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

Arnab Bag
Indian Institute of Technology Kharagpur
Kharagpur, India

Sikhar Patranabis
IBM Research India
Bengaluru, India

Debdeep Mukhopadhyay
Indian Institute of Technology Kharagpur
Kharagpur, India

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 efficient 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 a multi-reader-single-writer construction that allows only the gate-keeper (or the data-owner) - a trusted entity in the Nomos framework, to update the encrypted database stored on the adversarial server. Nomos achieves forward and type-II 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 the state-of-the-art static conjunctive SSE schemes in practice.

CCS CONCEPTS

• **Security and privacy** → **Cryptography**; **Management and querying of encrypted data**.

KEYWORDS

Searchable encryption, Dynamic, Multi-client

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0482-6/24/07
https://doi.org/10.1145/3634737.3657018

ACM Reference Format:
Arnab Bag, Sikhar Patranabis, and Debdeep Mukhopadhyay. 2024. Tokenised Multi-client Provisioning for Dynamic Searchable Encryption with Forward and Backward Privacy. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3634737.3657018

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.

Several elegant cryptographic primitives such as Fully Homomorphic Encryption (FHE) [17, 18], Functional Encryption (FE) [4], Oblivious RAM (ORAM) [20, 21] and Private Information Retrieval (PIR) [2, 13, 29] allow implementing diverse functionalities over encrypted outsourced data. However, all of these approaches either incur prohibitively heavy computation/storage 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 untrusted parties. The benign leakage in SSE is formally 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) [6–12, 14–16, 19, 24–27, 30, 32] allows querying an encrypted database privately without decryption. In contrast to the abovementioned approaches, SSE offers efficient and low-bandwidth constructions for encrypted data processing, especially searching, while leaking only benign information to untrusted parties. Fundamentally, an SSE scheme offers the following capabilities.

• Allow an (or many) entity (potentially untrusted client) to efficiently search encrypted queries over an 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 the actual associated (encrypted) data. The database size, *query pattern* (the set of queries corresponding to the same keyword), and the *access pattern* (the set of database records matching a given query) are a few such leakages typically studied in the context of SSE. We present formal syntax of SSE with elaborate details in Section 2, and elaborate the study of these leakages in the full version of the paper[1] due lack of space.

Dynamic SSE. A dynamic SSE construction [6–8, 11, 15, 25, 26] allows dynamic updates (adding or deleting records) 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 [34].

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 the 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. SRSW constructions [8, 9, 11, 12, 14, 16, 19, 24–27, 32] have been extensively studied in the literature. A number of dynamic SRSW schemes [6–8, 11, 15, 25, 26] 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.

Multi-Reader–Single-Writer (MRSW): In the multi-reader–single-writer or MRSW setting, multiple clients can 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.

Ideally, cloud applications require multi-client access and dynamic update capability to cater to applications over diverse databases involving a potentially large number of users (or clients). Unfortunately, the literature on practically efficient MRSW and MRMW SSE schemes is sparse and entirely restricted to the static setting [5, 23], which prompts us to raise the following question.

[1] https://eprint.iacr.org/2024/573

Is there a dynamic MRSW/MRMW SSE scheme with strong forward and backward privacy, linear storage requirements and sublinear search overheads – 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 (assumed to be semi-honest) to the data-owner (a trusted party) retains this leakage. 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 the widespread adoption of SSE in encrypted processing tasks on the cloud.

This work aims to bridge this gap between secure and practically efficient SSE constructions and real multi-client cloud applications. We summarise our goals with the following question.

Can we design an efficient dynamic SSE scheme with forward and backward privacy in the MRSW setting?

We show in this paper that it is possible to design such a scheme, and we present Nomos 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 with forward and type-II backward privacy [7], 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 obtain search tokens from a trusted entity called gate-keeper, which 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 secret keys for token generation among multiple potentially semi-honest clients.

❷ **Leakage mitigation**. The Nomos construction mitigates a particular leakage originating from the cross-terms of conjunctive queries. This leakage is inherently present in the state-of-the-art ODXT scheme, and we discuss an attack flow that shows that the

trivial extension of ODXT to the multi-client setting remains vulnerable to this leakage that potentially breaks the scheme. 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 the repeated memory accesses (for XSet implemented using RBF) to mitigate the cross-term leakage. Using RBF has a 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 the *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. 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 us 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-user cloud applications where requests arrive asynchronously, and the gate-keeper/cloud needs to serve as early as possible, reducing waiting time (for example, assigning doctors appointments based on patient details in medical applications). It can be easily adopted into general cloud search applications where strong access control is necessary, such as office employee records and bank operations, to name a few. The tokenised functionality allows fine-grained user management and enforcing access permissions for each user individually in a multi-client setting.

❹ **Security analysis and implementation**. We provide a concrete security analysis of 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 semi-honest entities individually; that is, a semi-honest client follows the specified token generation and search protocol but can obtain/share additional information regarding the queries issued or the tokens received. We provide an overview of the Nomos leakage profile and concrete security analysis in the full version of the paper[1] due to lack of space.

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 5. The experimental results show that Nomos achieves linear storage overhead and sublinear search time, 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 the multi-client setting to demonstrate the devastating effect of cross-term leakage on a multi-client SSE construction. We outline the required security notions and challenges of designing multi-client SSE in Section 3.3. We present our main Nomos construction in Section 4. Finally, the experimental results and related discussion are provided in Section 5, and we end with a concluding remark on our work. We have diverted the formal security analysis to the full version of the paper[1] due to lack of space.

[2]https://redis.io/

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

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.

2.1 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 and N is number of ws in Δ. We represent the plain database as **DB**, and $\mathbf{DB}(q)$ represents the set of ids satisfying a conjunctive query q over **DB**. Similarly, for a single w, $\mathbf{DB}(\mathrm{w})$ 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 ξ is expressed as $v \xleftarrow{\$} \xi$. We denote a negligible function as $\mathsf{negl}()$. We denote the attribute of a w using $I(\mathrm{w})$, which is essentially encoded as an index. More precisely, I can be considered as a list of (w, a) pairs, where $I(\mathrm{w})$ returns the attribute a, where $\mathrm{w} \in \Delta$ and a is a valid keyword attribute. We represent a set of valid keyword attribute combinations using $\mathcal{P}$, from all unique keyword attributes for all ws in Δ. We denote the number of all unique keyword attributes using d. We also assume that during an update, a complete document (containing multiple keywords) is replaced, and in this process, the 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 C to represent a client and $\mathbf{C} = \{C_1, \ldots, C_t\}$ to represent a set of t clients. We represent a data-owner by the symbol $\mathcal{D}$. We denote the server using $\mathcal{S}$ and the gate-keeper using $\mathcal{G}$ (explained in Section 4). In a single client setting, $\mathcal{D}$ and $\mathcal{G}$ serve the same purpose, and we use the $\mathcal{D}$ symbol in the context of single client constructions. However, in an MRSW multi-client setting, $\mathcal{G}$ has the additional responsibility of generating search tokens for Cs. Thus, we denote the single entity responsible for database update in SRSW constructions, such as OXT, ODXT, by $\mathcal{D}$, and we denote the single entity responsible for database update *and search token generation for other clients*, as in MC-ODXT or Nomos, by $\mathcal{G}$. We denote a polynomial-time adversary by $\mathcal{A}$ and a simulator using SIM.

2.2 Cryptographic Primitives

We denote a pseudo-random function (PRF) by $F(K, \cdot)$ and a specific version mapping to $\mathbb{F}_p$ as $F_p(K, \cdot)$. 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) [3, 31] scheme with the routines AE = {AE.Enc,AE.Dec} that is IND-CPA and and strongly UF-CMA-secure (unforgeability guarantee) [3].

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 a cryptographic 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. At the end of the protocol, A receives the output c, 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 $y = t^{-1}$. We represent this OPRF evaluation as $y = \mathsf{OPRF}(K, x)$. DH-OPRF is used as a core primitive in our construction to allow multi-client search. Please refer [28] for more details on DH-OPRF[4]. Note that the main Nomos construction can be instantiated with a generic OPRF. We opted for DH-OPRF for the ease of analytical exposition and implementation.

2.3 SSE Syntax and Security Definition

We denote an SSE algorithm simply using SSE. We use **EDB** to denote the encrypted database stored on the encrypted server. We use $\mathcal{R}_q$ to denote the set of encrypted records returned upon searching 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. We call w_1 as the special-term or s-term, and $\mathsf{w}_2, \ldots, \mathsf{w}_n$ are denoted as the cross-terms or x-terms.

A dynamic MRSW SSE scheme SSE with tokenised multi-client search is defined by the following ensemble of algorithms.

• **SSE.Setup(1^λ)**. This is a PPT algorithm run by $\mathcal{G}$ (who is the data-owner) that samples the master security key sk, the AE key K_M, and initialises the encrypted database **EDB**.

• **SSE.Update(sk, {op, (w, id)}; EDB)**. This is a PPT algorithm jointly executed by $\mathcal{G}$ and $\mathcal{S}$, where $\mathcal{G}$'s input is secret key sk, the update record (w, id) and the type of the update op = {add, del}, and $\mathcal{S}$s input is **EDB**. At the end of the protocol, **EDB** stored on $\mathcal{S}$ is updated with the new record.

[4]The constructions in this manuscript can be instantiated with a general syntax of OPRF. We opt for the specific instance of DH-OPRF for the ease of exposition with the existing construction structure of ODXT [30] and OSPIR-OXT [23].

• **SSE.GenToken(q, sk)**. The GenToken routine is executed by a client $C_j \in C$ and $\mathcal{G}$, where C_j's input is a query q, and $\mathcal{G}$'s input is the secret key sk, and at the end of execution, C_j receives the search token tk.

• **SSE.Search(tk; EDB)**. In this protocol, a client C_j sends the search token tk obtained from $\mathcal{G}$ using GenToken method to $\mathcal{S}$, and $\mathcal{S}$ looks up **EDB** to return the set of matched encrypted records $\mathcal{R}_q$ to C_j.

An SSE scheme is said to be correct if the SSE.Search routine returns all the matching encrypted ids from **EDB** for a query q.

Security of MRSW SSE. A dynamic MRSW SSE scheme SSE as described above is said to be secure if it follows the security properties stated below.

Security against a semi-honest client. The following leakage function parameterises the security of a dynamic MRSW SSE construction against a semi-honest client.

$$\mathcal{L}_C = \{\mathcal{L}_C^{\text{Setup}}, \mathcal{L}_C^{\text{Update}}, \mathcal{L}_C^{\text{GenToken}}, \mathcal{L}_C^{\text{Search}}\}$$

In this ensemble, $\mathcal{L}_C^{\text{Setup}}$ encapsulates the leakage to a semi-honest client during Setup, $\mathcal{L}_C^{\text{Update}}$ encapsulates the leakage to a semi-honest client during Update, $\mathcal{L}_C^{\text{GenToken}}$ encapsulates the leakage to a semi-honest client during GenToken, and $\mathcal{L}_C^{\text{Search}}$ encapsulates the leakage to a semi-honest client during Search.

A dynamic MRSW SSE scheme Π is said to be secure against a semi-honest PPT adversary $\mathcal{A}$, who is allowed make $Q = \mathsf{poly}(\lambda)$ queries, with respect to $\mathcal{L}_C$, there exists a polynomial time simulator SIM, such that

$$|\Pr[\mathbf{Real}_{\mathcal{A}}^{\Pi}(\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\text{SIM}}^{\Pi}(\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$

where $\mathbf{Real}_{\mathcal{A}}^{\Pi}$ and $\mathbf{Ideal}_{\mathcal{A},\text{SIM}}^{\Pi}$ are defined in Algorithm 1 and 2 of in Appendix A, respectively.

Security against a semi-honest server. The following leakage function parameterises the security of a dynamic MRSW SSE construction against a semi-honest server.

$$\mathcal{L}_{\mathcal{S}} = \{\mathcal{L}_{\mathcal{S}}^{\text{Setup}}, \mathcal{L}_{\mathcal{S}}^{\text{Update}}, \mathcal{L}_{\mathcal{S}}^{\text{GenToken}}, \mathcal{L}_{\mathcal{S}}^{\text{Search}}\}$$

In this ensemble, $\mathcal{L}_{\mathcal{S}}^{\text{Setup}}$ encapsulates the leakage to a semi-honest server during Setup, $\mathcal{L}_{\mathcal{S}}^{\text{Update}}$ encapsulates the leakage to a semi-honest server during Update, $\mathcal{L}_{\mathcal{S}}^{\text{GenToken}}$ encapsulates the leakage to a semi-honest server during GenToken, and $\mathcal{L}_{\mathcal{S}}^{\text{Search}}$ encapsulates the leakage to a semi-honest server during Search.

A dynamic MRSW SSE scheme Π is said to be secure against a semi-honest PPT adversary $\mathcal{A}$, who is allowed make $Q = \mathsf{poly}(\lambda)$ queries, with respect to $\mathcal{L}_{\mathcal{S}}$, there exists a polynomial time simulator SIM, such that

$$|Pr[\mathbf{Real}_{\mathcal{A}}^{\Pi}(\lambda) = 1] - Pr[\mathbf{Ideal}_{\mathcal{A},\text{SIM}}^{\Pi}(\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$

where $\mathbf{Real}_{\mathcal{A}}^{\Pi}$ and $\mathbf{Ideal}_{\mathcal{A},\text{SIM}}^{\Pi}$ are defined in Algorithm 3 and 4 in Appendix A, respectively.

SSE data structures. SSE constructions heavily rely on the underlying data structures to store and efficiently search over encrypted data. We consider two widely used SSE-specific data structures in this work, namely TSet and XSet. We assume that **EDB** comprises of both TSet and XSet (as required by the construction discussed later in Section 3 and 4).

TSet. TSet is an encrypted variant 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 TSet keys 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 to support conjunctive queries (especially in the cross-tag family of constructions [9, 27, 30]). XSet stores the cross-term–specific information that is used during conjunctive query search. Note that XSet does not store any encrypted information of individual ws or ids; rather, it stores flags or bits associated with cross-terms that are generated using CRHF or PRFs. At a high level, an XSet has the following syntax,

- Insertion: XSet[index] = b, $b \in \{0, 1\}$
- Retrieval: b = XSet[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 [9]. 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 only state-of-the-art dynamic scheme with an efficient update and conjunctive query search. Unfortunately, ODXT itself does not support multi-client search. We first transform ODXT into a multi-client construction MC-ODXT (see Appendix B) following [23]. However, we show that MC-ODXT is vulnerable to cross-term 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 multi-client extension of ODXT (or MC-ODXT) following the approach in [23]. This attack demonstrates that the presence of the same cross-terms across different queries leads to severe leakage through XSet access pattern, which can completely break MC-ODXT. More precisely, we show that a semi-honest C colluding with the semi-honest $\mathcal{S}$ can exactly recover the query keywords of a legitimate C. Provided that enough query instances, the colluding C-$\mathcal{S}$ pair can recover the entire keyword dictionary. This glaring vulnerability puts MC-ODXT (or any construction following the same approach) at risk of a severe data leak. Our main construction Nomos adopts a computationally "lightweight" redundancy-based approach to reduce this leakage while incurring minimal extra storage overhead without affecting the performance.

We briefly summarise the workflow of MC-ODXT algorithms presented in Appendix B to identify the source of the leakage and subsequently discuss the attack exploiting this leakage. The MC-ODXT workflow below involves a client C obtaining a search token for a conjunctive query $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$ from the data owner $\mathcal{G}$ and querying the server $\mathcal{S}$ using those search tokens. Note that, in this MRSW setting, we use the $\mathcal{G}$ notation instead $\mathcal{D}$ as it has the additional responsibility of generating Cs 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. If there is an initial **DB** present, $\mathcal{G}$ repeatedly invokes Update on **DB** entries to update **EDB** stored on $\mathcal{S}$. The Update algorithm generates a unique TSet address addr for each (w,id) pair 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 unique addr is obtained in each update invocation). Furthermore, the update routine treats an ADD or DEL op identically, as there is no conditional execution based on ADD or DEL. $\mathcal{S}$ stores the encrypted id in TSet (which is a part of **EDB**) along with a w-specific deblinding token (α, this is required during search) received from $\mathcal{G}$. Additionally, $\mathcal{G}$ generates an xtag (or cross-tag) by combining w with id (concatenated with op) through PRF and raising to the power of g (such that during a search, xtag can be recomputed obliviously using the query tokens and α). $\mathcal{G}$ sends xtag to the server, and the server sets a bit 1 at address xtag in XSet.

Prior to interacting with $\mathcal{S}$ in the Search, a C obtains a blinded search token from $\mathcal{G}$ for a conjunctive query comprising of two components. The first one constitutes of strap (or trapdoor corresponding to the s-term) and bstags (or blinded search tags for TSet look-up) corresponding to the s-term w_1. The second component consists of bxtraps (or blinded trapdoors for x-terms) corresponding to the x-terms $\mathsf{w}_2, \ldots, \mathsf{w}_n$. C sends these tokens to $\mathcal{S}$ to look up **EDB**. In this process, $\mathcal{S}$ retrieves the encrypted ids using the de-blinded bstags. It also computes the deblinded xtags for each x-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 C. C locally checks if the id has been added for *all* ws in q, and not deleted even for one w of 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 (w, id) pair, the same xtag address needs to be generated each time the Search protocol encounters the same (w, id) pair. This association is revealed even across different queries having the 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.

Table 1: An example of SSE execution sequence.

Entity	Time	Operation	Query/Data
C_1	T1	Search	$\mathsf{w}_1 \wedge \mathsf{w}_2 \wedge \mathsf{w}_3$
$\mathcal{G}$	T2	Update	$(\mathsf{w}_3, \mathsf{id})$
C_1	T3	Search	$\mathsf{w}_1 \wedge \mathsf{w}_3$
C_2	T4	Search	$\mathsf{w}_2 \wedge \mathsf{w}_3$

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 $\mathcal{G}$, and queried again (as an x-term) at T3 by C_1 followed by C_2 at T4. Observe that these three instances of queries and update involve w_3. The second and third instances generate the same xtag for (w_3, id) pair following the MC-ODXT construction. Since $\mathcal{S}$ is assumed to be semi-honest, it can "see" that the same xtag is accessed in these instances. $\mathcal{S}$'s ability to observe these distinct accesses for xtags is the base of this attack below.

This two-phase attack assumes a semi-honest C_i that colludes with $\mathcal{S}$ in the attack process. In the build phase, 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 a semi-honest entity, it executes the search according to the protocol). Since C_i colludes with the $\mathcal{S}$, 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. C_i can repeat this process multiple times to obtain multiple (w, xtag) mappings, and $\mathcal{S}$ can grow the recorded information covering more ws.

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

3.1 Formalising MC-ODXT Attack Process

We denote a semi-honest client using C_i that colludes with a semi-honest server $\mathcal{S}$ and a benign client using C_j. We assume that the colluding client C_i shares the query keywords for which it received the query tokens tk with $\mathcal{S}$ (without shuffling) as well while executing the Search routine as specified. Upon receiving the (w, tk), $\mathcal{S}$ builds a local database XDB that stores records of the form (w, xtag). We assume that 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 build phase, where the colluding C_i engages with $\mathcal{S}$ to build the XDB. In the attack phase, $\mathcal{S}$ obtains the xtags corresponding to benign C_j's query q, and looks-up XDB to recover the ws in q. The attack accuracy (probability of exact keyword recovery) improves as more unique ws from Δ are covered in the build phase to populate XDB. Therefore, increasing the number of query iterations t in the build phase leads to higher successful keyword recovery as more ws are covered in XDB. The attack perfectly recovers all cross-terms with probability 1 for the ideal case when XDB contains all ws of Δ.

3.2 Experimental Evaluation

We executed the attack in Algorithm 1 on MC-ODXT to highlight the severity of the leakage discussed above. We used the Enron email dataset for this experiment, and the platform details for this experiment are available in Section 5. The experiment builds XDB from the recorded xtags and the associated ws. Subsequently, in the attack phase, the xtags corresponding to a benign client's queries are recorded and looked up in XDB for successful w recovery. The

Algorithm 1 Query recovery attack MC-ODXT in presence of colluding semi-honest client and semi-honest server

Input: Query tokens (tks) and ws from semi-honest C_i, query tokens of benign C_j
Output: $\mathcal{W}$: the set of cross-term ws present in C_j's query

1: **function** CrossAttack
2: **Build Phase**
Server
3: Initialise empty database XDB
Server + Colluding Client
4: **for** $l = 1$ to t **do**
Colluding Client
5: Generate a random query $q_l \xleftarrow{\$} \Delta^{*}$ ▷ *Combination of keywords from* Δ
6: Obtain search tokens tk_{q_l} for q_l from $\mathcal{G}$
7: Send q_l and tk_{q_l} (without shuffling) to $\mathcal{S}$
Server
8: Recover xtags using tk_{q_l} for $w \in q_l$ available from q_l sent by C_i
9: Set $\mathsf{XDB}[\mathsf{xtag}_j] = w_j, \forall w_j \in q_l$ received from C_i
10: **Attack Phase**
Benign Client
11: Obtain search token tk_q for $q = w_1 \wedge \ldots \wedge w_n$ from $\mathcal{G}$
12: Send tk_q to $\mathcal{S}$
Server
13: Compute the xtags from tk_q
14: Look-up XDB using the computed xtags: $w_i = \mathsf{XDB}[\mathsf{xtag}_i]$
15: Repeat this for all xtags to recover $\mathcal{W} = \{w_2, \ldots, w_n\}$
16: Return $\mathcal{W}$

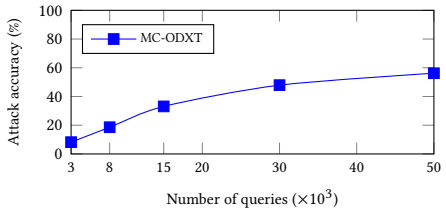

Figure 1: Attack accuracy vs number of queries by the semi-honest client in the building phase. The number of queries by the benign clients was 2000.

attack accuracy, defined as the ratio of the number of correct x-term looked up to the total number of x-terms looked up, is plotted in Figure 1 against the number of benign clients queries t in the building phase. The attack accuracy improves as the number of queries in the building phase increases, allowing the $\mathcal{S}$ to cover more keywords in XDB. In this evaluation, the CrossAttack of Algorithm 1 successfully recovered more than 50% of the x-terms of benign clients' queries.

3.3 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 needs 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.

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. An adversarial client must not be able to modify or reuse received query tokens with previously obtained (or future) query tokens[5].

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, C does not have to share the actual q with $\mathcal{G}$. Instead, C shares hashed values mapped to group elements with $\mathcal{G}$. However, in a dynamic construction following the ODXT structure, it needs to generate several bstags (corresponding to each update operation of the s-term) and additional deblindings for these tokens, which requires modifications of the approach of [23].

Our final construction Nomos adopts a similar approach to provision multi-client search, 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 exploit later as demonstrated in CrossAttack of Algorithm 1. To prevent this leakage, the XSet look-up access pattern needs to be hidden from $\mathcal{S}$. In our construction Nomos, we opt for a redundancy-based mechanism for XSet look-up that produces different access patterns for the same $(\mathsf{w}, \mathsf{id})$ pair.

Note. We would like to mention that in this work, we specifically focus on cross-term-based leakage. Since the s-terms have a lower frequency, the chance of s-terms colliding across queries from multiple clients compared to x-terms is low. Thus, this type of leakage is more severe for x-terms, and we prioritise mitigating x-term leakage in this development. We leave devising a similar strategy for s-terms as an important future work.

4 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 entity 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 l clients $\{C_1, \ldots, C_l\}$ who are allowed to obtain search tokens and search over the database. Each 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 using gate-keeper (denoted by $\mathcal{G}$) who can update **EDB** and issue search tokens to C (the gate-keeper name signifies additional responsibility to generate search tokens for Cs). $\mathcal{G}$ holds the secret key (sk) to generate the search tokens and update the database. Since Nomos is an MRSW construction, $\mathcal{G}$ is the only entity allowed to update **EDB** and is considered a trusted party. Note that, although $\mathcal{G}$ is considered a trusted entity who can "see" the data, Nomos ensures query privacy of clients by not revealing query keywords to $\mathcal{G}$.

Trust in $\mathcal{G}$ We stress here that although $\mathcal{G}$ is a trusted entity, it does not learn the client's search result, along with not knowing the query keywords. Thus, $\mathcal{G}$ is trusted for updates only. This trust level in the MRSW setting is aptly applicable to real-life examples, where $\mathcal{G}$ can be considered an administrative entity, such as the government or an employer, who can update the database and enforce search policies on the clients accessing the data.

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

We elucidate the MRSW setting here for better clarity and understanding. Consider a genome analysis service provider offering a cloud-based gnome referencing service for clients who can query over standard genome data provided by the analysis service. Following the current cloud-based service trend, assume that the genome analysis service has outsourced the infrastructure support to a third-party cloud storage and computing resource provider, such as Amazon AWS. In this example, the genome service provider can be modelled as $\mathcal{G}$, the cloud server acts as $\mathcal{S}$, and the clients requesting for look-up can be considered as Cs. Naturally, genome data is considered sensitive private information, which needs to be stored on $\mathcal{S}$ in encrypted form, and a C needs to query the data for reference information. Such applications can be aptly handled by a scalable MRSW SSE scheme like Nomos.

Nomos setting assumes $\mathcal{S}$ as a semi-honest party and treats each client as a semi-honest party individually (who can potentially collude with $\mathcal{S}$ while following the Nomos description as specified). $\mathcal{G}$ is a trusted entity that can update the encrypted database and generate search tokens for C's queries (as $\mathcal{G}$ holds the secret keys). The multi-client provisioning in Nomos follows the approach [23] and incorporates modifications for dynamic updates. Note that a de-centralised trust in $\mathcal{G}$ is ideal for the MRMW setting, where multiple parties update the encrypted database. In contrast, the MRSW setting allows only one party to update the database - typically the data-owner. Thus, the MRSW setting allows provisioning multi-client search using basic OPRF primitive rather than relying

[5]Note that, token forgery implies the presence of a malicious client; whereas the attack in Algorithm 1 requires only a colluding semi-honest client. Nonetheless, our main Nomos construction considers token forgery prevention as a necessary feature in the multi-client setting.

Algorithm 2 NOMOS SETUP

Input: Security parameters λ
Output: sk, UpdateCnt, and **EDB**

function NOMOS.SETUP
- Gate-keeper
- Sample a uniformly random ley 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 uniformly random key K_M from $\{0,1\}^\lambda$ for AE
- Initialise UpdateCnt, TSet, XSet to empty maps
- Gate-keeper keeps 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
- Set **EDB** = (TSet, XSet)
- Send **EDB** to server

Algorithm 3 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 is the number of hash functions for insertion into RBF, (w, id) pair to be updated, update operation op
Output: Updated **EDB**

function NOMOS.UPDATE
- Gate-keeper
- Parse sk = (K_T, K_X, K_Y) and UpdateCnt
- Set $K_Z \leftarrow F((H(\mathsf{w}))^{K_S}, 1)$
- If UpdateCnt[w] is NULL then set UpdateCnt[w] = 0
- Set UpdateCnt[w] = UpdateCnt[w] + 1
- Set addr = $(H(\mathsf{w}||\text{UpdateCnt}[\mathsf{w}]||0))^{K_T[I(\mathsf{w})]}$
- Set val = $(\mathsf{id}||\mathsf{op}) \oplus (H(\mathsf{w}||\text{UpdateCnt}[\mathsf{w}]||1))^{K_T[I(\mathsf{w})]}$
- Set $\alpha = F_p(K_Y, \mathsf{id}||\mathsf{op}) \cdot (F_p(K_Z, \mathsf{w}||\text{UpdateCnt}[\mathsf{w}])^{-1})$
- Set $\text{xtag}_i = H(\mathsf{w})^{K_X[I(\mathsf{w})] \cdot F_p(K_Y, \mathsf{id}||\mathsf{op}) \cdot i}$, where $i \in [\ell]$
- Send $(\text{addr}, \text{val}, \alpha, \{\text{xtag}_i\}_{i \in [\ell]})$ to server

Server
- Parse **EDB** = (TSet, XSet)
- Set TSet[addr] = (val, α)
- Set XSet[xtag_i] = 1, for $i \in [\ell]$

on independent system-oriented approaches like trusted execution environments or advanced primitives like MPC, as it handles updates only from the data-owner, unlike the MRMW setting. This model is adopted in existing works like OSPIR-OXT [23], which we have followed here.

4.1 Setup and Update

The NOMOS SETUP routine of Algorithm 2 is executed by $\mathcal{G}$ that initialises the system (including **EDB** on $\mathcal{S}$). Subsequently, $\mathcal{G}$ and $\mathcal{S}$ can jointly execute NOMOS UPDATE of Algorithm 3 repeatedly on the data records from **DB**.

Update process. The UPDATE algorithm is invoked by $\mathcal{G}$ with (w, id) and op as input. $\mathcal{S}$ receives the encrypted values, along with tags generated by $\mathcal{G}$ and updates the TSet and XSet. The UPDATE process of Algorithm 3 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. Note that, Algorithm 3 incorporates the RBF's (discussed in Section 4.2) redundant xtag generation process. The changes in NOMOS algorithm(s) from MC-ODXT are highlighted in red.

4.2 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 (w, id) combinations from different queries (and updates). Intuitively, to mitigate this leakage, these memory accesses (to the same address for a particular (w, 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 multiple "copies" of the XSet bit value are stored at multiple addresses. A random subset of these locations is looked up in each subsequent access for the same (w, id) pair.

Randomising XSet access. We opt for a Bloom filter (BF) (which physically stores XSet) 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 look-up. However, straightforwardly plugging in BF into MC-ODXT does not hide the repeated access pattern as k addresses for a particular (w, 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 for look-up, 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, $\mathcal{S}$ receives a different set of BF addresses for each repeated access of a particular (w, id) pair and hence can not correlate among previously accessed entries. We call this BF variant 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 need to be generated from xtags. 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.

$$\mathcal{G}(\text{UPDATE}) \quad : \text{xtag}_i = H(\mathsf{w})^{K_X[I(\mathsf{w})] \cdot F_p(K_Y, \mathsf{id}||\mathsf{op}) \cdot i},\ i \in [\ell]$$

$$\mathcal{G}(\text{GENTOKEN}) \quad : \overline{\text{bxtrap}'}_j = \overline{\text{bxtrap}'}_j \cup \{(\text{bxtrap}'_j)^{\beta_i}\}$$

The revised final UPDATE and GENTOKEN algorithms are presented in Algorithm 3 and 4, respectively. The SEARCH routine is modified to compute final k addresses for RBF and is presented in Algorithm 5 in Section 4.3.

Since UPDATE protocol should be 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) pair records into RBF-based XSet. The generated XSet addresses (for all (w,id) pairs) must be shuffled by $\mathcal{G}$ prior to sending to $\mathcal{S}$ in batches. This random shuffling is

necessary to prevent $\mathcal{S}$ from associating any XSet access pattern to a probable keyword. In contrast, such shuffling in MC-ODXT is ineffective against $\mathcal{S}$ associating a probable x-term to an observed (or a set of observed) XSet address as only one single xtag match is required for a particular (w, id) pair update and search.

4.3 Token Generation and Search

The multi-client search process starts with the token generation process outlined in Algorithm 4 - a two-party protocol between a C and $\mathcal{G}$. We briefly summarise the workflow of the GenToken method of Algorithm 4 that generates the search tokens while maintaining the query privacy of legitimate clients and preventing token forgery by an adversarial client.

Token generation phase. The Nomos Search algorithm follows the ODXT [30] Search process, which generates two types of search tokens – the stags and the xtraps. The stags are generated from the s-term (the least-frequently updated term or w_1 in Algorithm 4, without loss of generality), which are used to retrieve encrypted ids through TSet look-up. Whereas xtraps are generated from x-terms which are used to check the validity of a (w, 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, 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 C_i to send query ws in blinded form. The major difference from OSPIR-OXT [23] is that ODXT generates an stag for each update count for the s-term, whereas OSPIR-OXT generates a single stag (strap in GenToken). This is a direct consequence of the dynamic update capability of ODXT, and GenToken routine computes the blinded exponentiations for each stag. Similarly, 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 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 its own part of the OPRF computation (party B's computation in OPRF as discussed in Section 2) by processing bstrap, bstag and bxtrap. The blinded strap (bstrap′) computation 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 potentially malicious client from modifying the search tokens 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 the Search routine).

In the final phase of GenToken routine, 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 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 $\mathcal{G}$ at Setup and shared with $\mathcal{S}$.

Algorithm 4 Nomos GenToken

Input: $q = \{\mathsf{w}_1, \cdots, \mathsf{w}_n\}$ (n keywords in q), $\mathcal{P}$ is the set of allowed attribute sequences, ℓ and k hashes are used for insertion and query in RBF, respectively

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 $m = \mathsf{UpdateCnt}[\mathsf{w}_1]$ ▷ *the update count of the least frequently updated w*

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

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

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

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

Set $c_j = (H(\mathsf{w}_1||j||1))^{s_j}$, 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 $\mathsf{strap}' = (a_1)^{K_S}$

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

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

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

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

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

  $\overline{\mathsf{bxtrap}'}_j = \{\}$

  $\overline{\mathsf{bxtrap}'}_j = \overline{\mathsf{bxtrap}'}_j \cup \{(\mathsf{bxtrap}'_j)^{\beta_i}\}$, for $\beta_i \in \{\beta_1, \ldots, \beta_k\}$

Set $\mathsf{env} = \mathsf{AE.Enc}_{K_M}(\rho_1, \cdots, \rho_n, \gamma_1, \cdots, \gamma_m)$

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

Client

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

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

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

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

  Initialise $\overline{\mathsf{bxtrap}}_j = \{\}$

  for $l = 1$ to k **do**

    $\overline{\mathsf{bxtrap}}_j = \overline{\mathsf{bxtrap}}_j \cup \{(\overline{\mathsf{bxtrap}'[j][l]})^{r_j^{-1}}\}$

Output $(\mathsf{strap}, \mathsf{bstag}_1, \cdots, \mathsf{bstag}_m, \delta_1, \cdots, \delta_m, \overline{\mathsf{bxtrap}}_2, \cdots, \overline{\mathsf{bxtrap}}_n, \mathsf{env})$ as search token

Search phase. The Nomos Search of Algorithm 5 is jointly executed by a C and $\mathcal{S}$ without any involvement of $\mathcal{G}$. However, 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). At a high level, the Search protocol proceeds in two stages - first, C computes the final xtraps from the received bxtraps. Note that the resulting xtokens are still blinded as C does not have $\mathcal{G}$'s blinding

Algorithm 5 NOMOS SEARCH

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

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

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[6].

[6]We emphasise that NOMOS follows the vast majority of the SSE literature (including ODXT itself) in its index-only focus and does not incorporate a dedicated mechanism to handle actual document retrieval. We leave extending NOMOS to full SSE scheme with final encrypted document retrieval as an interesting future work.

S-term information to clients. In this multi-client setting we assume that C_i's obtain the s-term frequency information from $\mathcal{G}$ via a suitable privacy-preserving mechanism without revealing the keyword to $\mathcal{G}$. This is a reasonable assumption following from the prior works including OXT [9], OSPIR-OXT [23], HXT [27], and ODXT [30] (as the static constructions do not have update capability, they use keyword frequency instead of the update frequency). We elaborate more on possible privacy-preserving mechanisms for frequency information retrieval (such as based on private information retrieval [2, 13, 29]) in Appendix B.1.

4.4 Computation and Storage Overhead

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

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

The SEARCH protocol computes $k \cdot |q| \cdot |\text{UpdateCnt}[\text{w}_1]|$ group operations to compute the blinded xtokens. $\mathcal{S}$ performs $|\text{UpdateCnt}[\text{w}_1]|$ TSet look-ups that require $|\text{UpdateCnt}[\text{w}_1]|$ group operations for deblinding. Additionally, $\mathcal{S}$ computes a total $k \cdot |q| \cdot |\text{UpdateCnt}[\text{w}_1]|$ XSet addresses for look-up. Therefore, NOMOS SEARCH incurs $O(k \cdot |q| \cdot |\text{UpdateCnt}[\text{w}_1]|)$ 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, C needs to send $O(k \cdot |q| \cdot |\text{UpdateCnt}[\text{w}_1]|)$ tokens to $\mathcal{S}$, and it receives $O(|\text{UpdateCnt}[\text{w}_1]|)$ encrypted values back as the result. Hence, the asymptotic communication overhead of NOMOS SEARCH routine is $O(k \cdot |q| \cdot |\text{UpdateCnt}[\text{w}_1]|)$.

Storage overhead. We analyse the NOMOS **EDB** storage overhead with respect to the plain database **DB**. The storage overhead for **EDB** in NOMOS is essentially the combined 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

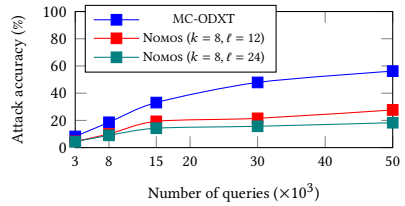

Figure 2: Attack accuracy vs number of queries t by the colluding client in the build phase. The number of queries by the benign clients was set to 2000.

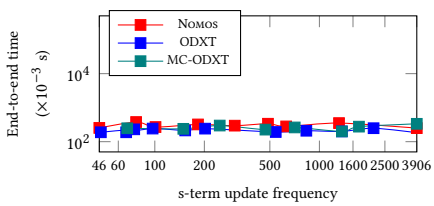

Figure 3: End-to-end query latency for two-keyword queries of the form $q = \mathsf{w}_a \wedge \mathsf{w}_v$ (s-term w_a update frequency is fixed at 70).

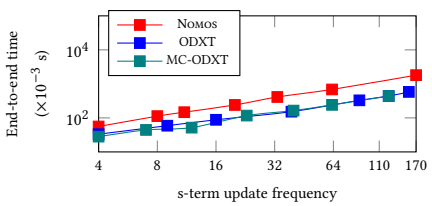

Figure 4: End-to-end query latency for two keyword queries of the form $q = \mathsf{w}_v \wedge \mathsf{w}_a$ (s-term w_v update frequency is varied).

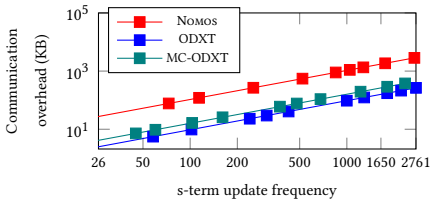

Figure 5: End-to-end communication overhead for two-keyword queries of the form $q = \mathsf{w}_v \wedge \mathsf{w}_a$ (s-term w_v update frequency is varied).

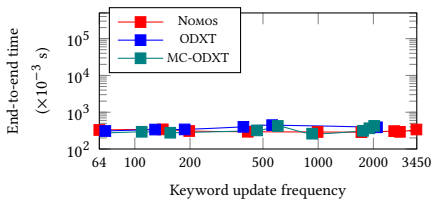

Figure 6: End-to-end query time for multi-keyword queries of the form $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$ (s-term w_1 update frequency is fixed at 60).

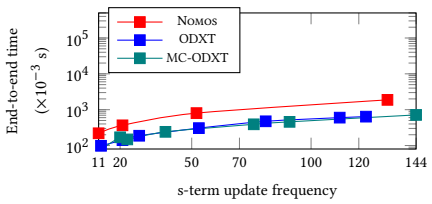

Figure 7: End-to-end query time for multi-keyword queries of the form $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$ (s-term w_1 update frequency is varied).

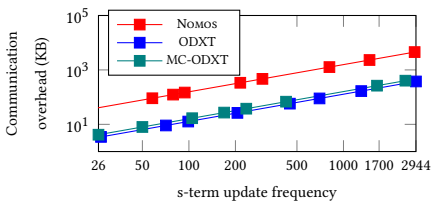

Figure 8: End-to-end communication overhead for multi-keyword queries of the form $q = \mathsf{w}_1 \wedge \ldots \wedge \mathsf{w}_n$ (s-term w_1 update frequency is varied).

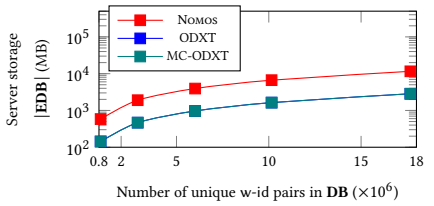

Figure 9: Server-side storage overhead (EDB size) for Nomos, MC-ODXT and ODXT for same plain input database size.

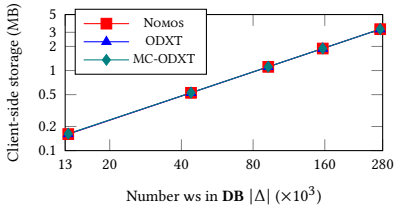

Figure 10: Client-side storage overhead for Nomos, ODXT and MC-ODXT for same plain input database size.

that stores encryptions of $O(|\mathbf{DB}|)$ items. As a result, Nomos has linear $O(|\mathbf{DB}|)$ asymptotic storage overhead in practice.

Security analysis. We divert the elaborate security analysis of Nomos to the full version of the paper[1] due to lack of space.

5 IMPLEMENTATION DETAILS AND RESULTS

In this section, we describe a prototype implementation of Nomos and evaluate its overhead and performance over real-world databases.

Data set and platform. We used the Enron email data set[7] for our experiments. The database contains 517,401 documents (emails) and 20 million keyword-document pairs, with a total size 1.9 GB. The final Nomos algorithm was implemented using C++11 with native multithreading support and was compiled using GCC9. We used Redis as the database backend system to store **EDB** comprising TSet and XSet. We ran the experiments on dual 24-core Intel Xeon E5-2690 2.6GHz CPU with 128GB RAM and 512GB SSD running Ubuntu 20.04 64bit operating system with gigabit network link.

Query processing. We evaluated Nomos's performance for two different types of queries - 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. In another set, the frequency of w_1 is kept fixed and max frequency of $\{\mathsf{w}_2, \ldots, \mathsf{w}_n\}$ is varied. These experiments examine the sublinear search and communication overhead of Nomos for two and multi-keyword queries. The query keywords are sampled randomly from Δ.

[7] https://www.cs.cmu.edu/~enron

5.1 Experiments on Leakage

We executed the CrossAttack method of Algorithm 1 on Nomos to compare the leakage with MC-ODXT. The attack accuracy is plotted in Figure 2 against the number of queries issued by the colluding client in the build phase of the attack. Clearly, Nomos has a significantly lower attack accuracy compared to MC-ODXT. This low attack accuracy is a direct consequence of the redundancy of RBF during XSet accesses. The value of the RBF parameter ℓ was set to 12, while the parameter k was set to 8. Thus, Nomos achieves more than 50% reduction in attack accuracy with four additional indices for XSet insertion. The parameter k needs to be kept at a fixed value to maintain a desired false positive rate for the same database parameters and experiment setting as of the non-RBF version. In contrast, ℓ is responsible for the redundancy in RBF leading to a lower attack accuracy due to the increased redundant accesses per $(\mathsf{w}, \mathsf{id})$ pair. However, a high value of ℓ would incur larger storage to accommodate additional XSet entries, and thus implies a trade-off. We view this as a leakage-versus-storage trade-off in the multi-client setting that is unavoidable in practice. We present comprehensive discussion on the choice of parameters and analysis of RBF in Appendix C.

5.2 Experiments on the Search Latency

We considered two types of queries to evaluate the search performance of Nomos as stated earlier in this section. For two-keyword queries, we fix the frequency of the constant term w_a and vary the frequency of the variable term w_v 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 3 and queries of the form $q = \mathsf{w}_v \wedge \mathsf{w}_a$ in Figure 4. Observe that, in Figure 3, the end-to-end search latency remains almost constant. Whereas in Figure 4, the search latency varies linearly with the frequency of the variable term. This behaviour validates the sub-linearity of the Nomos search algorithm where the search latency linearly depends on the frequency of the s-term of $|\mathbf{DB}(\mathsf{w}_1)|$, which is sublinear in terms of total number of records in the database.

The communication overheads of Nomos GenToken and Search are plotted in Figure 5 for two-keyword queries. Nomos incurs sub-linear communication overhead (linear in terms of s-term update frequency) for GenToken and Search as shown in the plot. However, in comparison with ODXT, Nomos has the additional *necessary* overhead of search tokens generated by GenToken, and the Search communication overhead increases 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 $n \in [3, 6]$, and we denote w_1 as the s-term in the multi-keyword queries. The end-to-end search latency for two sets of experiments is plotted for fixed and variable s-term update frequencies in Figure 6 and Figure 7, respectively. Observe that the search overhead remains sublinear (proportional to the frequency of the s-term) in terms of the number of total records in **EDB**. Similarly, the communication overhead remains sublinear in the total database size scaled by the number of cross-terms, as illustrated in Figure 8.

5.3 Evaluation of the Storage Overhead

We varied the number of ws in **DB** and generated the corresponding **EDB** by executing Nomos Update. We compare the Nomos storage overhead with ODXT to illustrate the storage overhead as a trade-off with lesser leakage. The **EDB** overhead for both Nomos and ODXT are plotted in Figure 9, and the client-side storage overhead is plotted in Figure 10. Observe that the storage overhead profile for both Nomos and ODXT remains linear with **DB** size, and Nomos **EDB** overhead is approximately 2.5 times of ODXT which is manageable on the cloud in practice. All three constructions require $O(|\Delta|)$ client-side storage as illustrated in Figure 10.

The performance overhead of Nomos is slightly higher compared to MC-ODXT as Nomos generates more tokens due to RBF-based XSet look-up. The increased performance and storage overhead of Nomos is a necessary leakage-versus-overhead trade-off to allow secure searches in the multi-client setting of Nomos. The storage overhead of Nomos is less than 2×-2.5× of ODXT (and MC-ODXT) for storage that is practically manageable in a cloud infrastructure. Compared with the plain version of search, encrypted search incurs higher overhead which varies depending upon the cryptographic parameters/algorithms used, data structure, and the database system used. However, it provides strong confidentiality against unauthorised access compared to the typical unencrypted search process and incurs less leakage and communication overhead than the naïve symmetrically encrypted database with trivial decryption.

In summary, Nomos provides secure multi-client search with dynamic updates at the expense of minimal additional storage overhead. Modern cloud services can easily manage this additional storage to provide a secure environment to process encrypted data which multiple users can access. Naturally, depending upon the requirement, a service provider needs to make necessary modifications to the core algorithms. These modifications mainly include parsing the data into a suitable format to store and process as a multi-map. The data owner is responsible for this pre-processing of unstructured data to be updated into the encrypted database on the remote server. Thus, in general, Nomos can be adopted in a broad set of practical applications with minimal modification, such as healthcare, government records, and banks, to name a few.

6 CONCLUSION AND FUTURE DIRECTIONS

We introduced the first forward and backward secure dynamic multi-client SSE scheme Nomos supporting conjunctive Boolean queries. Nomos is an MRSW construction that builds upon the state-of-the-art SRSW dynamic construction ODXT [30]. We showed that the straight-forward extension of ODXT to the multi-client setting is completely insecure against collusion between a semi-honest client and a semi-honest server due to a cross-term–based leakage. 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 presented extensive experimentation to demonstrate the practical performance of Nomos over real-world databases. We leave extending Nomos to the MRMW setting as an interesting direction of future research.

ACKNOWLEDGMENTS

The authors would like to thank the grant "Design and Implementation of Efficient and Secure Searchable Encryption" sponsored by MHRD-STARS (Scheme for Transformational and Advanced Research in Sciences), India for partially supporting the work.

REFERENCES

[1] Megumi Ando and Marilyn George. 2022. On the Cost of Suppressing Volume for Encrypted Multi-maps. *Proceedings on Privacy Enhancing Technologies* 4 (2022), 44–65.

[2] Amos Beimel and Yuval Ishai. 2001. Information-theoretic private information retrieval: A unified construction. In *ICALP*.

[3] Mihir Bellare and Chanathip Namprempre. 2000. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT*.

[4] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. 2005. Hierarchical Identity Based Encryption with Constant Size Ciphertext. In *EUROCRYPT*.

[5] Christoph Bösch, Pieter H. Hartel, Willem Jonker, and Andreas Peter. 2014. A Survey of Provably Secure Searchable Encryption. *ACM Comput. Surv.* 47 (2014).

[6] Raphael Bost. 2016. $\Sigma o \varphi o \varsigma$: Forward secure searchable encryption. In *CCS*.

[7] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*.

[8] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS*.

[9] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO*.

[10] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New Constructions for Forward and Backward Private Symmetric Searchable Encryption. In *CCS*.

[11] Yan-Cheng Chang and Michael Mitzenmacher. 2005. Privacy preserving keyword searches on remote encrypted data. In *ACNS*.

[12] Melissa Chase and Seny Kamara. 2010. Structured Encryption and Controlled Disclosure. In *ASIACRYPT*.

[13] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private information retrieval. In *FOCS*.

[14] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS*.

[15] Mohammad Etemad, Alptekin Küpçü, Charalampos Papamanthou, and David Evans. 2018. Efficient Dynamic Searchable Encryption with Forward Privacy. *PoPETS* (2018).

[16] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel-Catalin Rosu, and Michael Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS*.

[17] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *ACM STOC*.

[18] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *CRYPTO*.

[19] Eu-Jin Goh. 2003. Secure Indexes. *IACR Cryptology ePrint Archive* (2003).

[20] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*.

[21] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43 (1996).

[22] Ariel Hamlin, Abhi Shelat, Mor Weiss, and Daniel Wichs. 2018. Multi-Key Searchable Encryption, Revisited. In *Public-Key Cryptography - PKC 2018 (Lecture Notes in Computer Science)*. Springer.

[23] Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2013. Outsourced symmetric private information retrieval. In *CCS*.

[24] Seny Kamara and Tarik Moataz. 2017. Boolean Searchable Symmetric Encryption with Worst-Case Sub-linear Complexity. In *EUROCRYPT*.

[25] Seny Kamara and Charalampos Papamanthou. 2013. Parallel and dynamic searchable symmetric encryption. In *FC*.

[26] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *ACM CCS*.

[27] Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K. Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shifeng Sun, Dongxi Liu, and Cong Zuo. 2018. Result Pattern Hiding Searchable Encryption for Conjunctive Queries. In *CCS*.

[28] Moni Naor and Omer Reingold. 2004. Number-theoretic constructions of efficient pseudo-random functions. *JACM* 51 (2004).

[29] Rafail Ostrovsky and William E Skeith III. 2007. A survey of single-database private information retrieval: Techniques and applications. In *PKC*.

[30] Sikhar Patranabis and Debdeep Mukhopadhyay. 2021. Forward and Backward Private Conjunctive Searchable Symmetric Encryption. In *NDSS*.

[31] Phillip Rogaway. 2002. Authenticated-encryption with associated-data. In *CCS*.

[32] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *IEEE S&P*.

[33] Wolfgang Stadje. 1990. The collector's problem with group drawings. *Advances in Applied Probability* 22, 4 (1990), 866–882.

[34] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX Security Symposium*.

A SECURITY GAMES FOR DYNAMIC MULTI-CLIENT SSE

We present the **Real** and **Ideal** security games for dynamic multi-client SSE as stated in Section 2.3 of the main manuscript.

Games for adversarial client The **Real** and **Ideal** games for the security of a dynamic MRSW SSE scheme Π from an adversarial client are presented in Algorithms 6 and 7.

Algorithm 6 Experiment $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ adversarial client

function $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$
  $N \leftarrow \mathcal{A}(\lambda)$
  $(\mathsf{sk}, \mathsf{st}_0, \mathbf{EDB}_0) \leftarrow \Pi.\textsc{Setup}(\lambda, N)$
  for $k \leftarrow 1$ to Q **do**
    if query-type == gentoken **then**
      $q_k \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
      $(\mathsf{st}_k, \mathsf{t}_k) \leftarrow \textsc{GenToken}(\mathsf{sk}, \mathsf{st}_{k-1}, q_k)$
    else if query-type == search **then**
      $t_k \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
      $(\mathsf{st}_k, \mathbf{EDB}_k, \mathbf{DB}(q_k)) \leftarrow \textsc{Search}(\mathsf{sk}, \mathsf{st}_{k-1}, \mathsf{t}_k; \mathbf{EDB}_{k-1})$
    τ_k denotes the view of the adversary after the k^{th} query
  $b \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_Q, \tau_1, \ldots, \tau_Q)$
  return b

Algorithm 7 Experiment $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathsf{SIM}}(\lambda)$ for adversarial client

function $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathsf{SIM}}(\lambda)$
  Parse the $\mathcal{L}$ as: $\{\mathcal{L}^{\textsc{Setup}}, \mathcal{L}^{\textsc{Update}}, \mathcal{L}^{\textsc{GenToken}}, \mathcal{L}^{\textsc{Search}}\}$.
  $N \leftarrow \mathcal{A}(\lambda)$
  $(\mathsf{st}_{\mathsf{SIM}}, \mathbf{EDB}_0) \leftarrow \mathsf{SIM}_0(\mathcal{L}^{\textsc{Setup}}(\lambda, N))$
  for $k \leftarrow 1$ to Q **do**
    if query-type == gentoken **then**
      $q_k \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
      $(\mathsf{st}_{\mathsf{SIM}}, \mathsf{t}_k) \leftarrow \mathsf{SIM}_1(\mathsf{st}_{\mathsf{SIM}}, \mathcal{L}^{\textsc{GenToken}}(q_k))$
    else if query-type == search **then**
      $t_k \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
      $(\mathsf{st}_{\mathsf{SIM}}, \mathbf{EDB}_k, \mathbf{DB}(q_k)) \leftarrow \mathsf{SIM}_{\textsc{Search}}(\mathsf{st}_{\mathsf{SIM}}, \mathcal{L}^{\textsc{Search}}(\mathsf{t}_k); \mathbf{EDB}_{k-1})$
    τ_k denote the view of the adversary after the k^{th} query
  $b \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_Q, \tau_1, \ldots, \tau_Q)$
  return b

Games for adversarial server The **Real** and **Ideal** games for the security of a dynamic MRSW SSE scheme Π from an adversarial server are presented in Algorithms 8 and 9.

Algorithm 8 Experiment $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ adversarial server

function $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$
  $N \leftarrow \mathcal{A}(\lambda)$
  $(\mathsf{sk}, \mathsf{st}_0, \mathbf{EDB}_0) \leftarrow \Pi.\textsc{Setup}(\lambda, N)$
  for $k \leftarrow 1$ to Q **do**
    if query-type == search **then**
      $t_k \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
      $(\mathsf{st}_k, \mathbf{EDB}_k, \mathbf{DB}(q_k)) \leftarrow \Pi.\textsc{Search}(\mathsf{sk}, \mathsf{st}_{k-1}, t_k; \mathbf{EDB}_{k-1})$
    else if query-type == update **then**
      $(\mathsf{op}_k, \{\mathsf{w}_k, \mathsf{id}_k\}) \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
      $(\mathsf{st}_k, \mathbf{EDB}_k) \leftarrow$
      $\Pi.\textsc{Update}(\mathsf{sk}, \mathsf{st}_{k-1}, (\mathsf{op}_k, \{\mathsf{w}_k, \mathsf{id}_k\}); \mathbf{EDB}_{k-1})$
    τ_k denotes the view of the adversary after the k^{th} query
  $b \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_Q, \tau_1, \ldots, \tau_Q)$
  return b

Algorithm 9 Experiment $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}(\lambda)$ for adversarial server

function $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathrm{SIM}}(\lambda)$
  Parse $\mathcal{L}_{\mathcal{S}}$ as: $\{left(\mathcal{L}^{\textsc{Setup}}, \mathcal{L}^{\textsc{Update}}, \mathcal{L}^{\textsc{GenToken}}, \mathcal{L}^{\textsc{Search}}\}$.
  $N \leftarrow \mathcal{A}(\lambda)$
  $(\mathrm{st}_{\mathrm{SIM}}, \mathbf{EDB}_0) \leftarrow \mathrm{SIM}_0(\mathcal{L}^{\textsc{Setup}}(\lambda, N))$
  for $k \leftarrow 1$ to Q **do**
    if query-type == search **then**
      $t_k \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
      $(\mathrm{st}_{\mathrm{SIM}}, \mathbf{EDB}_k, \mathbf{DB}(q_k)) \leftarrow \mathrm{SIM}_2(\mathrm{st}_{\mathrm{SIM}}, \mathcal{L}^{\textsc{Search}}(t_k); \mathbf{EDB}_{k-1})$
    else if query-type == update **then**
      $(\mathrm{op}_k, \{\mathrm{w}_k, \mathrm{id}_k\}) \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_{k-1}, \tau_1, \ldots, \tau_{k-1})$
      $(\mathrm{st}_k, \mathbf{EDB}_k) \leftarrow$
      $\mathrm{SIM}_1(\mathrm{st}_{\mathrm{SIM}}, \mathcal{L}^{\textsc{Update}}((\mathrm{op}_k, \{\mathrm{w}_k, \mathrm{id}_k\})); \mathbf{EDB}_{k-1})$
    τ_k denote the view of the adversary after the k^{th} query
  $b \leftarrow \mathcal{A}(\lambda, \mathbf{EDB}_Q, \tau_1, \ldots, \tau_Q)$
  return b

B 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 [23]. The MC-ODXT Setup algorithm (Algorithm 10) executed by $\mathcal{G}$ samples OPRF, PRF and AE keys. A client $\mathcal{C}$ engages with $\mathcal{G}$ in the GenToken protocol (Algorithm 12) to obtain the blinded search tokens for a conjunctive query to search over the **EDB**. After obtaining the search tokens, $\mathcal{C}$ engages with $\mathcal{S}$ in the Search protocol (Algorithm 13) to retrieve the set of ids matching the conjunctive query. $\mathcal{G}$ jointly executes the Update protocol (Algorithm 11) with $\mathcal{S}$ to insert contents into or delete contents from **EDB**.

Algorithm 10 MC-ODXT Setup

Input: λ
Output: sk, UpdateCnt, **EDB**
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 UpdateCnt, TSet, XSet to empty maps
  Gate-keeper keeps 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
  Set **EDB** = (TSet, XSet)
  Send **EDB** to server

B.1 Private retrieval of s-term information

As stated in Section 4.3 of the main body, the s-term update frequency requires a commonly accessible record in this setting. This necessitates a secure mechanism to disclose the keyword frequencies to prevent $\mathcal{G}$ from knowing $\mathcal{C}$'s query keywords. Naïvely, this can be achieved via a publicly hosted website or storage with $\mathcal{G}$ having the write permission (implicitly adopted by OXT [9], OSPIR-OXT [23], HXT [27], and ODXT [30]). In this work we assumed that $\mathcal{C}$'s have some suitable mechanism to retrieve the frequency information from $\mathcal{G}$, and focused on the core construction to allow multi-client search capability.

A plausible way to allow such frequency information retrieval in a privacy-preserving manner is to employ a private information retrieval (PIR) [2, 13, 29] based mechanism involving a $\mathcal{C}$ and the $\mathcal{G}$. The idea is (at a high level) that the $\mathcal{G}$ holds an array of keyword frequency values, indexed by keywords, and the $\mathcal{C}$ wants to look up the frequency of a particular keyword without revealing to the $\mathcal{G}$ what is the keyword being looked up. This lends itself to a classic single-server PIR-based solution. In fact, one could also use a multi-server PIR solution [2] where the $\mathcal{G}$ simply distributes the frequency information across multiple (mutually distrusting) servers, thus decentralising the trust assumptions involved.

We note, however, that using a (multi-server) PIR-based frequency information retrieval in the search token generation phase brings in additional computation overhead, and possibly, additional rounds. We leave incorporating a dedicated s-term update frequency retrieval mechanism as a part of the MRMW extension of Nomos.

Algorithm 11 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 $K_T[I(\mathrm{w})]$ and $K_X[I(\mathrm{w})]$ for attribute $I(\mathrm{w})$ 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 UpdateCnt
  Set $K_Z \leftarrow F((H(\mathrm{w}))^{K_S}, 1)$
  If UpdateCnt[w] is NULL then set UpdateCnt[w] = 0
  Set UpdateCnt[w] = UpdateCnt[w] + 1
  Set addr = $(H(\mathrm{w}||\mathrm{UpdateCnt}[\mathrm{w}]||0))^{K_T[I(\mathrm{w})]}$
  Set val = $(\mathrm{id}||\mathrm{op}) \oplus (H(\mathrm{w}||\mathrm{UpdateCnt}[\mathrm{w}]||1))^{K_T[I(\mathrm{w})]}$
  Set $\alpha = F_p(K_Y, \mathrm{id}||\mathrm{op}) \cdot (F_p(K_Z, \mathrm{w}||\mathrm{UpdateCnt}[\mathrm{w}]))^{-1})$
  Set xtag = $H(\mathrm{w})^{K_X[I(\mathrm{w})] \cdot F_p(K_Y, \mathrm{id}||\mathrm{op})}$
  Send (addr, val, α, xtag to server
  Server
  Parse **EDB** = (TSet, XSet)
  Set TSet[addr] = (val, α)
  Set XSet[xtag] = 1

Algorithm 12 MC-ODXT GenToken

Input: $q = \mathrm{w}_1 \wedge \ldots \wedge \mathrm{w}_n$. $\mathcal{P}$ is the set of allowable attribute sequences, K_S, K_T, K_X, K_M
Output: strap, $\mathrm{bstag}_1, \cdots, \mathrm{bstag}_m, \delta_1, \cdots, \delta_m, \mathrm{bxtrap}_2, \cdots, \mathrm{bxtrap}_n$, env
function MC-ODXT.GenToken
  Client
  Pick $r_1, \cdots, r_n \xleftarrow{\$} \mathbb{Z}_p^*$
  Set m = UpdateCnt[w_1]
  Pick $s_1, \cdots, s_m \xleftarrow{\$} \mathbb{Z}_p^*$
  Set $a_j \leftarrow (H(\mathrm{w}_j))^{r_j}$, for $j = 1, \cdots, n$
  Set $b_j \leftarrow (H(\mathrm{w}_1||j||0))^{s_j}$, for $j = 1, \cdots, m$
  Set $c_j \leftarrow (H(\mathrm{w}_1||j||1))^{s_j}$, for $j = 1, \cdots, m$
  Set av = $(I(\mathrm{w}_1), \cdots, I(\mathrm{w}_n)) = (I_1, \ldots, I_n)$
  Gate-keeper
  Abort if 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 strap$' \leftarrow (a_1)^{K_S}$
  Set $\mathrm{bstag}'_j \leftarrow (b_j)^{K_T[I_1] \cdot \gamma_j}$, for $j = 1, \cdots, m$
  Set $\delta'_j \leftarrow (c_j)^{K_T[I_1]}$, for $j = 1, \cdots, m$
  Set $\mathrm{bxtrap}'_j \leftarrow (a_j)^{K_X[I_j] \cdot \rho_j}$ for $j = 2, \cdots, n$
  Set env = $\mathrm{AE.Enc}_{K_M}(\rho_1, \cdots, \rho_n, \gamma_1, \cdots, \gamma_m)$
  Send strap$'$, $\mathrm{bstag}'_1, \cdots, \mathrm{bstag}'_m, \delta'_1, \cdots, \delta'_m, \mathrm{bxtrap}'_2, \cdots, \mathrm{bxtrap}'_n$, env to Client
  Client
  Set strap $\leftarrow (\mathrm{strap}')^{r_1^{-1}}$
  Set $\mathrm{bstag}_j \leftarrow (\mathrm{bstag}'_j)^{s_j^{-1}}$, for $j = 1, \cdots, m$
  Set $\delta_j \leftarrow (\delta'_j)^{s_j^{-1}}$, for $j = 1, \cdots, m$
  Set $\mathrm{bxtrap}_j \leftarrow (\mathrm{bxtrap}'_j)^{r_j^{-1}}$, for $j = 2, \cdots, n$
  Output (strap, $\mathrm{bstag}_1, \cdots, \mathrm{bstag}_m, \delta_1, \cdots, \delta_m, \mathrm{bxtrap}_2, \cdots, \mathrm{bxtrap}_n$, env as search token

Algorithm 13 MC-ODXT Search

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

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

C REDUNDANT BLOOM FILTER

The plain Bloom filter (denoted by BF) is a probabilistic data structure suitable for membership checking within a set encoded in the BF. At a high level, to index an element (or insert), a BF uses k different hash functions to obtain k addresses, which are set to 1. During membership check (or look-up), k indices are obtained from the queried element and checked for 1's. These BF insertion and look-up/query routines are outlined in Algorithm 14 and Algorithm 15 respectively.

Note that if we directly plug BF into MC-ODXT, replacing the XSet insertion with BF.Insert and XSet retrieval with BF.Query, the construction essentially works the same. However, the leakage is not mitigated as the bfidx_is (or xtags in the context of SSE) generated during BF query are deterministically generated using k hash functions. Hence, the server still can associate a w using the observed BF (or XSet) addresses.

Algorithm 14 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 $\{h_1, \ldots, h_k\}$ for BF indices
  Initialise empty index set BFIdxSet
  for $i \leftarrow 1$ to k **do**
    $\text{bfidx}_i \leftarrow h_i(\text{x})$
    BFIdxSet = BFIdxSet $\cup \{\text{bfidx}_i\}$
  Shuffle elements in BFIdxSet
  Send BFIdxSet to the server
  Server
  for Each idx $\in$ BFIdxSet **do**
    Set BF[idx] = 1

Algorithm 15 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 $\{h_{i_1}, \ldots, h_{i_k}\}$ for **BF** indices
  Initialise empty index set BFIdxSet
  for $j \in \{i_1, \ldots, i_k\}$ **do**
    $\text{bfidx}_j \leftarrow h_j(\text{x})$
    BFIdxSet = BFIdxSet $\cup \{\text{bfidx}_j\}$
  Shuffle elements in BFIdxSet
  Send BFIdxSet to the server
  Server
  for Each idx $\in$ BFIdxSet **do**
    if BF[idx] $\neq 1$ **then**
      Return False
  Return True

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 (w, id) pair). During a query, BF accesses only a random subset of size k of these ℓ indices. Such that the server "sees" that each time different k indices are being accessed and can not correlate with the recorded information. We call this redundant element-based Bloom filter construction as a Redundant Bloom Filter (RBF). The updated RBF.Insert and RBF.Query routines are provided in Algorithm 16 and Algorithm 17 respectively.

Algorithm 16 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 $\{h_1, \ldots, h_\ell\}$ for RBF indices
  Initialise empty index set RIdxSet
  for $i \leftarrow 1$ to ℓ **do**
    $\text{rbfidx}_i \leftarrow h_i(\text{x})$
    RIdxSet = RIdxSet $\cup \{\text{rbfidx}_i\}$
  Shuffle elements in RIdxSet
  Send RIdxSet to the server
  Server
  for Each idx $\in$ RIdxSet **do**
    Set RBF[idx] = 1

Algorithm 17 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**
    $\text{rbfidx}_j \leftarrow h_j(\text{x})$
    RIdxSet = RIdxSet $\cup \{\text{rbfidx}_j\}$
  Shuffle elements in RIdxSet
  Send RIdxSet to the server
  Server
  for Each idx $\in$ RIdxSet **do**
    if RBF[idx] $\neq 1$ **then**
      Return False
  Return True

C.0.1 RBF Overhead. In RBF, the server sets ℓ locations, and accesses k locations during query, where $\ell > k$. The value of k needs to be chosen suitably to have negligible false positive probability (similar to normal BF) without blowing up storage.

Storage overhead. A conventional BF requires

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

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.

Communication overhead. An RBF sends k indices for a single element while inserting into and querying on BF. Thus, the communication overhead can be expressed as $O(1)$ for each inserted element (from $O(k)$, 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 the insertion of a single element 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 the same k values are chosen for RBF and BF, the communication overhead essentially remains the same for both RBF and BF during look-up.

C.1 Security Analysis of RBF

The redundant access in RBF aims to mitigate the cross-term leakage in a practical yet finite manner. We analyse the effect of repeated RBF look-ups to evaluate the leakage profile of RBF. Since RBF relies on plain redundancy rather than standard cryptographic hardness assumptions, the analysis primarily assesses the advantage of RBF compared to BF (for which the CrossAttack of Section 3 in the main manuscript works) with respect to the redundancy parameters ℓ and k.

Recall from Section 4.2 of the main manuscript that RBF uses ℓ hashes to index a value during insertion, while it uses a random k-subset of these ℓ hashes during look-up to hide the access pattern. Since it requires more number of RBF accesses to deterministically relate a specific $(\mathsf{w}, \mathsf{id})$ pair to the ℓ indices in RBF, compared to a single on in BF, in practice it results in less leakage.

Previous works [1, 22] explored that redundancy-based approaches would require a linear amount of storage on the client side to eliminate access pattern leakage (which indicates that pattern leakage may not be possible to eliminate in practice). Hence, RBF focuses on reducing the attack probability compared to a plain BF up to a limit. We argue in this discussion that, in practice, it is not feasible for the adversary to build the required $(\mathsf{w}, \mathsf{id})$–xtag association within a reasonable time. Hence, this brings a notion of access upper-bound in RBF beyond which the advantage would reduce to a traditional BF. Analysing this RBF access upper bound is equivalent to addressing the following question.

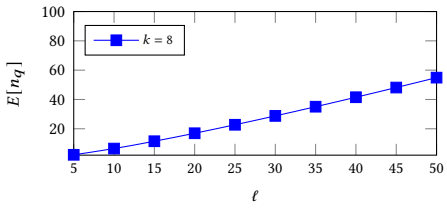

Figure 11: Growth of the RBF access upper bound with redundancy. The look-up indexing hashes were fixed at random 8-sized subset.

Maximum how many RBF access does the adversarial server needs to "see" for the same (w, id) pair from a benign client's queries before it can figure out that the same (w, id) pair is being accessed?

As it turns out, this problem is identical to the well-known *coupon collector's problem* with coupons collected in batches. When translated to the coupon collector's problem, the problem statement can be expressed in the following way.

Maximum how many draws are necessary to collect all ℓ coupons if in each draw uniformly random k-subset of the ℓ coupons is collected with replacement?

The closed-form expression of this upper bound can be expressed as below.

$$E[n_q] = \sum_{j=1}^{\ell} (-1)^{j+1} \cdot \binom{\ell}{k} \cdot \frac{\binom{\ell}{j}}{1 - \frac{\binom{\ell - j}{k}}{\binom{\ell}{j}}} \tag{1}$$

The complete proof can be found in [33] (albeit in a slightly different form). We plot the growth of this upper bound $E[n_q]$ with the number of indexing hashes ℓ in Figure 11, which shows the monotonically increasing behaviour. The parameter k remains constant to maintain the same desired false positive probability of a BF. This bound grows slower than the exponential that is ideally required to eliminate the access pattern completely in practice. We argue that if practical client search policies are formed based on the query semantics and the upper bound, this approach prevents cross-term–based leakage.

In reality, a client's queries are concentrated on a particular topic, and a client typically issues only a limited number of queries in a search session. Therefore, the essential requirement of encountering the same $(\mathsf{w}, \mathsf{id})$ pair across these queries is limited to a query session only. It is unlikely that a client would issue more queries than the upper bound $E[n_q]$ in a session, as plotted in Figure 11. Thus, the administrator can enforce the search policy such that a client cannot issue more queries than the upper bound $E[n_q]$. We state the following assumption to reflect the RBF's influence in Nomos security analysis.

Assumption. Given an RBF with ℓ hashes for insertion and a uniformly random k-subset of these ℓ hashes for look-up, τ number of RBF look-ups for the same $(\mathsf{w}, \mathsf{id})$ pair are indistinguishable from a random RBF element (corresponding to a random $(\mathsf{w}, \mathsf{id})$ pair) look-up using a uniformly random k-subset of these ℓ hashes up to $E[n_q]$ accesses, where $\tau < E[n_q]$.

Remarks. The current analysis of access bound assumes the server would definitely be able to figure out after the expected number of queries given by the closed-form expression in Equation (1). In that case, the problem is the same as the Coupon collector's problem, as stated above and followed in the abovementioned analysis. In case of an approximation or relatedness measure is considered in the attack process instead of an exact match for xtags to figure out if two sets of k xtags are "related" rather than knowing that these are from exactly the same (w, id) pair. In that case, a different analysis approach is necessary. However, the exact-match base analysis suffices to provide an in-depth overview of the RBF security, and we resorted to the upper-bound-based analysis to choose RBF parameters and simplify the analysis, compared to a "relation"-based analysis which we leave for as an extension of this work.