, A., Papamanthou, C., Evans, D.: Efficient dynamic searchable encryption with forward privacy. PoPETs **2018**(1), 5–20 (2018)
15. Etemad, M., Küpçü, A., Papamanthou, C., Evans, D.: Efficient dynamic searchable encryption with forward privacy. Proc. Priv. Enhancing Technol. **2018**(1), 5–20 (2018)
16. Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., Steiner, M.: Rich queries on encrypted data: Beyond exact matches. In: ESORICS 2015. pp. 123–145 (2015)

17. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: ACM STOC'09. pp. 169–178 (2009)
18. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: CRYPTO 2013. pp. 75–92 (2013)
19. Goh, E.: Secure indexes. IACR Cryptology ePrint Archive **2003**, 216 (2003)
20. Goldreich, O.: Secure multi-party computation. Manuscript. Preliminary version **78**(110) (1998)
21. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996)
22. Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M., Steiner, M.: Outsourced symmetric private information retrieval. In: ACM CCS 2013. pp. 875–888 (2013)
23. Kamara, S., Moataz, T.: Boolean searchable symmetric encryption with worst-case sub-linear complexity. In: EUROCRYPT 2017. pp. 94–124 (2017)
24. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: FC 2013. pp. 258–274 (2013)
25. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: International conference on financial cryptography and data security. pp. 258–274. Springer (2013)
26. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: ACM CCS 2012. pp. 965–976 (2012)
27. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 965–976 (2012)
28. Lai, S., Patranabis, S., Sakzad, A., Liu, J.K., Mukhopadhyay, D., Steinfeld, R., Sun, S., Liu, D., Zuo, C.: Result pattern hiding searchable encryption for conjunctive queries. In: ACM CCS 2018. pp. 745–762 (2018)
29. Naor, M., Reingold, O.: Number-theoretic constructions of efficient pseudo-random functions. Journal of the ACM (JACM) **51**(2), 231–262 (2004)
30. Patranabis, S., Mukhopadhyay, D.: Forward and backward private conjunctive searchable symmetric encryption. In: NDSS 2021 (2021)
31. Song, D.X., Wagner, D.A., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE S&P 2000. pp. 44–55 (2000)
32. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: USENIX Security Symposium 2016. pp. 707–720 (2016)

A Security of Nomos

The proofs for Theorem 1 and 2 are presented in this section. We follow the formal leakage profile outlined in Section 6. We first state the cryptographic assumptions that are necessary for the proofs below.

One-More Gap Diffie-Hellman Assumption. Denote a prime order cyclic group $\mathbb{G}$ with order p (polynomially large in the security parameter λ) and its generator by g. The One-More Gap Diffie-Hellman (OM-GDH) assumption holds in $\mathbb{G}$ if the advantage $\mathsf{Adv}^{DDH}_{\mathbb{G},\mathcal{A}}(\lambda)$ is negligible for all adversaries $\mathcal{A}$. $\mathsf{Adv}^{DDH}_{\mathbb{G},\mathcal{A}}(\lambda)$ is the probability of $\mathcal{A}$ winning the following game.

The game samples a $r \leftarrow \mathbb{Z}_p^*$ at random, and samples two other random elements of $\mathbb{G}$ - (h_1, h_2). The values (h_1, h_2) are shared with $\mathcal{A}$ and it makes a query to the DDH oracle that returns $b \leftarrow a^r$ upon receiving a. $\mathcal{A}$ is allowed make any number of queries to a Decisional Diffie-Hellman (DDH) oracle $\mathsf{DDH}_t(\cdot,\cdot)$, which takes input as (h, v) and returns 1 if $v = h^r$ or 0 otherwise. At the end of game, $\mathcal{A}$ outputs (v_1, v_2) and it wins the game if $v_1 = (h_1)^r$ and $v_2 = (h_2)^r$.

Extended Decisional Diffie-Hellman Assumption. For a prime order cyclic group $\mathbb{G}$ and its generator g, and two arbirary integers $m, n \in \mathbb{N}$, define the following matrix

$$M := \begin{bmatrix} g^{\alpha_1 \cdot \beta_1} & g^{\alpha_1 \cdot \beta_2} & \dots & g^{\alpha_1 \cdot \beta_n} \\ g^{\alpha_2 \cdot \beta_1} & g^{\alpha_2 \cdot \beta_2} & \dots & g^{\alpha_2 \cdot \beta_n} \\ \vdots & \vdots & \ddots & \vdots \\ g^{\alpha_m \cdot \beta_1} & g^{\alpha_m \cdot \beta_2} & \dots & g^{\alpha_m \cdot \beta_n} \end{bmatrix}$$

where $\alpha_i \leftarrow \mathbb{Z}_p^*,\ i \in [m]$ and $\beta_j \leftarrow \mathbb{Z}_p^*,\ j \in [n]$. The extended DDH assumption states that

$$|Pr[\mathcal{A}(g, M) = 1] - Pr[\mathcal{A}(g, M') = 1] \leq \mathsf{negl}(\lambda)$$

where M' is distributed as follows,

$$M := \begin{bmatrix} g^{\gamma_{1,1}} & g^{\gamma_{1,2}} & \dots & g^{\gamma_{1,n}} \\ g^{\gamma_{2,1}} & g^{\gamma_{2,2}} & \dots & g^{\gamma_{2,n}} \\ \vdots & \vdots & \ddots & \vdots \\ g^{\gamma_{m,1}} & g^{\gamma_{m,2}} & \dots & g^{\gamma_{m,n}} \end{bmatrix}$$

where $\gamma_{i,j} \leftarrow \mathbb{Z}_p^*,\ i \in [m],\ j \in [n]$.

Search leakages in SSE schemes. The following leakages are typically incurred by an SSE scheme to the server during SEARCH execution. Assume that Q is a sequence of conjunctive queries issued over time.

Keyword frequency. The total number of time the keywords appear in documents: $N = \sum_{i=1}^{d} |\Delta(\mathsf{w}_i)|$.

Equality pattern. Equality pattern corresponds to the queries that have equal s-terms. Typically, equality pattern $\bar{s} \in [|\Delta|]^Q$ is expressed as a sequence of s-term, where each s-term is assigned an integer with repetitions for each s-term.

Size pattern. Size pattern refers to the number of documents retrieved from encrypted database for the s-term of each query.

Result pattern. Result pattern is the set of ids matching the conjunction of the keywords in the query.

Conditional intersection pattern. Conditional intersection pattern refers to the ids matching a common cross-term for queries with different s-terms.

A.1 Security against Malicious Clients

We provide the proof of Theorem 1 here. We build the simulators SIM $= (\mathrm{SIM}_0, \mathrm{SIM}_1, \mathrm{SIM}_2)$ which are given in Algorithm 6, 7 and 8. We prove Theorem 1 via a sequence of games, where $\mathbf{Real}_{\mathcal{A}}^{\Pi}(\lambda)$ models the interaction of $\mathcal{A}$

with real instance of NOMOS, and $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}(\lambda)$ models the interaction of $\mathcal{A}$ with ideal instance of NOMOS using SIM. We denote the games using G_i, starting from G_0 which is the identical to $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$, and the last one G_{12} is identical to $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}(\lambda)$.

Algorithm 6 Simulator SIM_0

Input: λ

Output: $K_S, K_Y, K_X, K_T, K_P, K_M, QList, TList$

```
function SIM_0(λ)
    Select key K_S for F_G, K_X and K_T for OPRF, K_Y for F_p and K_M for AEAD.
    Initialise empty table QList, which will be indexed by ciphertexts env, and a
    table TList, indexed by keywords w in Δ, which initially holds and empty set
    for each w.
    Return K_S, K_Y, K_X, K_T, K_P, K_M, QList, TList
```

Algorithm 7 Simulator SIM_1

Input: $\mathsf{st}, \mathcal{P}$

Output: Updated st

```
function SIM_1(st)
    On input (a_1, ..., a_n), (I_1, ..., I_n) from A, abort if (I_1, ..., I_n) ∉ P
    Pick ρ_i ←$ Z_p^* for i ∈ [n]
    Pick γ_i ←$ Z_p^* for i ∈ [m]
    Set τ_s ← (a_s)^{K_S}, τ_1 ← (a_1)^{K_T[I_1]·ρ_1}, and τ_i ← (a_i)^{(K_X[I_i]···ρ_i)} for i ∈ [2, n]
    Set ε_i ← (b_i)^{(K_T[I_1]·γ_i)} for i ∈ [m]
    Set μ_i ← (c_i)^{(K_T[I_1])} for i ∈ [m]
    Update QList in st by setting QList(env) ← (I_1, ..., I_n; ρ_1, ..., ρ_n; γ_1, ..., γ_m)
    Set env ← AuthEnc(K_M, (ρ_1, ..., ρ_n, γ_1, ..., γ_m)) and output
    (env, τ_s, τ_1, ..., τ_n, ε_1, ..., ε_m, μ_1, ..., μ_m)
```

Game G_1. In G_1 we modify game G_0 by adding an abort if AE ciphertexts env_1 and env_2 are accepted by $\mathcal{S}$ in SEARCH protocol have not been genuinely generated by $\mathcal{G}$ in GENTOKEN.

Lemma 1. *Game G_1 is indistinguishable from game G_0.*

Proof. By Strong-UF-CMA unforgability of AE scheme, G_1 is indistinguishable from G_0.

Game G_2. In game G_2 we add an abort if ever two GENTOKEN instances generate the same env ciphertext.

Lemma 2. *Game G_2 is indistinguishable from G_1.*

Algorithm 8 Simulator SIM_2

Input: st, upon receiving $(\mathsf{env}, \mathsf{bstag}, \mathsf{xtoken}[1], \ldots, \mathsf{xtoken}[m])$ from $\mathcal{A}$
Output: Updated st

1: **function** $\text{SIM}_2(\mathsf{st})$
2: Retrieve $(I_1, \ldots, I_n; \rho_1, \ldots, \rho_n) \leftarrow \mathsf{QList}(\mathsf{env})$, abort if $\mathsf{QList}(\mathsf{env}) = \perp$
3: Set $\mathsf{stag} \leftarrow (\mathsf{bstag})^{1/\rho_1}$. If there exists $\mathsf{w}_1 \in \Delta$, s.t. $\mathsf{stag} \leftarrow (H(\mathsf{w}_1))^{K_T[I_1]}$, and $I(\mathsf{w}_1) = I_1$ then set $\mathsf{strap} \leftarrow (H(\mathsf{w}_1))^{K_S}$ and $K_Z \leftarrow F(\mathsf{strap}, 1)$. Abort if no such w_1 found.
4: Set $c \leftarrow 0$ and $\mathsf{found} \leftarrow \mathsf{False}$ and perform the following loop while $\mathsf{found} \leftarrow \mathsf{False}$
- Set $c \leftarrow c + 1$ and $z_c \leftarrow F_p(K_Z, c)$. For $\mathsf{xtoken}[c] = (\mathsf{xtoken}[c, 2], \ldots, \mathsf{xtoken}[c, n])$, if $\exists(\mathsf{w}_2, \ldots, \mathsf{w}_n) \in \Delta^{n-1}$, s.t. $\mathsf{xtoken}[c, i] = (H(\mathsf{w}_i))^{K_X[I(\mathsf{w}_i)] \cdot z_c \cdot \rho_i}$ for $i = 2, \ldots, n$ then set $\mathsf{found} \leftarrow \mathsf{True}$.
- Abort if $\mathsf{found} = \mathsf{False}$ and $\mathsf{xtoken}[c]$ is the last element in $\mathcal{A}$'s message.

5: Send $q = (\mathsf{w}_1, \ldots, \mathsf{w}_n)$ to $\mathsf{SSE}_{\mathcal{L}}$ where $\mathsf{w}_1, \ldots, \mathsf{w}_n$ are the keywords found above. Since av is guaranteed to be included in $\mathcal{P}$, SIM_2 receives back $\mathbf{DB}(q), \mathsf{TSetL}$.
6: Set $S' \leftarrow \phi$ and $D \leftarrow \mathbf{DB}(q)$. $\forall \mathsf{id} \in D$ s.t. $(c, \mathsf{id}, e) \in \mathsf{TList}(\mathsf{w}_1)$, add c to S' and delete id from D.
7: Pick S as random $|D|$-element subset in $\{1, \ldots, \mathsf{TSetL}\} \setminus S'$, and while S is non-empty do:
- Remove a random element c from S and a random element id from D.
- Set $\mathsf{rin} \leftarrow P_\tau(K_p, \mathsf{id})$, $\mathsf{e} \leftarrow \mathsf{Enc}(K_e, \mathsf{rid})$.
- Update TList in st by adding $(c, \mathsf{id}, \mathsf{e})$ to $\mathsf{TList}(\mathsf{w}_1)$.

8: Starting from the last counter c encountered above, perform the following loop while $c \leq \mathsf{TSetL}$:
- Set $z_c \leftarrow F_p(K_Z, c)$. If $\mathsf{xtoken}[c, i] = (H(\mathsf{w}_i))^{K_X[I(\mathsf{w}_i)] \cdot x_c \cdot \rho_i}$ for $i = 2, \ldots, n$ and if there exists $(c, \mathsf{id}, \mathsf{e})$ in $\mathsf{TList}(\mathsf{w}_1)$ s.t. $\mathsf{id} \in \mathbf{DB}(q)$, then send e to $\mathcal{A}$ and set $c \leftarrow c + 1$.

9: Send stop to $\mathcal{A}$ and halt.

Proof. Since ρ_i's are generated at random from $\mathbb{Z}_p^*$, and collision in ciphertext implies collision in the plaintext, hence; it will be a contradiction to have such an occurrence. Therefore, G_2 is indistinguishable from G_1.

Game G_3. Abort if for any two distinct w_1 and w_2 in Δ, two OPRF instances from different keys K_T or K_X collide – two distinct OPRF evaluations involving w_1 and w_2 using K_T or K_X output the same value.

Lemma 3. *G_3 is indistinguishable from G_2.*

Proof. Since OPRFs using K_T and K_X are essentially secure PRF instances, probability of such collisions is negligible and hence; G_3 is indistinguishable from G_2.

Game G_4. In this game, we change the way of interaction with the Search protocol. In G_0, the game computes $\mathsf{stag} \leftarrow \mathsf{bstag}^{1/\rho_1}$. However, after that, it searches through Δ to find a w such that corresponding stag generation results in stag. If such stag is found, it obtains corresponding value from from TSet as a tuple (sval, α); otherwise aborts.

Lemma 4. *G_4 is indistinguishable from G_3.*

Proof. Since collisions in stag have been eliminated in previous games and each stag is uniquely generate from w, the retrieval in G_4 is essentially the same as from the stag received as in G_3, except a negligible probability of error. On the other hand, if two stags do not match, the TSet has negligible probability of returning a non-empty result. Therefore, in this case G_4 is indistinguishable from G_3.

Game G_5. In this game, instead of encrypting $(\rho_1, \ldots, \rho_n, \gamma_1, \ldots, \gamma_m)$, the game encrypts a set of random values $(\rho'_1, \ldots, \rho'_n, \gamma'_1, \ldots, \gamma'_m)$. The game keeps QList indexed by env. When responding to a GenToken request, it also stores the attributes $(I(\mathsf{w}_1), \ldots, I(\mathsf{w}_n))$ and actual blinding factors $(\rho_1, \ldots, \rho_n, \gamma_1, \ldots, \gamma_m)$. During Search, the game retrieves $(\rho_1, \ldots, \rho_n, \gamma_1, \ldots, \gamma_m)$ from QList.

Lemma 5. *G_5 is indistinguishable from G_4.*

Proof. In game G_2, only envs that are uniquely generated in GenToken invocations are accepted. Hence, by IND-CCA guarantee of AE, G_5 is indistinguishable from G_4.

Game G_6. We consider the case where the game identifies w_1 from bstag and ρ_1 recovered from $\mathsf{QListenv}$, such that $(\mathsf{bstag})^{1/\rho_1} = (H(\mathsf{w}_1)^{K_T[I_1]})$ where $I_1 = I(\mathsf{w}_1)$. This is identical to G_5, except G_6 ignores $\mathsf{stag} = \mathsf{bstag}^{1/\rho_1} = (H(\mathsf{w}_1))^{K_T[I(\mathsf{w}_1)]}$ but $I_1 \neq I(\mathsf{w}_1)$.

Lemma 6. *The probability of finding such w_1 is negligible and thus G_5 and G_6 are indistinguishable.*

Proof. Denote $K_T[I_1] \cdot \rho_1$ as e_j used for computing b_1 from a_1 during j'th GenToken invocation. By construction, e_js are randomly uniformly distributed in $\mathcal{Z}_p^*$. The process of validating whether w_1 is associated with bstag in Search can be alternatively written as validating whether $\mathsf{bstag} = (H(\mathsf{w}_1))^{(K_T[I(\mathsf{w}_1)]/K_T[I_1]) \cdot e_j}$ where e_j is used in GenToken that generated env used in the subsequent Search instance. In that case, the discrete logarithm of $(H(\mathsf{w}_1))^{e_j}$ and bstag must be equal to $K_T[I(\mathsf{w}_1)]/K_T(I-1)$. As K_T is not used in any other way in G_6 and K_T sub-keys are all randomly uniformly sampled from $\mathbb{Z}_p^*$ and the number of validations is polynomially bounded, the success probability of $I(\mathsf{w}_1) = I_1$ is negligible. Thus G_6 and G_5 are indistinguishable from each other.

Game G_7. This game adds an abort if two instances of Search invocation by $\mathcal{A}$ involve the same env, but have different bstags - bstag and bstag', such that $\mathsf{bstag}^{1\rho_1} = (H(\mathsf{w}_1))^{K_T[I(\mathsf{w}_1)]}$ and $\mathsf{bstag}'^{1\rho_1} = (H(\mathsf{w}'_1))^{K_T[I(\mathsf{w}'_1)]}$. Denote by e_j the operation $K_T[I(\mathsf{w}_1)]$ done in j'th instance of GenToken.

Lemma 7. *The probability of encountering two pairs $(\mathsf{w}_1, \mathsf{bstag})$ and $(\mathsf{w}'_1, \mathsf{bstag}')$ resulting in the same env is negligible and therefore G_7 is indistinguishable from G_6.*

Proof. The game emulates G_3 by picking r_1, and r_2 in $\mathbb{Z}_p^*$ and sets $H(\mathsf{w}) \leftarrow (h_1)^{r_1}(h_2)^{r_2}$ where h_1, h_2 DH challenge inputs. Also, it picks a random index j in $[1, \eta]$ where η is the number of maximum invocations of GenToken by $\mathcal{A}$ allowed by $\mathcal{A}$. The experiment sends a_1 to the DH challenger that replies with $b_1 \leftarrow (a_1)^t$ where t is chosen by DH challenger, and b_1 is passed to $\mathcal{A}$. Subsequently, for each Search invocation where $\mathcal{A}$ sends env, the experiment takes bstag sent by $\mathcal{A}$ and env. For each query q from $\mathcal{A}$ to OPRF, the DDH oracle is consulted with $(a_1, b_1, H(\mathsf{w}), \mathsf{bstag})$ to figure out if this is a DDH tuple. If for two instances of Search the checks are verified for $(\mathsf{w}, \mathsf{bstag})$ and $(\mathsf{w}', \mathsf{bstag}') \neq (\mathsf{w}, \mathsf{bstag})$, the following computation is performed - $(\mathsf{bstag}^{r_2}(\mathsf{bstag}')^{-r_2})^{1/(r_1 r_2' - r_1' r_2)}$ and $(\mathsf{bstag}^{r_1}(\mathsf{bstag}')^{-r_1})^{1/(r_1' r_2 - r_1 r_2')}$, where $H(\mathsf{w}) = (h_1)^{r_1}(h_2)^{r_2}$ and $H(\mathsf{w}') = (h_1)^{r_1'}(h_2)^{r_2'}$. The success probability of this event is $(1/m) \cdot p_6$, where m is the upper bound on the number of GenToken instances excluding the probability of $r_1 r_2' = r_1' r_2$.

Game G_8. In this game, we replace the PRF instances of the form $F_p(K, \cdot)$ with a random function $F_R(\cdot)$ with a range onto $\mathbb{Z}_p^*$.

Lemma 8. *G_8 and G_7 are indistinguishable.*

Proof. By the indistinguishability property of a PRF G_8 and G_7 are indistinguishable.

Game G_9. In this game the Search process of G_8 is modified in the following way. G_9 finds w_1 using bstag and (I_1, ρ_1) from $\mathsf{QList}(\mathsf{env})$ as described earlier and computes $\mathsf{strap} \leftarrow OPRF(K_S, \mathsf{w}_1)$ and $(K_Z, K_e) \leftarrow (F_\tau(\mathsf{strap}, 1), F_\tau(\mathsf{strap}, 2))$. After that, for each c, it computes $z_c \leftarrow F_p(K_Z, c)$. Provided $\mathsf{xtoken}[c] = (\mathsf{xtoken}[c, 2], \ldots, \mathsf{xtoken}[c, n])$, it searches for w_i in Δ such that $\mathsf{xtoken}[c, i] = (OPRF(K_X, \mathsf{w}_i))^{z_c \cdot \rho_i}$ and $\mathsf{id}_c \in \mathbf{DB}(\mathsf{w}_i)$ where id_c is the c'th (e, y) entry in $\mathbf{t} = \mathbf{T}[\mathsf{w}_1]$. Any such w_i is found for all i is sent to $\mathcal{A}$.

Let $\Delta(\mathsf{id})$ denote the set of ws which appear in id. For G_9 to succeed in the above check for any c, i, G_8 should also be successful as $\mathsf{xtoken}[c, i] = (OPRF(K_X, \mathsf{w}_i))^{z_c \cdot \rho_i}$ would imply $(\mathsf{xtoken}[c, i])^{y_c/]rho_i}$ is equal to $(OPRF(K_X, \mathsf{w}_i))^{\mathsf{xid}_c}$. If $\mathsf{id}_c \in \mathbf{DB}(\mathsf{w}_i)$ then this particular xtrap entry is present in XSet. G_9 and G_8 can only differ if $(\mathsf{xtoken}[c, i])^{y_c/\rho_i} \in \mathsf{XSet}$ and $\mathsf{xtoken}[c, i] \neq (OPRF(K_X, \mathsf{w}))^{z_c \cdot \rho_i}$ for any (c, i) and for all $\mathsf{w} \in \Delta(\mathsf{id}_c)$.

Denote the exponent $K_X[I_i] \cdot \rho_i$ with a_j in GenToken as $e_{i,j}$. G_8 can choose $e_{i,j}$ a random value in $\mathbb{Z}_p^*$ and compute a ρ_i as $e_{i,j}/K_X[I_i]$. Therefore, we can express that $(\mathsf{xtoken}[c, i])^{y_c/\rho_i}$ finds a match in XSet as the existence of an $\bar{\mathsf{id}}$ and $\bar{\mathsf{w}} \in \Delta(\bar{\mathsf{id}})$ such that following expression holds.

$$(\mathsf{xtoken}[c, i])^{1/z_c} = (H(\bar{\mathsf{w}}))^{e_{i,j} \cdot \frac{K_X[I(\bar{\mathsf{w}})]}{K_X[I_i]} \cdot \frac{F_R(\bar{\mathsf{id}})}{F_R(\mathsf{id}_c)}} \tag{1}$$

Also, as $\mathsf{xtoken}[c, i] \neq (OPRF(K_X, \mathsf{w}))^{z_c \cdot \rho_i}$ for $\mathsf{w} \in \Delta(\mathsf{id}_c)$, the following expression holds.

$$(\mathsf{xtoken}[c, i])^{1/z_c} \neq (H(\mathsf{w}))^{e_{i,j} \cdot \frac{K_X[I(\mathsf{w})]}{K_X[I_i]}} \tag{2}$$

Lemma 9. *The probability of that Equation* (1) *holds and Equation* (2) *does not hold for every* $\mathsf{w} \in \Delta$ *is negligible and* G_9 *is indistinguishable from* G_8.

Proof. If Equation (1) holds for $\bar{\mathsf{id}} = \mathsf{id}_c$, and $I(\bar{\mathsf{w}}) = I_i$, then $(\mathsf{xtoken}[c,i])^{1/z_c} = (H(\bar{\mathsf{w}}))^{e_{i,j}}$ for $\bar{\mathsf{w}} \in \Delta(\mathsf{id}_c)$. However, this is a contradiction with Equation (2) where $I(\bar{\mathsf{w}}) = I_i$ implies $(\mathsf{xtoken}[c,i])^{1/z_c} \neq (H(\bar{\mathsf{w}}))^{e_{i,j}}$. We consider the following two cases below for $\bar{\mathsf{id}} \neq \mathsf{id}_c$ (case 1) and $I(\bar{\mathsf{w}}) \neq I_i$ (case 2) to show that these two cases occur with only negligible probability.

Case 1. This argument essentially shows that $\bar{\mathsf{id}} \neq \mathsf{id}$ holds for negligible probability. In order to show that, G_8 is slightly modified to G_8' where it does not append (e, y) to $T[\mathsf{w}]$ tuples Instead, it appends (e, id, z) such that does not have to query $F_I(\cdot)$ while creating $\mathbf{T}$. Furthermore, it does not create XSet during Setup. The test procedure of G_8 is modified for each c and i in Search in the following way. G_8' searches for $\bar{\mathsf{w}} \in \Delta$ and $\bar{\mathsf{id}} \in \mathbf{DB}(\mathsf{w})$ instead of checking $(\mathsf{xtoken}[c,i])^{y_c/\rho_i}$ in XSet, using id_c, z_c stored in c'th entry in $\mathsf{T}[\mathsf{w}_1]$. These two are equivalent cases and G_8' essentially is an identical view of G_8. Additionally, test of Equation (1) can be done by checking $a = b^{F_I(\bar{\mathsf{id}})/F_I(\mathsf{id}_c)}$ where $a = (\mathsf{xtoken}[c,i])^{1/z_c}$ and $b = (H(\bar{\mathsf{w}}))^{e_{i,j} \cdot (K_X[I(\bar{\mathsf{w}})]/K_X[I_i])}$. In summary, G_8' can work with $F_I(\cdot)$ as follows. G_8' presents a tuple $(a, b, x, \bar{x})$ to the oracle which returns 1 if $a^{F_I(x)} = b^{F_I(\bar{x})}$, otherwise 0. As H has $\mathbb{G}$ as the range, all $(a, b, x, \bar{x})$ invocation in G_8' has $b \neq 1$ (except for negligible probability). Also, as the random function $F_I(\cdot)$ maps to $\mathbb{Z}_p^*$ which is invoked polynomially bounded number of times by G_8', there is negligible probability of that the oracle returns $x \neq \bar{x}$. This implies that Equation (1) holds with negligible probability.

Case 2. In this case, we show that Equation (2) holds with negligible probability for $I(\bar{\mathsf{w}}) \neq I_i$. Recall that, G_8' uses K_X for testing Equation (1) that is equivalent to presenting a query of the form $(a, b, I, \bar{I})$ where $a = (\mathsf{xtoken}[c,i])^{1/z_c \cdot F_I(\mathsf{id}_c)}$, $b = (H(\bar{\mathsf{w}}))^{e_{i,j} \cdot F_I(\bar{\mathsf{id}})}$, $I = I_i$, and $\bar{I} = I(\bar{\mathsf{w}})$ and gets back 1 if $a^{K_X[I]} = b^{K_X[I]}$. Since H randomly maps to $\mathbb{G}$, $b = 1$ occurs with negligible probability. Furthermore, as K_X is randomly constructed by choosing from $\mathbb{Z}_p$ and G_8' makes polynomially many queries, the probability of such successful queries is negligible.

Game G_{10}. In this game, G_9 is modified to identify w_i from $\mathsf{xtoken}[c,i]$ and (z_c, ρ_i) with verification using $\mathsf{xtoken}[c,i] = (OPRF(K_X, \mathsf{w}_i))^{z_c \cdot \rho_i}$ and $\mathsf{id}_c \in \mathbf{DB}(\mathsf{w}_i)$ and $I(\mathsf{w}_i) = I_i$ retrieved from $\mathsf{QList}(\mathsf{env})$. The differentiating part of the games - $\mathsf{xtoken}[c,i] = (H(\mathsf{w}_i))^{K_X[I(\mathsf{w}_i)] \cdot z_c \cdot \rho_i}$ given $\mathsf{id}_c \in \mathbf{DB}(\mathsf{w}_i)$ and $I(\mathsf{w}_i) \neq I_i$.

Lemma 10. *The above event occurs with negligible probability and therefore,* G_{10} *is indistinguishable from* G_9.

Proof. Recall the notation $e_{i,j} = K_X[I_i] \cdot \rho_i$ used in GenToken. $\mathsf{xtoken}[c,i]$ can be expressed as $\mathsf{xtoken}[c,i] = (H(\mathsf{w}_i))^{z_c \cdot e_{i,j} \cdot \frac{K_X[I(\mathsf{w}_i)]}{K_X[I_i]}}$. In summary, the discrete log of this expression is equal to $K_X[I_i]/K_X[I(\mathsf{w}_i)]$. G_{10} does not use K_X in any other way for this validation. As K_X is generated randomly from $\mathbb{Z}_p^*$ and allowed to make only polynomial number of checks, the probability of successful checks for $I(\mathsf{w}_i) \neq I_i$ is negligible and G_{10} is indistinguishable from G_9.

Game G_{11}. The game aborts if it receives the following - env, an index i, two counter c, c' and two keywords $\mathsf{w}_i \neq \mathsf{w}'_i$, such that $\mathsf{xtoken}[c,i] = (H(\mathsf{w}_i))^{K_X[I_i] \cdot z_c \cdot \rho_i}$ and $\mathsf{xtoken}[c',i] = (H(w'_i))^{K_X[I_i] \cdot z_{c'} \cdot \rho_i}$. The game considers the Search invocations that will collide under these modifications. Following the exponent notation $e_{i,j} = K_X[I_1] \cdot \rho_i$ in the j'th invocation of GenToken, the xtoken expressions is modified to $\mathsf{xtoken}[c,i]^{1/z_c} = (H(\mathsf{w}_i))^{e_{i,j}}$ and $\mathsf{xtoken}[c',i]^{1/z'_c} = (H(\mathsf{w}'_i))^{e_{i,j}}$.

Lemma 11. *The probability of getting two* $(\mathsf{w}_i, \mathsf{xtoken}[c,i], z_c)$ *and* $(\mathsf{w}'_i, \mathsf{xtoken}[c',i], z'_c)$ *for* $\mathsf{w}_i \neq \mathsf{w}'_i$ *is negligible. Thus* G_{11} *is indistinguishable from* G_{10}.

Proof. Let μ be the probability of getting two such tuples. Similar to G_7, an OM-GDH challenge (h_1, h_2) is identical G_{11}. However, on each query w by $\mathcal{A}$ to H, it selects (r_1, r_2) in $\mathbb{Z}_p^*$ and outputs $H(\mathsf{w}) \leftarrow (h_1)^{r_1}(h_2)^{r_2}$. It also samples a random number j between $[1, l]$ where l is the maximum number of time GenToken invocation by $\mathcal{A}$ is allowed. During each invocation, it selects i at random from $[2, n]$ and sends the a_i from the GenToken to OM-GDH challenger, and gets back $b_i \leftarrow (a_i)^t$, where t is chosen by the OM-GDH challenge game. $\mathcal{A}$ gets back this b_i in response. In each Search invocation where $\mathcal{A}$ sends env, for each c the game takes $\mathsf{xtoken}[c,i]$ received from $\mathcal{A}$, and each query w issued by $\mathcal{A}$ to H, the DDH oracle is consulted if $(a_1, b_1, H(q), (\mathsf{xtoken}[c,i])^{1/z_c})$ is a DDH tuple. If for two instances $(\mathsf{w}, \mathsf{xtoken}[c,i], z_c)$ and $(\mathsf{w}', \mathsf{xtoken}[c',i], z_{c'})$ are the same for $\mathsf{w} \neq \mathsf{w}'$, the game computes h_1^t, h_2^t as $((\mathsf{xtoken}[c,i])^{r_2/z_c}(\mathsf{xtoken}[c',i])^{-r_2/z_{c'}})^{1/(r_1 r'_2 - r'_1 r_2)}$ and $((\mathsf{xtoken}[c,i])^{r_1/z_c}(\mathsf{xtoken}[c',i])^{-r_1/z_{c'}})^{1/(r'_1 r_2 - r_1 r'_2)}$ where $H(\mathsf{w}) = (h_1)^{r_1}(h_2)^{r_2}$ and $H(\mathsf{w}') = (h_1)^{r_1}(h_2)^{r_2}$. The probability of such an event is $\mu/(ln)$ where n is the number of ws in the query. This is negligible and G_{11} is indistinguishable from G_{10}.

Game G_{12}. Game G_{12} modifies G_{11} as follows. It does not refer to TSet or XSet (it skips the creation process) and samples the keys K_S, K_T, K_Y, K_M. The GenToken instances are invoked the same way as in G_{11}. In Search, for a counter c, $\mathsf{xtoken}[c,i] = (OPRF(K_X, \mathsf{w}_i))^{z_c \cdot \rho_i}$, $\mathsf{id} \in \mathbf{DB}(\mathsf{w}_i)$, and $I(\mathsf{w}_i) = I_i$ for $i = 2, \ldots, n$, it constructs a query q from $\mathsf{w}_1, \ldots, \mathsf{w}_n$ and send to $\mathsf{SSE}_{\mathcal{L}_c}$. As $\mathsf{av} \in \mathcal{P}$, it returns $(\mathbf{DB}(q), |\mathbf{DB}(\mathsf{w}_1)|)$. The game samples $|\mathbf{DB}(q)|$ random indices from $[1, |\mathbf{DB}(\mathsf{w}_1)|]$ and assigns to each entry of $\mathbf{DB}(q)$. The game maintains TList such that $\mathsf{TList}(\mathsf{w}_1)$ stores $(c, \mathsf{id}, \mathrm{e})$ entries. This $(c, \mathsf{id}, \mathrm{e})$ entries are constructed such that c in $\mathsf{T}[\mathsf{w}_1]$ was assigned an id during Search and the associated ciphertext e. The game starts with empty TList and populates using the $\mathsf{id} \in \mathbf{DB}(q)$. G_8 can check that an $\mathsf{id} \in \mathbf{DB}(q)$ belongs to $\mathsf{TList}(\mathsf{w}_1)$ or not. For any id not in TList, G_8 assigns a random values of c in $[1, \mathsf{TSet}L$ not used so far. It computes (c, id) for a ciphertext e using K_e and K_S from w_1 of G_9. This view of G_{12} is essentially the same as of G_{11} and matches the combined description of Algorithm 6, 7, and 8 and therefore G_{12} is indistinguishable from G_{11}.

A.2 Security against Adversarial Server

We present proof of Theorem 2 here that defines the security against the adversarial server. The proof mainly follows from ODXT with modifications following

NOMOS and leakage function $\mathcal{L}^{\mathcal{S}}_{\text{NOMOS}}$. The proof essentially follows the proof outlined in [30] with modifications for NOMOS.

The proof follows a series of games, similar to the approach of malicious clients, where we start with the $\mathbf{Real}^{\Pi}_{\mathcal{A}}$ execution (game G_0 is identical to the real execution), and the last game is identical to $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\text{SIM}}$ (identical to game G_8). We denote the probability of a game G_i to output $b = 1$ by p_i.

Game G_1. This game replaces the OPRF instances of the form $OPRF(K_T, \cdot)$ with a randomly sampled value from $\mathbb{G}$ during transcript generation in UPDATE and SEARCH.

Lemma 12. *G_1 and G_0 are indistinguishable with probability $p_1 \approx p_0$.*

Proof. By indistinguishablity property of OPRF from OMGDH assumption, $p_1 \approx p_0$.

Game G_2. In this game, the OPRF instances of the form $OPRF(K_X, \cdot)$ are replaced by randomly sampled values from $\mathbb{G}$ during transcript generation in UPDATE and SEARCH.

Lemma 13. *Game G_2 and G_1 are indistinguishable with probability $p_2 \approx p_1$.*

Proof. By the indistinguishability property of OPRF from OMGDH assumption, $p_2 \approx p_1$.

Game G_3. This game replaces the PRF instances of the form $F_p(K_Y, \cdot)$ with a random function in the range $\mathbb{Z}_p^*$.

Lemma 14. *Game G_3 and G_2 are indistinguishable with probability $p_3 \approx p_2$.*

Proof. By the indistinguishability property of PRF, G_3 is indistinguishable from G_2 with probability $p_3 \approx p_2$.

Game G_4. In this game, the PRFs of the form $F_p(K_Z, \cdot)$ are replaced by a random sample from $\mathbb{Z}_p^*$.

Lemma 15. *Game G_4 is indistinguishable from G_3 with probability $p_4 \approx p_3$.*

Proof. $p_4 \approx p_3$ from the indistinguishability property of PRF.

Game G_5. This game modifies the way xtokens are generated. For a conjunctive query $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$, the game looks up the history updates to retrieve the set of update operations $(\mathsf{op}_j, (\mathsf{w}_1, \mathsf{id}_j))$ for the s-term w_1. Furthermore, for each x-term and the update operation $(\mathsf{op}_j, (\mathsf{w}_1, \mathsf{w}_j))$, the game computes the dynamic blinding factors $\alpha_{i,j}$ and the $\mathsf{xtag}_{i,j}$ following ODXT construction and generates $\mathsf{xtoken}_{i,j} = \mathsf{xtag}^{1/\alpha_{i,j}}$. (Rewrite!)

Lemma 16. *Game G_5 is identical to G_4 with probability $p_5 \approx p_4$.*

Proof. The xtoken values in G_5 and G_4 are identically distributed. For a $(\alpha_{i,j}, \mathsf{xtag}_{i,j}, \mathsf{xtoken}_{i,j})$ in G_4, $\mathsf{xtoken}_{i,j} = \mathsf{xtag}_{i,j}^{1/\alpha_{i,j}}$, that is same as of the G_5. Therefore, G_5 and G_4 are indistinguishable and $p_5 \approx p_4$. (Rewrite!)

Game G_6. The game modifies the α values by using randomly sampled values from $\mathbb{Z}_p^*$.

Lemma 17. *Game G_6 is identical from G_5 with probability $p_6 \approx p_5$.*

Proof. In G_4 the PRFs are replaced with random samples from $\mathbb{Z}_p^*$. This operation is done only once for each Update invocation for xtoken generation, and never repeated for any queries. Also, it is not evaluated on two same values for two different Update invocations. The α values are generated by multiplying a randomly sampled $\mathbb{Z}_p^*$ element in place of PRF with the inverse of a value in $\mathbb{Z}_p$ obtained in the same way. Hence, the distribution of α in G_6 is indistinguishable from G_5 with $p_6 \approx p_5$.

Game G_7. This game modifies the way xtags are generated during transcript generation of Update. It samples a $\gamma \leftarrow \mathbb{Z}_p^*$ and computes $\mathsf{xtag} = g^\gamma$, where g is a generator of $\mathbb{G}$.

Lemma 18. *Game G_7 is indistinguishable from G_6.*

To prove Lemma 18, we construct an alternative version below.

Lemma 19. *Game G_7 and G_6 are indistinguishable following the polynomial equivalence of the DDH assumption and the extended DDH assumption over any group $\mathbb{G}$.*

Proof. The xtag values for an update operation $(\mathsf{op}, (\mathsf{id}_j, \mathsf{w}_i))$ are computed in the following way in G_7 $\mathsf{xtag}_{i,j,\mathsf{op}} = g^{G_X(\mathsf{w}_i)\cdot G_Y(\mathsf{id}_j||\mathsf{op})}$ where $G_X(\cdot)$ and $G_Y(\cdot)$ are random functions uniformly sampled from the set of all functions mapping λ-bit strings to $\mathbb{Z}_p^*$ replacing $OPRF(K_X, \cdot)$ and $F_p(K_Y, \cdot)$, and g is the generator of group $\mathbb{G}$. We rewrite the expression as $\mathsf{xtag}_{i,j,\mathsf{op}} = g^{\alpha_i \cdot \beta_{j,\mathsf{op}}}$ where $\alpha_i = G_X(\mathsf{w}_i)$ and $\beta_{j,\mathsf{op}} = (\mathsf{id}_j||\mathsf{op})$. Whereas, in previous game we had $\mathsf{xtag}_{i,j,\mathsf{op}} = g^{\gamma_{i,j,\mathsf{op}}}$ where $\gamma_{i,j,\mathsf{op}} \leftarrow \mathbb{Z}_p^*$. The distribution of xtags are indistinguishable from G_6. Therefore, G_7 and G_6 are indistinguishable with probability $p_7 \approx p_6$.

Game G_8. Game G_8 is identical to G_7 with the following modifications. During transcript generation for Update operation, each $OPRF(K_T, \mathsf{w}||\mathsf{cnt}||b), b \in \{0,1\}$ instance is replaced with a random function of the form $G_T(t)$ where t is the timestamp of the particular update operation.

Lemma 20. *Game G_8 is indistinguishable from G_7 with probability $p_8 \approx p_7$.*

Proof. Note that $G_T(t)$ is never evaluated twice on the same input as the real instance has an increasing counter appended to the input, and $G_T(\cdot)$ is uniformly randomly sampled from the set of all λ-bit functions mapping to λ-bit values. Therefore, G_8 is indistinguishable from G_7.

Game G_9. In this game, simulator SIM$'$ replaces the challenger. SIM$'$ does not have access to the actual queries by $\mathcal{A}$. It uses the the following leakages for each update or conjunctive queries issued by $\mathcal{A}$.

SIM$'$ receives empty leakage from UPDATE. It uses the timestamp information to generate the TSet entries similar to the procedure in G_8. It samples uniformly random blinding factors α, and $\gamma \leftarrow \mathbb{Z}_p^*$ and computes $\mathsf{xtag} = g^\gamma$.

SIM$'$ learns the number of updates involving w_1 (the s-term) and the timestamp of these updates. It uses this information to simulate the xtoken computation similar to G_8.

Similarly, SIM$'$ uses the aforementioned information learned to compute $\mathsf{xtoken}_{i,j} = \mathsf{xtag}_{i,j}^{1/\alpha_{i,j}}$. Also, SIM$'$ learns whether two queries have the same s-term from the equality pattern and generates stoken values across multiple queries accordingly (consistently). Apart from these, SIM learns the set of ids in the final result.

Lemma 21. *Game G_9 is indistinguishable from G_8.*

Proof. The transcripts generated by SIM$'$ is identical to the one generated by $\mathcal{A}$ for each update and conjunctive queries. Therefore, game G_9 is identical to G_8.

Game G_{10}. In this game, SIM$'$ is replaced with SIM that has access to the leakage function $\mathcal{L}_\mathcal{S}$ as below.

$$\mathcal{L}_\mathcal{S} = \{\mathcal{L}_\mathcal{S}^{\text{SETUP}}, \mathcal{L}_\mathcal{S}^{\text{UPDATE}}, \mathcal{L}_\mathcal{S}^{\text{UPDATE}}\}$$

where $\mathcal{L}_\mathcal{S}^{\text{SETUP}} = \perp$, $\mathcal{L}_\mathcal{S}^{\text{UPDATE}}(\mathsf{op}, (\mathsf{w}, \mathsf{id})) = \perp$, and $\mathcal{L}_\mathcal{S}^{\text{UPDATE}}(q) = \{\mathsf{TimeDB}(q), \text{UPD}(q)\}$, as defined in Section 6.

Lemma 22. *Game G_{10} is indistinguishable from G_9.*

We construct the following Lemma to prove Lemma 22.

Lemma 23. SIM *can efficiently execute* SIM$'$ *from G_9 as a sub-routine.*

Proof. The proof essentially involves demonstrating that the leakage profile of SIM covers the leakage profile of SIM$'$.

SIM has access to the same empty update leakage of SIM$'$. Further, SIM UPD(q) leakage is covered by UPD(w_1) (the s-term). The equality pattern leakage in G_9 is covered by UPD(q_1) and UPD(q_2) for two different queries q_1 and q_2.

Hence, G_{10} is indistinguishable from G_9 and is identical to the ideal execution of the protocol.

B Forward and Backward Privacy of Nomos

Formal forward and backward privacy definition was introduced by Bost et al. [5], which we follow in the analysis below.

B.1 Forward Privacy of Nomos

Given the leakage profile of an adaptively secure dynamic conjunctive SSE

$$\mathcal{L} = (\mathcal{L}^{\textsc{Setup}}, \mathcal{L}^{\textsc{Update}}, \mathcal{L}^{\textsc{Search}})$$

adaptive forward privacy states that the update leakage can be expressed as

$$\mathcal{L}^{\textsc{Update}}(\mathsf{op}, (\mathsf{w}, \mathsf{id})) = \mathcal{L}'(\mathsf{op}, \mathsf{id})$$

where $\mathcal{L}'$ is a stateless function and $(\mathsf{op}, (\mathsf{w}, \mathsf{id}))$ is an arbitrary triplet. This essentially implies that the update operation hides the w being updated and therefore can not be linked to any search query containing w by a polynomially bounded $\mathcal{A}$.

In this context, Nomos Update leakage is

$$\mathcal{L}^{\textsc{Update}}(\mathsf{op}, (\mathsf{w}, \mathsf{id})) = \perp$$

as discussed in Section 6. Therefore, Nomos hides the w as well as the id from the $(\mathsf{op}, (\mathsf{w}, \mathsf{id}))$ input involved in the update process. The following corollary is straightforward from Theorem 2.

Corollary 1. *Provided* Nomos *is instantiated with DH OPRF and DH assumption holds in $\mathbb{G}$, F_p and F are secure PRFs, and (*AuthEnc,AuthDec*) is an IND-CPA and strongly UF-CMA-secure AE scheme, and all hash functions are modelled using the Random Oracle Model,* Nomos *is adaptively forward private.*

B.2 Backward Privacy of Nomos

Given the leakage profile of an adaptively secure dynamic conjunctive SSE

$$\mathcal{L} = (\mathcal{L}^{\textsc{Setup}}, \mathcal{L}^{\textsc{Update}}, \mathcal{L}^{\textsc{Search}})$$

adaptive type-II backward privacy states that the update and search leakages can be expressed as

$$\mathcal{L}^{\textsc{Update}}(\mathsf{op}, (\mathsf{w}, \mathsf{id})) = \mathcal{L}''(\mathsf{op}, \mathsf{id})$$
$$\mathcal{L}^{\textsc{Search}}(\mathsf{w}) = \mathcal{L}''(\mathsf{TimeDB}(\mathsf{w}), \textsc{Upd}(\mathsf{w}))$$

.

For Nomos, the update and search leakage has the following profile.

$$\mathcal{L}^{\textsc{Update}}(\mathsf{op}, (\mathsf{w}, \mathsf{id})) = \perp$$
$$\mathcal{L}^{\textsc{Search}}(q) = (\mathsf{TimeDB}(q), \textsc{Upd}(q))$$

for a conjunctive query q, which is extended to the conjunctive keyword setting from single keyword setting. Therefore, the following corollary is immediate from Theorem 2.

Corollary 2. *Provided* Nomos *is instantiated with DH OPRF and DH assumption holds in $\mathbb{G}$, F_p and F are secure PRFs, and (*AuthEnc,AuthDec*) is an IND-CPA and strongly UF-CMA-secure AE scheme, and all hash functions are modelled using the Random Oracle Model,* Nomos *is adaptively backward private.*

C Redundant Bloom Filter Construction

Bloom filter is a probabilistic data structure with the following insertion (Algorithm 9) and query (Algorithm 10) routines.

Algorithm 9 Bloom Filter Insert

Input: Input parameters: x - element to be inserted into BF
Output: Output parameters:

```
function BF.INSERT(x)
    Gate-keeper
    Select k hash functions {h_1, ..., h_k} for BF indices
    Initialise empty index set BFIdxSet
    for i ← 1 to k do
        bfidx_i ← h_i(x)
        BFIdxSet = BFIdxSet ∪ {bfidx_i}
    Shuffle elements in BFIdxSet
    Send BFIdxSet to the server
    Server
    for Each idx ∈ BFIdxSet do
        Set BF[idx] = 1
```

Algorithm 10 Bloom Filter Query

Input: Input parameters: x - element to be queried in **BF**
Output: Output parameters: True/False

```
function BF.QUERY(x)
    Client
    Select k hash functions {h_{i_1}, ..., h_{i_k}} for BF indices
    Initialise empty index set BFIdxSet
    for j ∈ {i_1, ..., i_k} do
        bfidx_j ← h_j(x)
        BFIdxSet = BFIdxSet ∪ {bfidx_j}
    Shuffle elements in BFIdxSet
    Send BFIdxSet to the server
    Server
    for Each idx ∈ BFIdxSet do
        if BF[idx] ≠ 1 then
            Return False
    Return True
```

Note that, if we directly plug this into NOMOS$_{\text{BASIC}}$, replacing the XSet insert with BF.INSERT and XSet retrieve with BF.QUERY, the construction essentially works the same and the security properties remain unchanged. The leakage is not mitigated the BFIDX$_i$ generated in BF are deterministically generated using

k hash functions. Hence, the server still can correlate a $(\mathsf{w}, \mathsf{id})$ pair from previous search with later update operation.

We modify this basic BF construction to allow storing redundant elements (total ℓ indices generated from ℓ hash functions) to be stored for each xtag (corresponding to each $(\mathsf{w}, \mathsf{id})$ pair). During query, we access only a random subset of size k of these ℓ indices. Such the server "sees" each time different k indices are being accessed and can not correlate previous accesses. We call this redundant element-based Bloom filter construction as Redundant Bloom Filter (RBF). The updated RBF.INSERT and RBF.QUERY routines are presented in Algorithm 11 and Algorithm 12 respectively.

Algorithm 11 Redundant Bloom Filter Insert

Input: Input parameters: x - element to be inserted into RBF
Output: Output parameters:

```
function RBF.INSERT(x)
    Gate-keeper
    Select ℓ hash functions {h_1, ..., h_ℓ} for RBF indices
    Initialise empty index set RIdxSet
    for i ← 1 to ℓ do
        rbfidx_i ← h_i(x)
        RIdxSet = RIdxSet ∪ {rbfidx_i}
    Shuffle elements in RIdxSet
    Send RIdxSet to the server
    Server
    for Each idx ∈ RIdxSet do
        Set RBF[idx] = 1
```

RBF Overhead In RBF, the server sets ℓ locations, and accesses k locations where $\ell > k$. The value of k needs to be large enough to have negligible false positive probability (similar to normal BF).

Storage Overhead. Traditional BF requires

$$k \cdot \sum_{\mathsf{w} \in \Delta} |\mathbf{DB}(\mathsf{w})|$$

storage for BF with k hashes. Here, we have k hashes during insert, and ℓ hashes during queries. Hence, the storage requirement of RBF is

$$\ell \cdot \sum_{\mathsf{w} \in \Delta} |\mathbf{DB}(\mathsf{w})|$$

.

RBF storage overhead is $\frac{\ell}{k}$ times (greater than one as $\ell > k$) than BF for the same database.

Algorithm 12 Redundant Bloom Filter Query

Input: Input parameters: x - element to be queried in **RBF**
Output: Output parameters: True/False

function **RBF**.QUERY(x)
  Client
  Select k hash functions $\{h_{i_1}, \ldots, h_{i_k}\}$ for **RBF** indices
  Initialise empty index set RIdxSet
  for $j \in \{i_1, \ldots, i_k\}$ **do**
    $\mathsf{rbfidx}_j \leftarrow h_j(\mathsf{x})$
    $\mathsf{RIdxSet} = \mathsf{RIdxSet} \cup \{\mathsf{rbfidx}_j\}$
  Shuffle elements in RIdxSet
  Send RIdxSet to the server
  Server
  for Each $\mathsf{idx} \in \mathsf{RIdxSet}$ **do**
    if $\mathbf{RBF}[\mathsf{idx}] \neq 1$ **then**
      Return False
  Return True

Communication Overhead. We need to send k indices for a single cross tag while inserting into and querying on BF. Thus, the communication overhead can be expressed as $O(1)$ (from $O(k)$) for each cross tag (as k remains constant for a particular database). For a complete query $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$, the query overhead can be expressed as follows.

$$k \cdot \sum_{\mathsf{w} \in \{\mathsf{w}_2, \ldots, \mathsf{w}_n\}} |\mathbf{DB}(\mathsf{w}_1)|$$

With RBF, the communication overhead during inserting a single cross tag is $O(1)$ (from $O(\ell)$) as ℓ remains constant for a database. For a conjunctive query of the form $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$, the communication overhead can be estimated as follows,

$$k \cdot \sum_{\mathsf{w} \in \{\mathsf{w}_2, \ldots, \mathsf{w}_n\}} |\mathbf{DB}(\mathsf{w}_1)|$$

since k indices are used for query (instead of all ℓ indices). Clearly, the communication overhead of RBF is $\frac{\ell}{k}$ times than BF (greater than one as $\ell > k$). However, if same k values are chosen for RBF and BF, the communication overhead essentially remains the same for both RBF and BF.

D MC-ODXT Construction Details

We present MC-ODXT construction algorithm in this Appendix. MC-ODXT is an extension of ODXT [30] to multi-client MRSW setting following the OPRF-based approach of OSPIR-OXT [22].

Algorithm 13 MC-ODXT SETUP

function MC-ODXT.SETUP
  Gate-keeper
  Sample a uniformly random key K_S from $\mathbb{Z}_p^*$ for OPRF
  Sample two sets of uniformly random keys $K_T = \{K_T^1, \ldots, K_T^d\}$ and $K_X = \{K_X^1, \ldots, K_X^d\}$ from $(\mathbb{Z}_p^*)^d$ for OPRF
  Sample uniformly random key K_Y from $\{0,1\}^\lambda$ for PRF F_p
  Sample shared uniformly random key K_M from $\{0,1\}^\lambda$ for AE
  Initialise $\mathsf{UpdateCnt}$, TSet, XSet to empty maps
  Gate-keeper keeps $\mathsf{sk} = (K_S, K_T, K_X, K_Y)$; $\mathsf{UpdateCnt}$ is disclosed to clients when required, and K_M is shared between gate-keeper and the server
  Set **EDB** $= (\mathsf{TSet}, \mathsf{XSet})$
  Send **EDB** to server

Algorithm 14 MC-ODXT UPDATE

Input: K_S, $K_T = \{K_T^1, \ldots, K_T^d\}$, $K_X = \{K_X^1, \ldots, K_X^d\}$, accessed as $K_T[I(\mathsf{w})]$ and $K_X[I(\mathsf{w})]$ for attribute $I(\mathsf{w})$ of w, K_Y, (w,id) pair to be updates, update operation op
Output: Updated **EDB**

function MC-ODXT.UPDATE
  Gate-keeper
  Parse (K_T, K_X, K_Y) and $\mathsf{UpdateCnt}$
  Set $K_Z \leftarrow F((H(\mathsf{w}))^{K_S}, 1)$
  If $\mathsf{UpdateCnt}[\mathsf{w}]$ is `NULL` then set $\mathsf{UpdateCnt}[\mathsf{w}] = 0$
  Set $\mathsf{UpdateCnt}[\mathsf{w}] = \mathsf{UpdateCnt}[\mathsf{w}] + 1$
  Set $\mathsf{addr} = (H(\mathsf{w}||\mathsf{UpdateCnt}[\mathsf{w}]||0))^{K_T[I(\mathsf{w})]}$
  Set $\mathsf{val} = (\mathsf{id}||\mathsf{op}) \oplus (H(\mathsf{w}||\mathsf{UpdateCnt}[\mathsf{w}]||1))^{K_T[I(\mathsf{w})]}$
  Set $\alpha = F_p(K_Y, \mathsf{id}||\mathsf{op}) \cdot (F_p(K_Z, \mathsf{w}||\mathsf{UpdateCnt}[\mathsf{w}])^{-1})$
  Set $\mathsf{xtag} = H(\mathsf{w})^{K_X[I(\mathsf{w})] \cdot F_p(K_Y, \mathsf{id}||\mathsf{op})}$
  Send $(\mathsf{addr}, \mathsf{val}, \alpha, \mathsf{xtag}$ to server
  Server
  Parse **EDB** $= (\mathsf{TSet}, \mathsf{XSet})$
  Set $\mathsf{TSet}[\mathsf{addr}] = (\mathsf{val}, \alpha)$
  Set $\mathsf{XSet}[\mathsf{xtag}] = 1$

Algorithm 15 MC-ODXT GenToken

Input: $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$. $\mathcal{P}$ is the set of allowable attribute sequences, K_S, K_T, K_X, K_M

Output: strap, $\mathsf{bstag}_1, \cdots, \mathsf{bstag}_m$, $\delta_1, \cdots, \delta_m$, $\mathsf{bxtrap}_2, \cdots, \mathsf{bxtrap}_n$, env

function MC-ODXT.GenToken

Client

Pick $r_1, \cdots, r_n \xleftarrow{\$} \mathbb{Z}_p^*$
Set $m = \mathsf{UpdateCnt}[\mathsf{w}_1]$
Pick $s_1, \cdots, s_m \xleftarrow{\$} \mathbb{Z}_p^*$
Set $a_j \leftarrow (H(\mathsf{w}_j))^{r_j}$, for $j = 1, \cdots, n$
Set $b_j \leftarrow (H(\mathsf{w}_1 || j || 0))^{s_j}$, for $j = 1, \cdots, m$
Set $c_j \leftarrow (H(\mathsf{w}_1 || j || 1))^{s_j}$, for $j = 1, \cdots, m$
Set $\mathsf{av} = (I(\mathsf{w}_1), \cdots, I(\mathsf{w}_n)) = (I_1, \ldots, I_n)$

Gate-keeper

Abort if $\mathsf{av} \notin \mathcal{P}$ ▷ *Abort if attributes do not match*
Pick $\rho_1, \cdots, \rho_n \xleftarrow{\$} \mathbb{Z}_p^*$
Pick $\gamma_1, \cdots, \gamma_m \xleftarrow{\$} \mathbb{Z}_p^*$
Set $\mathsf{strap}' \leftarrow (a_1)^{K_S}$
Set $\mathsf{bstag}'_j \leftarrow (b_j)^{K_T[I_1] \cdot \gamma_j}$, for $j = 1, \cdots, m$
Set $\delta'_j \leftarrow (c_j)^{K_T[I_1]}$, for $j = 1, \cdots, m$
Set $\mathsf{bxtrap}'_j \leftarrow (a_j)^{K_X[I_j] \cdot \rho_j}$ for $j = 2, \cdots, n$
Set $\mathsf{env} = \mathbf{AuthEnc}_{K_M}(\rho_1, \cdots, \rho_n, \gamma_1, \cdots, \gamma_m)$
Send strap', $\mathsf{bstag}'_1, \cdots, \mathsf{bstag}'_m$, $\delta'_1, \cdots, \delta'_m$, $\mathsf{bxtrap}'_2, \cdots, \mathsf{bxtrap}'_n$, env to Client

Client

Set $\mathsf{strap} \leftarrow (\mathsf{strap}')^{r_1^{-1}}$
Set $\mathsf{bstag}_j \leftarrow (\mathsf{bstag}'_j)^{s_j^{-1}}$, for $j = 1, \cdots, m$
Set $\delta_j \leftarrow (\delta'_j)^{s_j^{-1}}$, for $j = 1, \cdots, m$
Set $\mathsf{bxtrap}_j \leftarrow (\mathsf{bxtrap}'_j)^{r_j^{-1}}$, for $j = 2, \cdots, n$
Output (strap, $\mathsf{bstag}_1, \cdots, \mathsf{bstag}_m$, $\delta_1, \cdots, \delta_m$, $\mathsf{bxtrap}_2, \cdots, \mathsf{bxtrap}_n$, env as search token

Algorithm 16 MC-ODXT SEARCH

Input: strap, $\mathsf{bstag}_1, \cdots, \mathsf{bstag}_m$, $\delta_1, \cdots, \delta_m$, $\mathsf{bxtrap}_2, \cdots, \mathsf{bxtrap}_n$, env, $\mathsf{UpdateCnt}$

Output: IdList

function MC-ODXT.SEARCH
    Client
    Set $K_Z \leftarrow F(\mathsf{strap}, 1)$
    $m = \mathsf{UpdateCnt}[\mathsf{w}_1]$
    Initialise $\mathsf{stokenList}$ to an empty list
    Initialise $\mathsf{xtokenList}_1, \cdots, \mathsf{xtokenList}_m$ to empty lists
    for $j = 1$ to m **do**
        $\mathsf{stokenList} = \mathsf{stokenList} \cup \{\mathsf{bxtrap}_j\}$
        for $i = 2$ to n **do**
            Set $\mathsf{xtoken}_{i,j} = \mathsf{bxtrap}_i{}^{F_p(K_Z, w_1||j)}$
            Set $\mathsf{xtokenList}_j = \mathsf{xtokenList}_j \cup \mathsf{xtoken}_{i,j}$
        Randomly permute the tuple-entries of $\mathsf{xtokenList}_j$
    Send $(\mathsf{stokenList}, \mathsf{xtokenList}_1, \cdots, \mathsf{xtokenList}_m)$
    Server
    Upon receiving env from client, verify env; if verification fails, return $\perp$; otherwise decrypt env
    Parse $\mathbf{EDB} = (\mathsf{TSet}, \mathsf{XSet})$
    Initialise $\mathsf{sEOpList}$ to an empty list
    for $j = 1$ to $\mathsf{stokenList}.size$ **do**
        Set $\mathsf{cnt}_j = 1$
        Set $\mathsf{stag}_j \leftarrow (\mathsf{stokenList}[j])^{1/\gamma_j}$
        Set $(\mathsf{sval}_j, \alpha_j) = \mathsf{TSet}[\mathsf{stag}_j]$
        for $i = 2$ to n **do**
            Set $\mathsf{xtoken}_{i,j} = \mathsf{xtokenList}_j[i]$
            Compute $\mathsf{xtag}_{i,j} = (\mathsf{xtoken}_{i,j})^{\alpha_j/\rho_i}$
            If $\mathsf{XSet}[\mathsf{xtag}_{i,j}] == 1$, then set $\mathsf{cnt}_j = \mathsf{cnt}_j + 1$
        Set $\mathsf{sEOpList} = \mathsf{sEOpList} \cup \{(j, \mathsf{sval}_j, \mathsf{cnt}_j)\}$
    Sent $\mathsf{sEOpList}$ to client
    Client
    Initialise IdList to an empty list
    for $\ell = 1$ to $\mathsf{sEOpList}.size$ **do**
        Let $(j, \mathsf{sval}_j, \mathsf{cnt}_j) = \mathsf{sEOpList}[\ell]$
        Recover $(\mathsf{id}_j || \mathsf{op}_j) = \mathsf{sval}_j \oplus \delta_\ell$
        If op_j is ADD and $\mathsf{cnt}_j = n$ then set $\mathsf{IdList} = \mathsf{IdList} \setminus \{\mathsf{id}_j\}$
    Output IdList