Result-pattern-hiding Conjunctive Searchable Symmetric Encryption with Forward and Backward Privacy

Dandan Yuan
The University of Auckland
Auckland, New Zealand
dyua568@aucklanduni.ac.nz

Cong Zuo†
Beijing Institute of Technology
Beijing, China
zuocong10@gmail.com

Shujie Cui*
Monash University
Melbourne, Australia
Shujie.Cui@monash.edu

Giovanni Russello
The University of Auckland
Auckland, New Zealand
g.russello@auckland.ac.nz

ABSTRACT

Dynamic searchable symmetric encryption (DSSE) enables the data owner to outsource its database (document sets) to an untrusted server and make searches and updates securely and efficiently. Conjunctive DSSE can process conjunctive queries that return the documents containing multiple keywords. However, a conjunctive search could leak the *keyword pair result pattern* (KPRP), where attackers can learn which documents contain any two keywords involved in the query. File-injection attack shows that KPRP can be utilized to recover searched keywords. To protect data effectively, DSSE should also achieve *forward privacy*, *i.e.,* hides the link between updates to previous searches, and *backward privacy*, *i.e.,* prevents deleted entries being accessed by subsequent searches. Otherwise, the attacker could recover updated/searched keywords and records. However, no conjunctive DSSE scheme in the literature can hide KPRP in sub-linear search efficiency while guaranteeing forward and backward privacy.

In this work, we propose the first sub-linear KPRP-hiding conjunctive DSSE scheme (named HDXT) with both forward and backward privacy guarantees. To achieve these three security properties, we introduce a new cryptographic primitive: Attribute-updatable Hidden Map Encryption (AUHME). AUHME enables HDXT to efficiently and securely perform conjunctive queries and update the database in an oblivious way. In comparison with previous work that has weaker security guarantees, HDXT shows comparable, and in some cases, even better performance.

KEYWORDS

Dynamic searchable symmetric encryption, Conjunctive, Attribute-updatable hidden map encryption

1 INTRODUCTION

Searchable symmetric encryption (SSE) enables the client to outsource an encrypted database to an untrusted server and then search it securely. Dynamic SSE (DSSE) allows the client to securely update the database. In a typical setting of SSE, the database DB is a collection of documents associated with a search index, commonly represented by a set of keyword-document pairs. If a keyword-document pair is in the index, it means that the document contains the keyword. A search query returns the documents that have a specific relationship with searched keyword(s). The index, documents, and queries are all encrypted before being sent to the server.

An ideal goal of SSE is to efficiently and securely support query types as rich as the plaintext database, such as single-keyword query [7, 8, 11, 13, 15, 28, 43–45, 53, 56], Boolean query [12, 25, 31, 37, 39], range query [48, 55], and update query. However, there exists a trade-off among performance, security, and functionality for SSE. Existing SSE schemes usually achieve better performance and/or functionality at the cost of information leakage. For instance, Cash *et al.* [11] designed a SSE, called OXT, that sub-linearly supports *conjunctive query*, represented as $w_1 \wedge \cdots \wedge w_n$, *i.e.,* search the documents containing the n keywords, where $n > 1$. However, it leaks $\mathrm{DB}(w_1) \cap \mathrm{DB}(w_j)$, where $\mathrm{DB}(w)$ is the set of the documents containing the keyword w, and $2 \le j \le n$. Such leakage is referred to as *keyword pair result pattern* (KPRP), and it can be generalised to $\mathrm{DB}(w_i) \cap \mathrm{DB}(w_j)$, where $1 \le i < j \le n$. The file-injection attack [54] shows that KPRP leakage is not acceptable as attackers could leverage KPRP to first recover $\mathrm{DB}(w_i)$ and then learn w_i for $1 \le i \le n$, by injecting documents into the database.

In the dynamic setting, *forward and backward privacy* has been identified by the literature [8, 13, 39, 48, 55, 56] as two crucial security notions for DSSE. Forward privacy hides the link between an update query to previous searches. Achieving forward privacy is essential to resist the file-injection attack [54], otherwise updated keywords can be recovered. Backward privacy ensures that search queries do not reveal the results that were deleted. Bost *et al.* [8] introduce three types of backward privacy: from Type-I that has the least leakages to Type-III which reveals the most information.

A naive solution to hide the KPRP is to search each keyword with *response-hiding* single-keyword SSE, such as MITRA [13], and intersect the results on the client. *Response-hiding* SSE does not reveal the search result (*i.e.,* the identifiers of matching documents) in plaintext to the server. Despite the adopted single-keyword SSE could be

---

* Corresponding author
† This work was done while the author was at Nanyang Technological University.

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
Proceedings on Privacy Enhancing Technologies 2023(2), 40–58
© 2023 Copyright held by the owner/author(s).
https://doi.org/10.56553/popets-2023-0040

Table 1: Comparison with Existing Conjunctive SSE Schemes

Schemes	**Search**			**Update**			**Storage**		**Forward**	**Backward**	**KPRP**
	Computation	Communication	Round	Add	Edit	Delete	Client	Server	**Privacy**	**Privacy**	**Hiding**
Blind Seer[35]	$O(\gamma nm \log \lvert\mathrm{D}\rvert)$	$O(\gamma nm \log \lvert\mathrm{D}\rvert)$	$\log \lvert\mathrm{D}\rvert$	static			$O(1)$	$O(\lvert\mathrm{W}\rvert\lvert\mathrm{D}\rvert)$	-	-	✓
HXT[31]	$O(\gamma n \lvert\mathrm{DB}(w_1)\rvert)$	$O(\gamma n \lvert\mathrm{DB}(w_1)\rvert)$	2	static			$O(1)$	$O(\xi N)$	-	-	✓
DIEX[25]	$O(\pi_1 + n\lvert\mathrm{DB}(w_1)\rvert)$	$O(n+m)$	1	$O(\mathrm{W}_d)$	$O(v)$	$O(1)$	$O(\lvert\mathrm{W}\rvert \log \lvert\mathrm{D}\rvert)$	$O(\lvert\mathrm{W}\rvert^2 h)$	✓	✗	✗
VBTree[50]	$O(\tau \lvert\mathrm{DB}(w_1)\rvert \log \lvert\mathrm{D}\rvert)$	$O(n+m)$	1	$O(\log \lvert\mathrm{D}\rvert)$			$O(\lvert\mathrm{W}\rvert \log \lvert\mathrm{D}\rvert)$	$O(N \log \lvert\mathrm{D}\rvert)$	✓	✗	✗
BDXT[39]	$O(\pi_1 + n\lvert\mathrm{DB}(w_1)\rvert)$	$O(\pi_1 + n\lvert\mathrm{DB}(w_1)\rvert)$	2	$O(1)$	-	$O(1)$	$O(\lvert\mathrm{W}\rvert \log \lvert\mathrm{D}\rvert)$	$O(N)$	✓	Type-II	✗
ODXT[39]	$O(n\pi_1)$	$O(n\pi_1)$	1	$O(1)$	-	$O(1)$	$O(\lvert\mathrm{W}\rvert \log \lvert\mathrm{D}\rvert)$	$O(N)$	✓	Type-II	✗
FBSSE-CQ[57]	$O(\tau \lvert\mathrm{D}\rvert)$	$O(n + \lvert\mathrm{D}\rvert)$	1	$O(\lvert\mathrm{D}\rvert)$			$O(\lvert\mathrm{W}\rvert(\log \lvert\mathrm{D}\rvert + \lambda)$	$O(N\lvert\mathrm{D}\rvert)$	✓	Type-II	✓
HDXT	$O(\pi_1 + n\lvert\mathrm{DB}(w_1)\rvert)$	$O(\pi_1 + n\lvert\mathrm{DB}(w_1)\rvert)$	2	$O(\lvert\mathrm{W}\rvert/\mathrm{W}_d)$	$O(\lvert\mathrm{D}\rvert)$	$O(1)$	$O(\lvert\mathrm{W}\rvert(\log \lvert\mathrm{D}\rvert + \lambda)$	$O(\lvert\mathrm{W}\rvert\lvert\mathrm{D}\rvert)$	✓	Type-II	✓

$|\mathrm{W}|$, $|\mathrm{D}|$, and N are the number of keywords, documents, and keyword-document pairs in the database, respectively. $N \leq |\mathrm{W}||\mathrm{D}|$. n denotes the number of keywords involved in the query. m is the number of documents matching the query. π_1 is the number of updates related to w_1. $\tau = \sum_{i=1}^{n} \tau_i$, where τ_i is the number of updates related to w_i since the last search involved w_i. γ and ξ are the parameters for Bloom filter, which typically are 20 and 29. For an update, W_d denotes the number of keywords contained in the updated document and v denotes the number of keywords involved in an edit operation. λ is the security parameter and h denotes the average number of documents matched by any two keywords in the database.

forward and backward private, the naive solution causes computational and communication overhead worse than $O(\sum_{i=1}^{n} |\mathrm{DB}(w_i)|)$, which is inefficient especially when one or more of the keywords in the search query have high-frequency occurrence.

To the best of our knowledge, no DSSE scheme in the literature can support conjunctive queries securely and efficiently. As shown in Table 1, existing SSE schemes that support conjunctive queries are either static or leak KPRP (the static ones with KPRP leakage are not included in the table). Only Zuo *et al.*' scheme FBDSSE-CQ [57] hides KPRP while ensuring forward and Type-II backward privacy, but at the expense of linear search overheads.

Our Work. In this paper, we aim to fill this gap, *i.e.,* design a forward and backward private DSSE scheme that sub-linearly supports conjunctive queries and hides the KPRP. Our solution is inspired by HXT [31], a static SSE that supports KPRP-hiding conjunctions. However, compared with HXT [31], our solution is more efficient and supports update queries with solid security guarantees: thus, we named our approach HDXT. As done in OXT [12] and HXT [31], the big idea of HDXT is to perform the conjunctive search in two steps: searches for $\mathrm{DB}(w_1)$, and filters out those results that do not match $w_2 \wedge \cdots \wedge w_n$. w_1 is the keyword with the minimum occurrence among the n keywords, and it is called *s-term*. The other keywords are called *x-terms*. In HDXT, $\mathrm{DB}(w_1)$ is obtained with a response-hiding single-keyword DSSE scheme. The challenge lies in how to perform securely and efficiently the second step.

To overcome the challenge, we introduce a new cryptographic primitive: attribute-updatable hidden map encryption (AUHME). AUHME allows us to query if a set of pairs $\mathbf{m_p}$ is a subset of a larger set $\mathbf{m_a}$ securely. If the answer is no, it does not leak which pair(s) in $\mathbf{m_p}$ does not belong to $\mathbf{m_a}$. This property enables us to perform the second step without leaking KPRP. Specifically, assume W is the set of all the keywords in DB and D contains all the document identifiers of DB. HDXT has an index structure $\mathrm{DB}' = \{(w||id, v) \mid w \in \mathrm{W}, id \in \mathrm{D}\}$, where v is either 1 or 0, indicating whether document id contains w or not, respectively. DB' is encrypted with AUHME. For performing the second step, the client constructs $I_{id} = \{(w_2||id, 1), \cdots, (w_n||id, 1)\}$ for each $id \in \mathrm{DB}(w_1)$ and queries whether I_{id} is a subset of DB' with AUHME. If the subset query succeeds, id matches the conjunctive query. Otherwise, AUHME ensures whether $(w_i||id, 1)$ is in DB' for every $1 \leq i \leq n$ is concealed, which protects $\mathrm{DB}(w_1) \cap \mathrm{DB}(w_i)$, indicating KPRP is hidden successfully.

DB' can be securely updated with the update function of AUHME. Basically, the client caches recent updates locally and evicts the cache to DB' when it is full. The eviction is processed in an oblivious way such that the server cannot learn which entries of DB' were updated. Also, for any subset query for I_{id} ($id \in \mathrm{DB}(w_1)$) in a subsequent search, the server only learns the query result with respect to the latest DB', which will not reveal any information about the deleted keyword-document pairs. Consequently, the queries and updates over DB' satisfy forward and the highest level of backward privacy. That is, HDXT achieves forward and backward security as long as the adopted single-keyword DSSE does.

In Table 1, we summarise the performance overheads and security properties achieved by HDXT and other conjunctive DSSE schemes[1]. Compared with FBDSSE-CQ, HDXT has much less computational and communication overhead for search queries. HDXT also has less storage overhead on the server side. In Section 5.2, we compare HDXT with other schemes in detail.

We also experimentally compare the performance of HDXT with HXT and MITRA$_{\mathrm{CONJ}}$ [39] (the naive solution implemented by Patranabis and Mukhopadhyay). The results show that HDXT is 10.7× and 13× faster than HXT and MITRA$_{\mathrm{CONJ}}$ respectively, for the queries involving 11 keywords.

Our Contributions. Overall, our contribution can be summarized as below.

1. We are the first to introduce the concept of AUHME and design a selectively-semantically secure AUHME scheme.
2. We propose the first conjunctive DSSE scheme HDXT that hides KPRP while preserving sub-linear search efficiency. HDXT also achieves forward privacy and backward privacy with the level of at least Type-II.
3. We implement a prototype of HDXT and evaluate its performance with real-world datasets.
4. We prove that our AUHME scheme is selectively-semantically secure, and HDXT is adaptively secure while achieving the three security properties mentioned above.

2 PRELIMINARIES

In this section, we first introduce the notations used in the following sections. Then we provide the definitions for AUHME and DSSE.

---

[1] When analyzing HDXT, we assume that single-keyword DSSE is instantiated with MITRA [13], which is the state-of-art that realizes forward and Type-II backward privacy. Note that HDXT further satisfies Type-I backward privacy when the adopted single-keyword DSSE is Type-I backward private, such as ORION [13].

2.1 Notations

Throughout this paper, $\{0,1\}^l$ denotes the set of all binary strings of length l. $\{0,1\}^*$ denotes the set of arbitrary strings. 0^l represents the binary string of length l where every bit is 0. || denotes the concatenation of two strings. $\perp$ represents an empty string. $a_1 \overset{\$}{\leftarrow} S$ means a_1 is sampled uniformly at random from the set S. $|X|$ represents the cardinality of a set/map/list X.

A map X is a data structure that associates keys to values, where each entry contains exactly one unique key and its corresponding value. We also consider X as a set that contains (key, value) pairs.We use $X : S_1 \mapsto S_2$ to represent that the space for keys is S_1 and the space for values is S_2, $X \sqsubseteq S_1 \mapsto S_2$ to denote that the key space of X is a subset of (or equal to) S_1 and the value space of X is S_2.

2.2 Attribute-updatable Hidden Map Encryption

Predicate encryption can encrypt a message associated with an attribute A to a ciphertext and generate a key SK corresponding to a predicate f such that the ciphertext can be correctly decrypted using SK if and only if $f(A) = 1$, while ensuring that nothing about the message is leaked if $f(A) = 0$. This security property is called *payload-hiding*. The predicate encryption is *attribute-hiding* if the ciphertext also conceals information about A.

In this paper, we introduce a special attribute-hiding predicate encryption: Hidden Map Encryption (HME), where the attribute A is a map. Let $\mathcal{K}$, $\mathcal{K}_a$, and $\mathcal{V}$ be three finite sets, where $\mathcal{K}_a \subseteq \mathcal{K}$. HME works for a class of predicates $\Phi^{\text{hme}} = \{\phi^{\text{hme}}_{\mathbf{m_p}} | \mathbf{m_p} \sqsubseteq \mathcal{K}_a \mapsto \mathcal{V}\}$ where, for an attribute map $\mathbf{m_a} : \mathcal{K}_a \mapsto \mathcal{V}$,

$$\phi^{\text{hme}}_{\mathbf{m_p}}(\mathbf{m_a}) = \begin{cases} 1 & \text{if } \mathbf{m_p}[k] = \mathbf{m_a}[k] \text{ for each key } k \text{ in } \mathbf{m_p} \\ 0 & \text{otherwise} \end{cases}$$

That is, $\phi^{\text{hme}}_{\mathbf{m_p}}(\mathbf{m_a})$ is satisfied when the pairs in $\mathbf{m_p}$ are all included in $\mathbf{m_a}$, and we say $\mathbf{m_p}$ is a subset of $\mathbf{m_a}$ in this case.

We introduce the *attribute-updatability* property to HME, which means the attribute map can be updated without reproducing the ciphertext from scratch. Specifically, attribute-updatable HME (AUHME) supports two kinds of updates: adding a pair into $\mathbf{m_a}$ and editing the value of an existing pair in $\mathbf{m_a}$. Deleting a pair can be achieved through editing the value of the pair to $\perp$. Formally, in the symmetric-key setting, it consists of the following six algorithms:

- **Setup**$(1^\lambda) \rightarrow (msk, \delta)$: On input the security parameter 1^λ, it outputs a master secret key msk and a state δ.
- **Enc**$(msk, \mathbf{m_a} : \mathcal{K}_a \mapsto \mathcal{V}, M) \rightarrow C$: Taking as input the master secret key msk, an attribute map $\mathbf{m_a}$, and a message M, it outputs the ciphertext C.
- **GenKey**$(msk, \delta, \mathbf{m_p} \sqsubseteq \mathcal{K}_a \mapsto \mathcal{V}) \rightarrow dk$: Taking as input the master secret key msk, the current state δ, and a predicate map $\mathbf{m_p}$, it outputs a decryption key dk.
- **Query**$(dk, C) \rightarrow M$ or $\perp$: On input a decryption key dk and the ciphertext C, it outputs M or $\perp$.
- **GenUpd**$(msk, \delta, op, u_1 \in \mathcal{K}, u_2 \in \mathcal{V}) \rightarrow (UTok, \delta')$: On input the master secret key msk, the current state δ, an operator $op \in \{add, edit\}$, and a pair (u_1, u_2), it produces an update token $UTok$ and a possibly updated state δ'. Note that if $op = add$, u_1 is inserted into $\mathcal{K}_a$.
- **ApplyUpd**$(\text{UTok}, C) \rightarrow C'$: On input a token $UTok$ and the ciphertext C, it outputs the updated ciphertext C'.

The correctness of an AUHME scheme requires that, for all possible legal inputs, after running **Enc**, **GenKey**, and even after performing a polynomial number of updates on $\mathbf{m_a}$ with **GenUpd** and **ApplyUpd**, if $\phi^{\text{hme}}_{\mathbf{m_p}}(\mathbf{m_a}) = 1$, then **Query**$(dk, C) = M$, otherwise **Query**$(dk, C) = \perp$ with all but negligible probability.

A variation of predicate encryption is a *predicate-only* scheme where the inputs of **Enc** do not include any M, and **Query** only reveals whether the predicate is satisfied. For a predicate-only HME, **Query**$(dk, C) = \phi^{\text{hme}}_{\mathbf{m_p}}(\mathbf{m_a})$ for any $dk \leftarrow$ **GenKey**$(msk, \delta, \mathbf{m_p})$.

$\text{AUHMEReal}_{\mathcal{A}}(\lambda)$:

1. $\mathcal{A}(1^\lambda)$ outputs an attribute map $\mathbf{m_a} : \mathcal{K}_a \mapsto \mathcal{V}$. **Setup**$(1^\lambda)$ is run to generate (msk, δ).
2. $\mathcal{A}$ may make ρ_1 queries in an adaptive way. For a key generation query on a predicate map $\mathbf{m_p}$, $\mathcal{A}$ is given dk generated by **GenKey**$(msk, \delta, \mathbf{m_p})$. For an update query (op, u_1, u_2), $\mathcal{A}$ is given the update token $UTok$ outputted by **GenUpd**$(msk, \delta, op, u_1, u_2)$.
3. $\mathcal{A}$ chooses a message M and is given the ciphertext C generated by **Enc**$(msk, \mathbf{m_a}, M)$.
4. $\mathcal{A}$ may make ρ_2 queries adaptively, which are processed as in (2)
5. With the view observed by $\mathcal{A}$ as the input, $\mathcal{A}$ outputs a bit b.

$\text{AUHMEIdeal}_{\mathcal{A},\mathcal{S}}(\lambda)$:

1. $\mathcal{A}(1^\lambda)$ outputs an attribute map $\mathbf{m_a} : \mathcal{K}_a \mapsto \mathcal{V}$.
2. $\mathcal{A}$ may adaptively makes ρ_1 queries. For a query on $\mathbf{m_p}$, $\mathcal{A}$ is given dk outputted by $\mathcal{S}(\mathcal{L}^h_q(\mathbf{m_p}), \phi^{\text{hme}}_{\mathbf{m_p}}(\mathbf{m_a}))$. For an update (op, u_1, u_2), $\mathcal{A}$ is given $UTok$ generated by $\mathcal{S}(\mathcal{L}^h_u(op, u_1, u_2))$.
3. $\mathcal{A}$ chooses a message M and is given the ciphertext C generated by $\mathcal{S}(1^{|M|}, |\mathbf{m_a}|)$.
4. $\mathcal{A}$ may make ρ_2 queries adaptively as in (2).
5. Taking as input the view observed by $\mathcal{A}$, $\mathcal{A}$ outputs a bit b.

Figure 1: Selective Simulation-Based Definition of AUHME

Intuitively, the security for AUHME requires that the adversary learns nothing about M and $\mathbf{m_a}$, a query only reveals the query result, and an update discloses nothing. Here we consider the relaxed security where queries and updates might leak a little information. We denote the allowed leakage as $\mathcal{L}^h = (\mathcal{L}^h_q(\mathbf{m_p}), \mathcal{L}^h_u(op, u_1, u_2))$, which captures the query and update leakages, respectively. Briefly, we require that $\mathcal{L}^h_q(\mathbf{m_p})$ only exposes which keys exist in the predicate map $\mathbf{m_p}$, which is called *key pattern*. $\mathcal{L}^h_u(op, u_1, u_2)$ can reveal op but leak nothing about (u_1, u_2). In Definition 2.1, we provide the security definition for AUHME in the simulation-based setting.

Definition 2.1. *We say an AUHME scheme is $\mathcal{L}^h$-selectively-semantically secure if, for any security parameter λ and any probabilistic polynomial-time adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ and a negligible function negl such that;*

$$|\Pr(\text{AUHMEReal}_{\mathcal{A}}(\lambda) = 1) - \Pr(\text{AUHMEIdeal}_{\mathcal{A},\mathcal{S},\mathcal{L}^{\text{h}}}(\lambda) = 1)| \leqslant \text{negl}(\lambda)$$

where $\text{AUHMEReal}_{\mathcal{A}}(\lambda)$ *and* $\text{AUHMEIdeal}_{\mathcal{A},\mathcal{S},\mathcal{L}^h}(\lambda)$ *are shown in Fig.1*[2].

---

[2] For predicate-only AUHME, we omit the message M in Fig.1.

2.3 Dynamic Searchable Symmetric Encryption

Let the database DB be $\{(id_i, \mathrm{W}_i)\}_{i=1}^{|\mathrm{D}|}$, where $id_i \in \{0,1\}^l$ is the identifier of a document and $\mathrm{W}_i \subseteq \{0,1\}^*$ is the set of keywords contained in the document id_i. $\mathrm{D} = \{id_i\}_{i=1}^{|\mathrm{D}|}$ and $\mathrm{W} = \cup_{i=1}^{|\mathrm{D}|}\mathrm{W}_i$ store all the document identifiers and keywords in the database, respectively. Given a search formula $\psi(\overline{\mathrm{w}})$ involving a collection of keywords $\overline{\mathrm{w}} \subseteq \mathrm{W}$, $\mathrm{DB}(\psi(\overline{\mathrm{w}}))$ represents the identifiers of the documents that satisfy $\psi(\overline{\mathrm{w}})$. $\psi(\overline{\mathrm{w}})$ is a conjunctive query if it combines every keyword $w \in \overline{\mathrm{w}}$ with the operator '$\wedge$' (AND). An identifier id_i satisfies a conjunction over $\overline{\mathrm{w}}$ iff $\overline{\mathrm{w}} \subseteq \mathrm{W}_i$. Moreover, we define the extended database DB′ to be $\{(w||id, b) \mid w \in \mathrm{W}, id \in \mathrm{D}\}$, where b is 1 if $id \in \mathrm{DB}(w)$, and is 0 otherwise. Finally, note that a dynamic database supports inserting a new document into the database (*add*), adding keywords in W to an existing document ($edit^+$), removing keywords from an existing document ($edit^-$), and deleting documents from the database (*del*). Formally, DSSE consists of the following three protocols:

- **Setup**$(\lambda, \mathrm{DB}; \perp) \rightarrow (K, s; \mathrm{EDB})$: On input the security parameter λ, and a database DB, the client outputs a secret key K and a state s. The server outputs an encrypted database EDB without any input.
- **Search**$(K, s, \psi(\overline{\mathrm{w}}); \mathrm{EDB}) \rightarrow (s', \mathrm{DB}(\ \psi(\overline{\mathrm{w}})); \mathrm{EDB}')$: The client takes as input the secret key K, the current state s, and a search formula $\psi(\overline{\mathrm{w}})$. The server has EDB as the input. Eventually, the client outputs a possibly updated state s' and the search result $\mathrm{DB}(\psi(\overline{\mathrm{w}}))$. The server outputs a possibly updated encrypted database EDB′.
- **Update**$(\mathrm{K}, s, op, in; \mathrm{EDB}) \rightarrow (s'; \mathrm{EDB}')$: The client has five parameters as inputs that include the secret key K, the current state s, an operator $op \in (add, edit^+, edit^-, del)$, and the updated information $in = (id, \mathrm{W}_{id})$ where id is a document identifier and W_{id} is a collection of keywords. The server takes as input the encrypted database EDB. Finally, the client outputs an updated secret state s', and the server outputs an updated encrypted database EDB′.

The correctness for SSE requires that for every database DB, every encrypted database EDB generated from DSSE.**Setup** or DSSE.**Update**, and every supported search formula $\psi(\overline{\mathrm{w}})$, the search query on $\psi(\overline{\mathrm{w}})$ should return $\mathrm{DB}(\psi(\overline{\mathrm{w}}))$ to the client.

As done in previous literature [7, 11, 28], we use three functions $\mathcal{L} = (\mathcal{L}^{Stp}(\mathrm{DB}), \mathcal{L}^{Srch}(\mathrm{DB}, \psi(\overline{\mathrm{w}})), \mathcal{L}^{Updt}(\mathrm{DB}, op, in))$ to capture the leakages for the setup, search, and update protocols, respectively. We borrow the formal definition for DSSE from [11, 28], which is shown in Definition 2.2.

Definition 2.2. *Let* $\prod$ *=* ***{Setup, Search, Update}*** *denote a DSSE scheme. We say* $\prod$ *is* $\mathcal{L}$ *− adptively − secure if for any security parameter* λ*, any probabilistic polynomial-time adversaries* $\mathcal{A}$*, there exist a a simulator* $\mathcal{S}$ *and a negligible function negl such that:*

$$|\Pr(\mathrm{SSEReal}_{\mathcal{A}}^{\prod}(\lambda) = 1) - \Pr(\mathrm{SSEIdeal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\prod}(\lambda) = 1)| \leqslant \mathrm{negl}(\lambda)$$

where $\mathrm{SSEReal}_{\mathcal{A}}^{\prod}(\lambda)$ *and* $\mathrm{SSEIdeal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\prod}(\lambda)$ *are defined as:*

- $\mathrm{SSEReal}_{\mathcal{A}}^{\prod}(\lambda)$*: At first,* $\mathcal{A}$ *chooses a database* DB*, and obtains* EDB *by invoking the function* ***Setup****(*λ, DB*). Then it repeatedly performs search queries* ***Search****(*$\psi(\mathbf{w})$*) and update queries* ***Update****(*op, w, id*) in an adaptive way.* $\mathcal{A}$ *receives all the transcripts generated during the above operations, and outputs a bit* b*.*
- $\mathrm{SSEIdeal}_{\mathcal{A},\mathcal{S},\mathcal{L}}^{\Sigma}(\lambda)$*:* $\mathcal{A}$ *chooses a database* DB*, and calls* $\mathcal{S}(\mathcal{L}^{Stp}(DB))$ *to get the encrypted database* EDB*. After that, it adaptively performs search queries (update queries) by calling* $\mathcal{S}(\mathcal{L}^{Srch}(\mathrm{DB}, \overline{\mathrm{w}}))$ *(*$\mathcal{S}(\mathcal{L}^{Updt}(\mathrm{DB}, op, w, id))$*).* $\mathcal{A}$ *observes the transcripts of all operations and outputs a bit* b*.*

Forward Privacy. Forward privacy requires that an update reveals nothing about the updated keyword. We borrow the definition from [7, 8], which is shown in Definition 2.3.

Definition 2.3. ***(Forward Privacy of Conjunctive DSSE)*** *A* $\mathcal{L}$*−adptively−secure DSSE* Σ *=* ***{Setup, Search, Update}*** *is forward private iff the update leakage function* $\mathcal{L}^{Updt}$ *can be written as:*

$$\mathcal{L}^{Updt}(\mathrm{DB}, op, (id, \mathrm{W}_{id})) = \mathcal{L}'(op, id, |\mathrm{W}_{id}|)$$

where $\mathcal{L}'$ *is a stateless function.*

Backward Privacy. Backward privacy limits what the server could learn about a deleted entry from the queries issued after the deletion. Bost *et al.* [8] introduce three types of backward privacy for single-keyword DSSE, from Type-I to Type-III. Briefly, Type-I requires that a single-keyword search on w only reveals $\mathrm{DB}(w)$, when each document in $\mathrm{DB}(w)$ is inserted, and the total number of updates related to w. Type-II additionally leaks the timestamps of the updates related to w. The leakages of Type-III also include which deletion cancels which addition. To extend the definition to conjunctive DSSE, similar to [48], we say that a multi-keyword DSSE is backward private iff the update and search leakages about every keyword do not exceed what is revealed by a backward private single-keyword DSSE. Nevertheless, Bost *et al.*' definition has two assumptions: the initial database is empty; a keyword w cannot be inserted into the document from which w was previously removed. We generalize their definition by eradicating the two assumptions. Since our DSSE scheme at least achieves Type-II, we only define Type-I and Type-II backward privacy.

We use Q to represent the list of the issued queries, (t, q) to denote a conjunctive query, and (t, op, in) to stand for an update, where t is the timestamp. For a conjunction q, $q[i]$ is the i-th term involved in q. $t^{\triangleright}$ denotes the timestamp of the setup protocol and $\mathrm{DB}^{\triangleright}$ is the initial database. For a conjunction q, $\pi_i^{\triangleright}$ records the number of documents containing the keyword $q[i]$ in $\mathrm{DB}^{\triangleright}$.

For a keyword w, TimeDB(w) outputs the identifiers currently matching w and the timestamps these identifiers were first inserted into the database. Formally, $\mathrm{TimeDB}(w) = \{(t, id) | id \in \mathrm{DB}(w)$ and $\exists \mathrm{W}_{id} : (t, add, (id, \mathrm{W}_{id})) \in Q\} \cup \{(t^{\triangleright}, id) | id \in \mathrm{DB}(w)$ and id exists in $\mathrm{DB}^{\triangleright}\}$. Updates$(w)$ is the list of timestamps of updates related to w. Formally, $\mathrm{Updates}(w) = \{t | \exists \mathrm{W}_{id}$ that contains $w : (t, add, (id, \mathrm{W}_{id})) \in Q$ or $(t, edit^+, (id, \mathrm{W}_{id})) \in Q$ or $(t, edit^-, (id, \mathrm{W}_{id})) \in Q$ or $(t, del, (id, \mathrm{W}_{id})) \in Q\}$. For a conjunctive q, we write $(\mathrm{TimeDB}(q[i]))_{i=1}^n$ as $\mathrm{TimeDB}(q)$, $(\mathrm{Updates}(q[i]))_{i=1}^n$ as $\mathrm{Updates}(q)$, $(\pi_i^{\triangleright})_{i=1}^n$ as $\pi^{\triangleright}(q)$, $(\pi_i)_{i=1}^n$ as $\pi(q)$, where π_i is the sum of $\pi_i^{\triangleright}$ and the number of updates related to $q[i]$. We give the definition in Definition 2.4. Note that the existing definitions [39, 57] either have strict assumptions or are specialized to their own schemes.

Definition 2.4. ***(Backward Privacy of Conjunctive DSSE)*** *A* $\mathcal{L}$ *− adptively − secure DSSE* Σ *=* ***{Setup, Search, Update}*** *is* ***Type-I*** *backward private iff*

$$\mathcal{L}^{Updt}(\mathrm{DB}, op, (id, \mathrm{W}_{id})) = \mathcal{L}'(op, |\mathrm{W}_{id}|)$$
$$\mathcal{L}^{Srch}(\mathrm{DB}, q) = \mathcal{L}''(\mathrm{TimeDB}(q), \pi(q))$$

Type-II *backward private iff*

$$\mathcal{L}^{Updt}(\mathrm{DB}, op, (id, \mathrm{W}_{id})) = \mathcal{L}'(op, \mathrm{W}_{id})$$
$$\mathcal{L}^{Srch}(\mathrm{DB}, q) = \mathcal{L}''(\mathrm{TimeDB}(q), \mathrm{Updates}(q), \pi^{\triangleright}(q))$$

where $\mathcal{L}'$ *and* $\mathcal{L}''$ *are stateless functions.*

Definition 2.4 is applicable to single-keyword DSSE by considering a search on w as a conjunction $q = w$. Our definition differs slightly from Bost *et al.*'s definition [8] due to the complex setting we consider, for which we make a detailed analysis in Appendix C.

KPRP-hiding. The KPRP is a leakage related to the keywords involved in the same search query. A conjunction query q aims to obtain the documents containing all the keywords involved in q, *i.e.,* DB(q). The KPRP-hiding property means that the server could know DB(q) after the search, but otherwise cannot learn which other documents contain any two keywords involved in q. We define KPRP-hiding in Definition 2.5.

Definition 2.5. ***(KPRP-hiding of Conjunctive DSSE)*** *A* $\mathcal{L} - adptively - secure$ *conjunctive DSSE* $\Sigma = \{$ ***Setup, Search, Update***$\}$ *is KPRP-hiding iff the search leakage function* $\mathcal{L}^{Srch}(\mathrm{DB}, q)$ *does not reveal which identifiers belong to* $\mathrm{DB}(q[i]) \cap \mathrm{DB}(q[j])$ *for any* $1 \leq i < j \leq n$ *except for the ones in* DB(q).

3 AUHME CONSTRUCTION

This section presents our predicate-only AUHME construction. In our construction, for the attribute map, the key can be an arbitrary string and the value belongs to $\{0, 1\}$, *i.e.,* $\mathcal{K}$ is a finite set of arbitrary strings and $\mathcal{V}$ is $\{0, 1\}$. For update operations, the mapped value of any pair can only be updated to 0 or 1. Moreover, the new value must be different from the stale one; otherwise the update is invalid, which is forbidden in our construction. For simplicity, our construction does not consider deleting pairs from $\mathbf{m_a}$.

3.1 Overview of AUHME Construction

Query Process. The main purpose of our AUHME construction is to securely query if a predicate map $\mathbf{m_p}$ is a subset of the attribute map $\mathbf{m_a}$, *i.e.,* query if all the pairs in $\mathbf{m_p}$ are also included in $\mathbf{m_a}$, with two requirements: **R1**) the pairs of the two maps should be protected in any case; and **R2**) if $\mathbf{m_p}$ is not a subset of $\mathbf{m_a}$, which pairs in $\mathbf{m_p}$ are included in $\mathbf{m_a}$ should not be leaked.

To achieve **R1**, every pair in the two maps is encrypted with a pseudorandom function (PRF) $F : \{0, 1\}^{\lambda} \times \{0, 1\}^{*} \rightarrow \{0, 1\}^{\lambda}$. The ciphertexts of $\mathbf{m_a}$'s pairs are stored in a map C, where every entry is indexed by its associated encrypted key. Our strategy to achieve **R2** is based on the XOR MAC technique [4], where we XOR the ciphertexts of $\mathbf{m_p}$'s pairs and get a string $xors$. To conceal $xors$, we generate $d \leftarrow H(r||xors)$ as the query token, where r is a random string and $H : \{0, 1\}^{*} \rightarrow \{0, 1\}^{\lambda}$ is a hash function. During the query, only the ciphertexts of the pairs whose keys are included in $\mathbf{m_p}$ are picked out from C and XORed into $xors'$. Given r, we can check if $H(r||xors') = H(r||xors)$, which is true only when $xors' = xors$, indicating $\mathbf{m_p}$ is a subset of $\mathbf{m_a}$. The query process exposes the query result, $|\mathbf{m_p}|$, and the access pattern over $\mathbf{m_a}$, from which the adversary cannot break **R1** and **R2**.

Update Process. For an update, we aim to break the link between the update and previous queries, *i.e.,* conceal whether the updated key is included in any ever queried $\mathbf{m_p}$. As mentioned before, the attribute map $\mathbf{m_a}$ can be updated in two different ways: add a new pair, or edit the value of an existing pair. For an addition, the newly added pairs must have new keys, which means they must be not related to any $\mathbf{m_p}$. Thus, we can directly encrypt the new pairs with F and add their ciphertexts into C. However, it is challenging to edit pairs. During the query, the access pattern over $\mathbf{m_a}$ is leaked in order to generate $xors'$. To break the link between editing updates and queries, we have to edit the pairs obliviously; otherwise, they can be linked based on the access pattern.

To protect editing updates efficiently, we leverage an ORAM-like idea where we create a local cache for saving the recent editing updates and evict them to C when the cache is full. In the eviction procedure, we re-randomise all the pairs in C so as to hide the access pattern. Specifically, the pairs without updates are XORed with a string that does not affect their values, and the pairs with updates are XORed with a string that can change their values to the updated ones, which can be easily achieved as the value is either 1 or 0 in plaintext. The strings generated for the two cases are indistinguishable as they are encrypted with F. Thus, the adversary cannot tell which pairs are actually updated.

3.2 Details of AUHME Construction

Fig.2 gives the construction details, which are summarised as below.

- **AUHME.Setup**(1^{λ}) : It generates the secret key $msk = (k_1, k_2, k_3)$ and the initial state $\delta = (cnt, T, \zeta, S)$. Specifically, cnt counts the number of evictions that were executed. T is the cache for editing updates, which is a map with a capacity of ζ. S is $\perp$ except when performing an eviction. Within an eviction, S stores $F(k_1, k)$ for every key $k \in \mathbf{m_a}$. S can be pre-computed or pre-requested from C before the eviction.

- **AUHME.Enc**$(msk, \mathbf{m_a})$: The algorithm produces the ciphertext C, which is in the form of a map. For every element $(k_a, v_a) \in \mathbf{m_a}$, the corresponding element in C is (ℓ, v), where $\ell \leftarrow F(k_1, k_a)$ and $v \leftarrow F(k_2, \ell||v_a) \oplus F(k_3, \ell||cnt)$.

- **AUHME.GenUpd**$(msk, \delta, op, k_u, v_u)$: Given the pair (k_u, v_u) and operator op, the algorithm generates the update token tok and updates the state δ. tok is initialized to be an empty map. Hereafter we denote the attribute map associated with C as $\mathbf{cm_a}$, which is outdated when the local cache is not empty. Specifically, if $op = add$, the algorithm computes (ℓ, v) from (k_u, v_u) and the global counter cnt, and sets $tok[\ell] = v$. If $op = edit$, (k_u, v_u) is inserted to the cache T with **CInsert** algorithm, and T is evicted to C with **CEvict** when it is full. During **CInsert**, if k_u is already in T, it means $\mathbf{cm_a}[k_u] = 1 - T[\ell] = v_u$, where $\ell = F(k_1, k_u)$, because we assume all the updates are valid, *i.e.,* the new value must be different from the stale one. In this case, **CInsert** deletes $T[\ell]$; otherwise, it sets $T[\ell]$ to v_u. Recall that each value in C will be re-randomised with a string during an eviction. Such a string is derived from the encrypted key in C. So before running **CEvict**, we need obtain all the encrypted keys either from C or by re-encrypting all the keys of $\mathbf{m_a}$ (if they are accessible), and store them into S.

```
AUHME.Setup($1^\lambda$):
$k_1, k_2, k_3 \xleftarrow{\$} \{0,1\}^\lambda,\ cnt \leftarrow 0,\ S \leftarrow \perp$
$T \leftarrow$ empty map
Choose $\zeta$ as the capacity for $T$
$msk \leftarrow (k_1, k_2, k_3),\ \delta \leftarrow (cnt, T, \zeta, S)$
return $(msk, \delta)$

AUHME.Enc($msk, \mathbf{m_a}$):
$C \leftarrow$ empty map, $(k_1, k_2, k_3) \leftarrow msk$
for each pair $(k_a, v_a) \in \mathbf{m_a}$ do
    $\ell \leftarrow F(k_1, k_a)$
    $C[\ell] \leftarrow F(k_2, \ell || v_a) \oplus F(k_3, \ell || 0)$
end for
return $C$

AUHME.GenUpd($msk, \delta, op, k_u, v_u$):
$(k_1, k_2, k_3) \leftarrow msk,\ (cnt, T, \zeta, S) \leftarrow \delta$
$tok \leftarrow$ empty map
if $op = add$ then
    $\ell \leftarrow F(k_1, k_u)$
    $tok[\ell] \leftarrow F(k_2, \ell || v_u) \oplus F(k_3, \ell || cnt)$
    $UTok \leftarrow (add, tok)$
    return $(UTok, \delta)$
end if
                                        ▷ When $op = edit$
$T \leftarrow$ CInsert($msk, k_u, v_u, T$)
if $|T| + 1 < \zeta$ then
    $\delta \leftarrow (cnt, T, \zeta, \perp)$
    return $(\perp, \delta)$
else
    $S \leftarrow$ all the keys in $C$
    $tok \leftarrow$ CEvict($msk, cnt, S, T$)
    $T \leftarrow$ CClear($T$)
    $\delta \leftarrow (cnt + 1, T, \zeta, \perp)$
    $UTok \leftarrow (edit, tok)$
    return $(UTok, \delta)$
end if

AUHME.ApplyUpd($UTok, C$):
$(op, tok) \leftarrow UTok$
for each pair $(\ell, u) \in tok$ do
    if $op = add$ then
        $C[\ell] \leftarrow u$
    else if $op = edit$ then
        $C[\ell] \leftarrow C[\ell] \oplus u$
    end if
end for

AUHME.GenKey($msk, \delta, \mathbf{m_p}$):
$(k_1, k_2, k_3) \leftarrow msk,\ (cnt, T, -, -) \leftarrow \delta$
$L \leftarrow$ empty set, $xors \leftarrow 0^\lambda,\ \beta \leftarrow 1$
for each pair $(k_p, v_p) \in \mathbf{m_p}$ do
    $\ell \leftarrow F(k_1, k_p),\ L \leftarrow L \cup \{\ell\}$
    if CFind($msk, k_p, T$) $= 1 - v_p$ then
        $\beta \leftarrow 0$
    else if CFind($msk, k_p, T$) $= v_p$ then
        $v \leftarrow F(k_2, \ell || (1 - v_p)) \oplus F(k_3, \ell || cnt)$
        $xors \leftarrow xors \oplus v$
    else if CFind($msk, k_p, T$) $= \perp$ then
        $v \leftarrow F(k_2, \ell || v_p) \oplus F(k_3, \ell || cnt)$
        $xors \leftarrow xors \oplus v$
    end if
end for
if $\beta = 1$ then
    $r \xleftarrow{\$} \{0,1\}^\lambda,\ d \leftarrow H(r || xors)$
else
    $r \xleftarrow{\$} \{0,1\}^\lambda, d \xleftarrow{\$} \{0,1\}^\lambda$
end if
return $dk = (L, r, d)$

AUHME.Query($dk, C$):
$(L, r, d) \leftarrow dk,\ xors' \leftarrow 0^\lambda$
for each $\ell \in L$ do
    $xors' \leftarrow xors' \oplus C[\ell]$
end for
$d' \leftarrow H(r || xors')$
if $d' = d$ then
    return 1
else
    return 0
end if
```

Figure 2: Our AUHME Construction

```
CFind($msk, k, T$):
$(k_1, -, -) \leftarrow msk$
$\ell \leftarrow F(k_1, k)$
if $T[\ell]$ exists then
    return $T[\ell]$
else
    return $\perp$
end if

CInsert($msk, k, v, T$):
$(k_1, -, -) \leftarrow msk$
$\ell \leftarrow F(k_1, k)$
if $T[\ell]$ exists then
                        ▷ This means $\mathbf{cm_a}[k] = v$.
    Delete $T[\ell]$
else
    $T[\ell] \leftarrow v$
end if
return $T$

CEvict($msk, cnt, S, T$)
$tok \leftarrow$ empty map
for each $\ell \in S$ do
    if $T[\ell]$ does not exist then
                                ▷ No update for $\ell$
        $u \leftarrow F(k_3, \ell || cnt) \oplus$
                        $F(k_3, \ell || cnt + 1)$
    else            ▷ There is an update for $\ell$
        $b \leftarrow T[\ell]$
        $u \leftarrow F(k_2, \ell || b) \oplus$
    $F(k_2, \ell || (1 - b)) \oplus F(k_3, \ell || cnt) \oplus$
    $F(k_3, \ell || (cnt + 1))$
    end if
    $tok[\ell] \leftarrow u$
end for
return $tok$

CClear($T$)
Delete all entries of $T$
return $T$
```

Figure 3: Cache Algorithms Used in AUHME Construction

CEvict traverses through each $\ell \in S$ and generates the string u for updating $C[\ell]$ according to whether $T[\ell]$ exists. Assuming $\ell = F(k_1, k_a)$, if $T[\ell]$ does not exist, meaning no update on (k_a, v_a) is cached, $C[\ell]$ should be updated without modifying v_a. We achieve that by generating u that will only increase cnt by 1. If $T[\ell]$ exists, it implies that $\mathbf{cm_a}[k_a] = 1 - T[\ell]$, and we produce u that will update v_a to $T[\ell]$ and increase cnt.

When **CEvict** is done, **CClear** is called to clear T. Finally, the algorithm returns $(UTok, \delta)$, where $\delta = (cnt + 1, T, \zeta, \perp)$.

- **AUHME.ApplyUpd**($UTok, C$) : This algorithm updates the ciphertext C with $UTok = (op, tok)$. In the case that $op = add$, C is updated to the union of C and tok. If $op = edit$, which is for an eviction, the algorithm computes $C[\ell] = C[\ell] \oplus tok[\ell]$ for each key ℓ in tok.
- **AUHME.GenKey**($msk, \delta, \mathbf{m_p}$) : It generates the decryption key $dk = (L, r, d)$, *i.e.,* the token for querying if $\mathbf{m_p}$ is a subset of $\mathbf{m_a}$. Since the most recent editing updates are cached locally, the values stored in C may be out of date. Thus, for a pair $(k_p, v_p) \in \mathbf{m_p}$ and $\ell = F(k_1, k_p)$, we have 3 cases to process: 1) an update for k_p is cached in T but v_p does not match the cached value, *i.e.,* $T[\ell] \neq \perp$ and $T[\ell] \neq v_p$; 2) an update for k_p is cached in T and v_p matches the cached value, *i.e.,* $T[\ell] = v_p$; and 3) no update for k_p is cached in T, *i.e.,* $T[\ell] = \perp$. The first case indicates $\mathbf{m_p}$ is not a subset of the latest $\mathbf{m_a}$ for sure, and if all the pairs in $\mathbf{m_p}$ are in the second case, $\mathbf{m_p}$ must be a subset of $\mathbf{m_a}$. However, to avoid leaking information about updates, we generate dk and perform the query for all cases.
  L and r are generated in the same way, yet d is generated in different ways for the three cases. Specifically, L stores $\ell = F(k_1, k_p)$ for every $k_p \in \mathbf{m_p}$. r is a random string. For the first case, d is also a random string as we already know the query result. For the last two cases, d is $H(r||xors)$, where $xors$ is generated by XORing all the encrypted pairs of $\mathbf{m_p}$. In particular, in the second case (*i.e.,* when $T[\ell] = v_p$), $C[\ell]$'s plaintext must be $1 - v_p$; otherwise the update in T is invalid, which is forbidden. To ensure the correctness of the query, we encrypt $(k_p, 1 - v_p)$ for such pairs.
- **AUHME.Query**(dk, C) : It first parses dk to (L, r, d). Then it XORs every value in C whose key belongs to L and obtains the result $xors'$. If $H(r||xors') = d$, which demonstrates that $xors' = xors$, the algorithm outputs 1, otherwise it outputs 0.

```
Run $(K_T, s_T, \mathrm{TMap}) \leftarrow$
                    $\mathrm{RHS.Setup}(1^\lambda, \mathrm{DB})$

Client:
DB$'$ ← empty map
for each $w \in \mathrm{W}$ do
    for each $id \in \mathrm{DB}(w)$ do
        $\mathrm{DB}'[w||id] \leftarrow 1$
    end for
    for each $id' \in \mathrm{ID} \setminus \mathrm{DB}(w)$ do
        $\mathrm{DB}'[w||id'] \leftarrow 0$
    end for
end for
Randomly permute the entries of DB$'$
$(msk, \delta) \leftarrow \mathrm{HME.Setup}(1^\lambda)$
$\mathrm{XMap} \leftarrow \mathrm{HME.Enc}(msk, \mathrm{DB}')$
$K \leftarrow (K_T, msk)$
$s \leftarrow (s_T, \delta)$
Send XMap to the server
return $(K, s)$

Server:
return EDB=(TMap, XMap)
```

Figure 4: HDXT.Setup(1^λ, DB)

3.3 Complexities of AUHME

Encryption. Each pair in the map is encrypted with F, thus AUHME.Enc causes $O(|\mathbf{m_a}|)$ computational complexity. The storage overhead added by C is also $O(|\mathbf{m_a}|)$.

Query. AUHME.GenKey generates dk through traversing each pair in $\mathbf{m_p}$, resulting in $O(|\mathbf{m_p}|)$ computational overhead and token size. AUHME.Query procedure only processes each entry of C whose key is in L, also causing $O(|\mathbf{m_p}|)$ computational cost.

Update. To add a pair (k_u, v_u), AUHME.GenUpd derives two strings, resulting in $O(1)$ computational overhead and token size. When $op = edit$, if the cache is not full, $O(1)$ computational cost is paid to cache (k_u, v_u) and the token size is zero, otherwise an eviction happens. An eviction pseudorandomly derives $|\mathbf{m_a}|$ strings, which incurs $O(|\mathbf{m_a}|)$ computational overhead and token size. On average, the editing overhead amortized to each pair is $O(|\mathbf{m_a}|/\zeta)$. For AUHME.ApplyUpd, the number of processed pairs is equal to the token size. Therefore, the incurred overhead is $O(1)$ for an addition and $O(|\mathbf{m_a}|/\zeta)$ for an edit operation.

3.4 Security of AUHME

To capture the query leakage, we first define a vector $\mathbb{K}$. Initially, each key in $\mathbf{m_a}$ is inserted into $\mathbb{K}$ in sequence. When an addition involving (k_u, v_u) comes, k_u is inserted into $\mathbb{K}$. Then we define a function $\mathrm{Loc}(\mathbf{m_p})$ that outputs the *key pattern* about a predicate map $\mathbf{m_p}$. Formally, $\mathrm{Loc}(\mathbf{m_p})$ outputs a vector $\mathbf{v}$ that satisfies for all $1 \leq i \leq |\mathbf{m_p}|$

$$\mathbf{v}[i] = \begin{cases} j & \exists j : \mathbb{K}[j] = \text{ the } i\text{-th key in } \mathbf{m_p} \\ \perp & \text{otherwise} \end{cases}$$

We allow $\mathcal{L}_q^h(\mathbf{m_p})$ include $\mathrm{Loc}(\mathbf{m_p})$ and $\phi_{\mathbf{m_p}}^{\mathrm{hme}}(\mathbf{m_a})$.

An addition reveals the operator *add*. An edit operation leaks nothing except for whether it incurs an eviction. We define a function $\mathrm{IfEvic}(k_u, v_u)$. If the edit operation $(edit, k_u, v_u)$ makes an eviction occur, $\mathrm{IfEvic}(k_u, v_u)$ outputs 1, otherwise it outputs nothing. Formally, $\mathcal{L}_u^h(op, k_u, v_u)$ is *add* when $op = add$, It only contains $\mathrm{IfEvic}(k_u, v_u)$ when $op = edit$. We have Theorem 3.1.

THEOREM 3.1. *If F is a secure PRF and H is modeled as a random oracle, our AUHME construction is $\mathcal{L}$-selectively-semantically secure.*

Proof: The proof is presented in Appendix A.

```
Run $(s_T; \mathrm{TMap}) \leftarrow \mathrm{RHS.Update}(K_T,$
$s_T, op, in; \mathrm{TMap})$, where the client updates $s_T$ and the server updates TMap

Client:
$(id, \mathrm{W}_{id}) \leftarrow in$
if $op = add$ then
    $UT$ ← empty map
    for each $w$ in W do
        if $w \in \mathrm{W}_{id}$ then
            $(UTok, \delta) \leftarrow$ HME.
    GenUpd($msk, \delta, add, (w||id, 1)$)
        else
            $(UTok, \delta) \leftarrow$ HME.
    GenUpd($msk, \delta, add, (w||id, 0)$)
        end if
        $(add, ut) \leftarrow UTok$
        $UT \leftarrow UT \cup ut$
    end for
    Randomly permute the entries of $UT$
    $tok_x \leftarrow (add, UT)$
    Send $tok_x$ to the server
    $\mathrm{ID} \leftarrow \mathrm{ID} \cup \{id\}$
else if $op = edit^+/edit^-$ then
    for each $w \in \mathrm{W}_{id}$ do
        $(tok_x, \delta) \leftarrow$ EditPair($msk, \delta,$
                                        $op, id, w$)
        if $tok_x \neq \perp$ then
            Send $tok_x$ to the server
        end if
    end for
end if
return $s = (s_T, \delta)$

Server:
XMap ←HME.ApplyUpd(
                                $tok_x$, XMap)
return EDB = (TMap, XMap)
```

```
EditPair($msk, \delta, op, id, w$)
$(cnt, T, |\mathrm{W}|, \perp) \leftarrow \delta$
if $|T| + 1 \geq |\mathrm{W}|$ then
    $S$ ← empty set
    for each $w' \in \mathrm{W}$ do
        for each $id' \in \mathrm{ID}$ do
            $\ell \leftarrow F(k_1, w'||id')$
            $S \leftarrow S \cup \{\ell\}$
        end for
    end for
    $\delta \leftarrow (cnt, T, |\mathrm{W}|, S)$
end if
if $op = edit^+$ then
    $(tok_x, \delta) \leftarrow$ HME.GenUpd(
                $msk, \delta, edit, (w||id, 1)$)
else if $op = edit^-$ then
    $(tok_x, \delta) \leftarrow$ HME.GenUpd(
                $msk, \delta, edit, (w||id, 0)$)
end if
return $(tok_x, \delta)$
```

Figure 5: HDXT.Update(K, s, op, in)

```
Run $\mathrm{DB}(w_1) \leftarrow \mathrm{RHS.Search}($
$K_T, s_T, w_1; \mathrm{TMap})$, where the client receives $\mathrm{DB}(w_1)$.

Client:
$R_1, DK$ ← empty lists
Insert $\mathrm{DB}(w_1)$ into $R_1$ and Randomly permute the entries of $R_1$
for $j = 1$ to $|R_1|$ do
    $id \leftarrow R_1[j]$, $I_j$ ← empty map
    for $i = 2$ to $n$ do
        $I_j[w_i||id] \leftarrow 1$
    end for
    $dk_j \leftarrow \mathrm{HME.GenKey}(msk, \delta, I_j)$
    $DK \leftarrow DK \cup \{dk_j\}$
end for
Send $DK$ to the server

Server:
$Pos$ ← empty set
for $j = 1$ to $|DK|$ do
    $r \leftarrow \mathrm{HME.Query}(DK[j], \mathrm{XMap})$
    if $r = 1$ then
        $Pos \leftarrow Pos \cup \{j\}$
    end if
end for
Send $Pos$ to the client

Client:
$R$ ← empty set
for each $j \in Pos$ do
    $R \leftarrow R \cup \{R_1[j]\}$
end for
return $R$
```

Figure 6: HDXT.Search($K, s, w_1 \wedge \cdots \wedge w_n$)

4 HDXT- OUR CONJUNCTIVE DSSE SCHEME

This section presents our conjunctive DSSE construction: HDXT.

4.1 Overview of HDXT

In HDXT, the encrypted index consists of TMap and XMap. TMap is a structure produced by a response-hiding single-keyword DSSE scheme (denoted as RHS). Initially, XMap is obtained by using predicate-only AUHME to encrypt the extended database DB$'$ defined in Section 2.3. Within a conjunction $w_1 \wedge \cdots \wedge w_n$, the client first makes a single-keyword query on w_1 with TMap to obtain $\mathrm{DB}(w_1)$. Then for each $id \in \mathrm{DB}(w_1)$, it builds a predicate map I that stores a mapping from $w_i||id$ to 1 for $2 \leq i \leq n$ and issues an AUHME query to check if I is a subset of DB$'$. If the AUHME query returns 1, id matches the conjunction. The security of AUHME guarantees that the server cannot learn $\mathrm{DB}'[w_i||id]$ for all $2 \leq i \leq n$ if the AUHME query returns 0. Thus, KPRP-hiding can be ensured.

To update a keyword-document pair, TMap is trivially updated with RHS, and AUHME enables DB' to be updatable. As we have described in Section 3, to achieve secure edit operations, AUHME preserves a local cache of fixed size. For HDXT, the cache is kept by the client, and the cache capacity is set to $|\mathrm{W}|$. Note that the incurred client storage is comparable to many SSE schemes [7, 29, 30, 43].

Following the mainstream SSE, we assume there is an authentication scheme in place that enables the client and the server to verify each other's identities before exchanging any data. This can be implemented with the transport layer security (TLS) protocol, two-factor authentication [47, 49], or human-memorizable password-based authentication [14]. In addition, In line with [25, 39, 50], we prohibit incorrect updates introduced in [53].

4.2 Details of HDXT

Fig.4, Fig.5, and Fig.6 show the pseudocodes for HDXT. RHS is adopted in a black-box way, and AUHME is abbreviated as HME.

- $(K, s; \mathrm{EDB}) \leftarrow$ **HDXT.Setup**$(\lambda, \mathrm{DB}; \perp)$: The setup phase generates EDB = (TMap, XMap) from DB, with RHS and AUHME.

- $(s; \mathrm{EDB}) \leftarrow$ **HDXT.Update**$(K, s, op, in; \mathrm{EDB})$: Within an update, RHS is executed to update TMap. The update token for XMap is a map tok_x.
  When $op = add$, $\{(w||id, 1)|w \in \mathrm{W}_{id}\} \cup \{(w||id, 0)|w \in \mathrm{W} \setminus \mathrm{W}_{id}\}$ should join DB'. As shown in Line 3 - 17 (Fig.5), the client generates an AUHME addition token $UTok$ for each pair and then merges these addition tokens into tok_x.
  In the case that $op = edit^+/edit^-$, $\mathrm{DB}'[w||id]$ should be changed to 1 ($op = edit^+$) or 0 ($op = edit^-$) for each $w \in \mathrm{W}_{id}$. As presented in line 18 - 24 (Fig.5), for each $w \in \mathrm{W}_{id}$, the client calls **EditPair** (msk, δ, op, id, w) to generate tok_x, which is either empty or an eviction token.
  In **EditPair** (msk, δ, op, id, w), to make an eviction to be completed in one round, if the cache will overflow, the client computes all the keys in XMap and include them into the state of AUHME before calling HME.GenUpd.
  If $op = del$, XMap is unchanged. This will not affect subsequent searches, because the client will find that id was deleted during the related single-keyword searches on the *s-term*.

- $(\mathrm{DB}(w_1 \wedge con(w_2, \cdots, w_n)); \mathrm{EDB}) \leftarrow$ **HDXT.Search** $(K, s, w_1 \wedge \cdots \wedge w_n; \mathrm{EDB})$: Within a search on $w_1 \wedge w_2, \cdots, \wedge w_n$, HDXT first executes the search protocol of RHS, after which only the client gets $\mathrm{DB}(w_1)$. Then for each identifier $id \in \mathrm{DB}(w_1)$, it tests whether id satisfies $w_2 \wedge \cdots \wedge w_n$.
  Specifically, the client stores $\mathrm{DB}(w_1)$ into a list R_1 and randomly shuffles the elements of R_1. For $1 \leq j \leq |R_1|$, it takes id from $R_1[j]$ and builds a map $I_j = \{(w_i||id, 1)\}_{i=2}^{n}$. The client calls AUHME to generate the decryption key dk for I_j, which is then inserted into the j-th position of a list DK. DK is sent to the server. With $DK[j]$, the server calls AUHME to query whether I_j is a subset of DB'. If the AUHME query returns true, the server inserts j into a set Pos. Pos is then returned to the client. The final search result is $R = \{R_1[j]\}_{j \in Pos}$.

5 SECURITY AND PERFORMANCE ANALYSIS

In this section, we comprehensively analyze the security and performance achieved by HDXT.

5.1 Security of HDXT

To analyze the security of HDXT, we continue using the notions and functions introduced in Section 2.3. We denote the leakage function for RHS as $\mathcal{L}_{RHS}$, and also introduce the other four functions.

For a set Ids of document identifiers, $\mathrm{TimeIds}(Ids)$ outputs these identifiers and when each document was added. Formally, $\mathrm{TimeIds}(Ids) = \{(t, id)|id \in Ids \text{ and } \exists \mathrm{W}_{id} : (t, add, (id, \mathrm{W}_{id})) \in Q\} \cup \{(t^{\triangleright}, id)|id \in Ids \text{ and } id \text{ exists in } \mathrm{DB}^{\triangleright}\}$. Note that $\mathrm{TimeDB}(w)$ defined in Section 2.3 is equivalent to $\mathrm{TimeIds}(\mathrm{DB}(w))$.

$\mathrm{IP}(q)$ records the conditional intersection pattern with respect to a conjunction q. It is expressed as $(\mathrm{IP}(q[1], q[i]))_{i=2}^{n}$. For $2 \leq i \leq n$, if there exists a previous search q' that satisfies the following two conditions: 1) the j-th ($j \geq 2$) term is $q[i]$; 2) $\mathrm{DB}(q'[1]) \cap \mathrm{DB}(q[1]) \neq \emptyset$, $\mathrm{IP}(q[1], q[i])$ outputs the timestamp of q', j, and $\mathrm{TimeIds}(\mathrm{DB}(q'[1]) \cap \mathrm{DB}(q[1]))$. Formally, $\mathrm{IP}(q[1], q[i]) = \{(t, j, \mathrm{TimeIds}(\mathrm{DB}(q[1]) \cap \mathrm{DB}(q'[1]))|(t, q') \in Q \text{ and } q[i] = q'[j] \text{ and } \mathrm{DB}(q[1]) \cap \mathrm{DB}(q'[1]) \neq \emptyset\}$.

$\mathrm{AddTims}(q[1])$ outputs when the documents that belong to $\mathrm{DB}(q[1])$ were added to the database. Formally, $\mathrm{AddTims}(q[1]) = \{t|\exists id \in \mathrm{DB}(q[1]) \text{ and } \mathrm{W}_{id} : (t, add, (id, \mathrm{W}_{id})) \in Q\}$.

Based on Q and $|\mathrm{W}|$, the timestamps of the evictions can be obtained. If an eviction occurs within (op, in), $\mathrm{Evic}(op, in)$ outputs 1, otherwise it outputs nothing.

Within an update (op, in), updating TMap could expose $\mathcal{L}_{RHS}^{Upd}(\mathrm{DB}, op, in)$. When updating XMap, if $op = add$, the server could learn the operator add and $|\mathrm{W}|$, otherwise it learns $\mathrm{Evic}(op, in)$.

For a conjunction q, the single-keyword query on $q[1]$ reveals $\mathcal{L}_{RHS}^{Srch}(\mathrm{DB}, q[1])$. From the queries to XMap, the server could directly learn $|\mathrm{DB}(q[1])|$ through the number of the issued AUHME queries in q. Since an AUHME query could leak key pattern, it first could be linked to previous additions related to $\mathrm{DB}(q[1])$, which is captured by $\mathrm{AddTims}(q[1])$. Through the leaked key pattern, an AUHME query can also be associated with the previous conjunctions that have the same keys in predicate maps. The leakage caused by this association is no more than the information captured by $\mathrm{IP}(q)$. After each conjunction, the server could obtain $\mathrm{TimeIds}(\mathrm{DB}(q))$. Formally, we can get Theorem 5.1.

Theorem 5.1. *If* RHS *is* $\mathcal{L}_{RHS}$*-adaptively secure and AUHME is selectively-semantically secure as defined in Section 3, HDXT is* $\mathcal{L}_{HDXT}$*-adaptively secure where*

(1) $\mathcal{L}_{HDXT}^{Stp}(\mathrm{DB}) = (\mathcal{L}_{RHS}^{Stp}(\mathrm{DB}), |\mathrm{W}| \cdot |\mathrm{D}|)$

(2) $\mathcal{L}_{HDXT}^{Upd}(\mathrm{DB}, op, in) =$

$$\begin{cases} (\mathcal{L}_{RHS}^{Upd}(\mathrm{DB}, op, in), add, |\mathrm{W}|), & op = add \\ (\mathcal{L}_{RHS}^{Upd}(\mathrm{DB}, op, in), \mathrm{Evic}(op, in)) & op \neq add \end{cases}$$

(3) $\mathcal{L}_{HDXT}^{Srch}(\mathrm{DB}, q) = (\mathcal{L}_{RHS}^{Srch}(\mathrm{DB}, q[1]), \mathrm{TimeIds}(\mathrm{DB}(q)), |\mathrm{DB}(q[1])|, \mathrm{IP}(q), \mathrm{AddTims}(q[1]))$

Proof: The formal proof is presented in Appendix B.

KPRP-hiding & Forward Privacy. Theorem 5.1 demonstrates that the server cannot learn which identifiers belong to $\mathrm{DB}(q[i]) \cap$

$\mathrm{DB}(q[j])$ for any $1 \leq i < j \leq n$, except for $\mathrm{DB}(q)$. It demonstrates that HDXT successfully hides KPRP. The update leakage function clearly shows that HDXT inherits forward privacy from RHS.

Backward Privacy. The conditional intersection pattern $\mathrm{IP}(q)$ captures much less information than $\mathrm{TimeDB}(q)$; thus, $(\mathrm{TimeIds}(\mathrm{DB}(q)), |\mathrm{DB}(q[1])|, \mathrm{IP}(q), \mathrm{AddTims}(q[1])$ outputs less than $\mathrm{TimeDB}(q)$. Naturally, the level of backward privacy achieved by HDXT only depends on that of RHS. Particularly, if we use Mitra [13] to instantiate RHS, which reveals $(\pi_1^{\triangleright}, \mathrm{Updates}(q[1]))$ within a conjunction q and $|\mathrm{W}_{id}|$ during an update, HDXT realizes Type-II backward privacy. When RHS is instantiated with Orion [13] that only leaks $|\mathrm{DB}(q[1])|$ during q and $|\mathrm{W}_{id}|$ within an update, HDXT is Type-I backward private.

Mitigating Other Attacks. Existing attacks can be classified into: known-data/query attacks [5, 10, 23], inference attacks [21, 33, 34, 41], and injection attacks [40, 54]. For the first two types, the adversary is passive and requires an amount of auxiliary information, such as a subset of target databases/queries or a statistical distribution similar to the target databases/queries. For injection attacks, the adversary is active and capable of injecting a number of documents, without (or with quite less) auxiliary information.

Injection attacks [40, 54] are devastating for DSSE. HDXT mitigates the file-injection attack [54] by ensuring KPRP-hiding and forward privacy. Achieving forward privacy also helps to mitigate the injection attack [40] proposed by Poddar *et al.*. Their attack leverages the response length for search queries, whereas the adversary should be able to replay search queries after a round of updates independently. Forward private SSE updates the token of a search query after each related update, which makes the search unreplayable by anyone but the client.

Among the passive attacks, most of them [10, 21, 23, 41] exploit co-occurrence patterns, *i.e.,* the number of documents containing both w_i and w_j for any two queried keywords w_i and w_j. We claim that achieving KPRP-hiding is essential to prevent such attacks, otherwise KPRP directly exposes co-occurrence patterns. The other attacks demand explicit search patterns of single-keyword queries [33, 34] or volume patterns [5] that capture the number of keywords contained by the document that matches a query. To mitigate them, we can further reduce the leakages by instantiating the RHS of HDXT with search-pattern-hiding DSSE [18], which prevents RHS from revealing search and volume patterns. Furthermore, before making AUHME queries within a conjunction, the client can insert some randomly selected document identifiers into R_1 (Fig.6). This step adds noise into the leakages caused by queries over XMap. Besides, the client could issue searches on negated terms described in Section 6 to further perturb the above leakages.

5.2 Performance of HDXT

For clarity, we initialise RHS with Mitra [13] for performance analysis. In the following, unless otherwise specified, the overhead refers to the computational and communication overhead.

The setup phase generates TMap and XMap directly with RHS and AUHME, which results in $O(|\mathrm{DB}^{\triangleright}|)$ and $O(|\mathrm{W}||\mathrm{D}|)$ overheads, respectively. Within an update on $(op, (id, \mathrm{W}_{id}))$, HDXT uses RHS to update TMap, which costs $O(1)$ overhead for each keyword-document pair. To update XMap when $op = add$, HDXT invokes the addition procedure of AUHME $|\mathrm{W}|$ times, which causes $O(|\mathrm{W}|)$ total overhead and $O(|\mathrm{W}|/\mathrm{W}_d)$ average overhead per pair. When updating XMap during an edit query, an edit procedure of AUHME is invoked for every involved keyword-document pair. This edit process results in the same complexity as AUHME, which is $O(|\mathrm{W}||\mathrm{D}|/\zeta)$ as shown in Section 3.3. HDXT sets ζ to $|\mathrm{W}|$, hence the overhead amortized to each pair is $O(|\mathrm{D}|)$. Because a deletion only updates TMap, so its overhead is $O(1)$. For a conjunction q, RHS searches on $q[1]$ that brings $O(\pi_1)$ overhead. Then HDXT issues $|\mathrm{DB}(w_1)|$ AUHME queries. Each AUHME query is about a predicate map of size $n-1$, which incurs $O(n-1)$ overhead. The total overhead for a conjunction is $O(\pi_1 + n|\mathrm{DB}(q[1])|)$.

TMap and XMap cost $O(N)$ and $O(|\mathrm{W}||\mathrm{D}|)$ server storage overheads, respectively. For the client storage, RHS causes $O(|\mathrm{W}| \log |\mathrm{D}|)$ overhead. The client also requires $O(|\mathrm{W}|\lambda)$ bits to keep the local cache. The total client storage is $O(|\mathrm{W}|(\log |\mathrm{D}| + \lambda))$. Note that the eviction procedure in HDXT could be processed in a streaming manner to avoid excessive consumption of client storage.

Performance Comparison with Previous Work. Table 1 shows that HDXT outperforms FBDSSE-CQ [57] in every respect, especially search and storage efficiency. Compared with other schemes [25, 31, 35, 39, 50] that have weaker security, HDXT achieves very competitive search efficiency and might be less efficient in editing and storage efficiency.

We claim that the less efficient editing efficiency is a price HDXT pays for small leakages. In Section 6, we describe an extension of HDXT (called HDXT_{SU}), which achieves much better editing efficiency at the cost of increasing the leakage. Note that HDXT_{SU} still guarantees KPRP-hiding and forward privacy.

The server storage of HDXT is higher than the KPRP-hiding static solution HXT [31]. In HXT, the essential plaintext index structure is a Bloom filter [6] built from the database, which makes its server storage smaller than ours. However, Bloom filter is not friendly for updates. DB' adopted by HDXT enables secure updates while preserving efficient KPRP-hiding searches, at the cost of larger size. In the current literature, it is common to achieve better security or functionality at the expense of increasing server storage as the storage is getting much cheaper. For instance, compared with OXT[12], IEX [25] and CNFFilter [37] support disjunctive searches sub-linearly, yet they need much higher server storage than OXT.

5.3 Cache and Eviction Strategy

HDXT needs a cache T on the client to process updates. Its size ζ only affects the amortized edit complexity and has no impact on security and search performance; thus, it can be configured based on the storage capacity of the client.

Specifically, T is only used within edit and search queries. For an edit query on a keyword-document pair, it is either inserted into the cache or evicted to XMap with all the cached updates. An eviction is oblivious and reveals nothing. For performance, Section 5.2 shows that the amortized edit complexity is inversely proportional to ζ. During a search, the client uses the cache in the second round. To test whether an identifier $id \in \mathrm{DB}(w_1)$ matches the remaining keywords, the client issues an AUHME query that first accesses the cache $(n-1)$ times and then generates a decryption key $dk = (L, r, d)$. L remains constant for the same predicate map, r

is randomly generated, and d is also random from the perspective of the server. Therefore, neither the cached content nor the capacity has any impact on the search performance and security. In practice, the client could evict the cache to XMap in any state, such as when the server is idle, as long as the client storage is affordable.

6 EXTENSION

In this section, we first briefly describe HDXT_{SU} and then discuss how HDXT supports queries involving negated terms.

Performance Enhanced Update. HDXT updates all the entries of XMap in an eviction, which achieves high-level security guarantees but is costly. HDXT_{SU} improves the update performance by reducing the pairs to be updated in the eviction. Basically, HDXT_{SU} only updates the entries related to the documents that were edited since the last eviction (or the setup if no eviction happened). In this case, the server could learn which documents were edited since the last eviction. It cannot infer which keywords were updated, so forward privacy is still guaranteed.

HDXT_{SU} creates XMap in a slightly different way. In the setup phase, instead of building DB' for all the documents, the client builds ${\text{DB}'}_{id} = \{(w, b) | w \in \text{W}\}$ for each $id \in \text{D}$ separately, where b is 1 if $id \in \text{DB}(w)$, otherwise $b = 0$. The collection of all the encrypted ${\text{DB}'}_{id}$ is the XMap of HDXT_{SU}. For search queries, after obtaining $\text{DB}(w_1)$, the client builds $I' = \{(w_i, 1)\}_{i=2}^{n}$ and queries whether $I' \subseteq {\text{DB}'}_{id}$ for each $id \in \text{DB}(w_1)$. Within an eviction, for every document id that has at least one related edit operation in the local cache, HDXT_{SU} evicts the edit operations associated with id to ${\text{DB}'}_{id}$. By doing so, the overhead caused by an eviction is only linear with $|\text{W}| \cdot t$, where t is the number of edited documents since the last eviction. The amortized edit complexity can be reduced to $O(t)$. We present the detailed HDXT_{SU} in Appendix D.

Conjunctions on Negated Terms. HDXT can be trivially extended to support conjunctions on negated terms. A negated term aims to return the documents that do not contain the given keyword. Given a conjunction on negated terms (*e.g.*, $w_1 \wedge \neg w_2 \wedge \neg w_3$), RHS is first invoked to search for $\text{DB}(w_1)$. After that, if w_1 is a non-negated term (negated term), the client inserts $\text{DB}(w_1)$ $(\text{ID} \backslash \text{DB}(w_1))$ to a list R_1. Then for every $id \in R_1$, it constructs the predicate map $I = \{[w_i || id, b_i]\}_{i=2}^{n}$, where if w_i is a non-negated term, b_i is set to 1, otherwise it is 0. The client launches an AUHME query to check whether I is a subset of DB' as in Fig.6. If the query returns 1, id matches the conjunction. Note that HXT [31] cannot support conjunctions with negated terms, mainly due to its index structure.

7 PERFORMANCE EVALUATION

We implement a prototype of HDXT and compare its performance with the state-of-art conjunctive SSE schemes with KPRP-hiding.

7.1 Experiment Setting

Baselines. There currently exist four KPRP-hiding conjunctive SSE solutions: the naive solution shown in Section 1, Blind Seer [35], HXT [31], and FBDSSE-CQ [57]. Table 1 shows that HDXT is more efficient than Blind Seer and FBDSSE-CQ for search queries. This is because Blind Seer heavily relies on expensive secure two-party computation and requires non-constant rounds of client-server interactions, and FBDSSE-CQ incurs linear overheads for a search query. In this section, we compare the search performance of HDXT with the naive solution and HXT, and the update performance with the naive solution and FBDSSE-CQ. As done in [39], we use MITRA [13] to instantiate the naive solution, called $\text{MITRA}_{\text{CONJ}}$. MITRA is also used to instantiate RHS used in HDXT.

We use $\text{MITRA}_{\text{CONJ}}$ as one baseline to present the performance of HDXT. But note that, as described in [39], $\text{MITRA}_{\text{CONJ}}$ has serious leakages: the number of updates related to every keyword involved in a conjunction and the repetition of every searched keyword.

Implementation. We implement a prototype [52] for $\text{MITRA}_{\text{CONJ}}$, HXT, FBDSSE-CQ, HDXT, and HDXT_{SU} with C++. The cryptographic primitives are implemented based on Crypto++ library [16]. In particular, we use AES-ECB-128 + SHA-256 for pseudorandom functions, SHA-256 for hash functions, the C++ Bloom filter library [36] for the Bloom filter used in HXT, and the elliptic curve secp256r1 for group operations in HXT. RocksDB [17] is deployed for the storage on the client and the server. gRPC [20] is adopted for communication between the client and the server.

Test-bed. We use two machines to conduct the experiments. Both machines run Ubuntu 18.04 LTS: the first machine has 16× Intel Core Processor (Broadwell, IBRS 2.15GHz), 64GB RAM, and 4TB hard disk drives; the second has 16 cores (Intel Core i9-9900 CPU 3.10 GHz), 31GB RAM, and 483 GB SSD disk space. The experiments are executed in the network setting, where the first machine runs as the server and the second plays as the client.

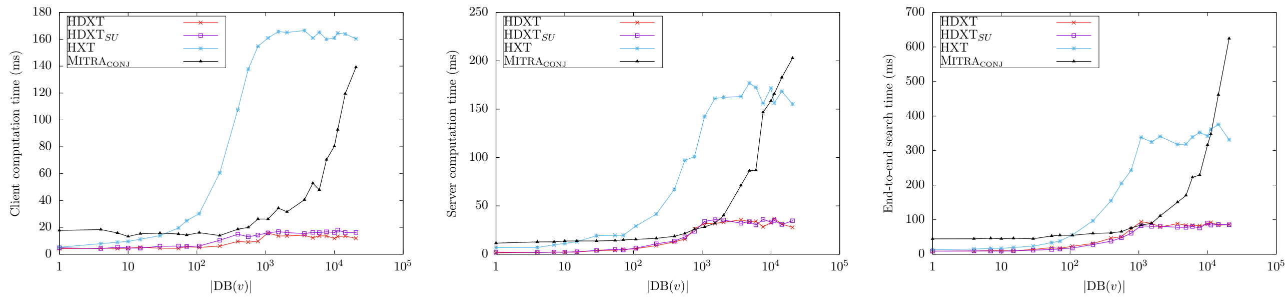
Dataset. We extract two datasets from Wikimedia [1]. The first dataset contains 23643 documents, 60879 keywords, and 8373977 keyword-document pairs. The second one comprises 86386 documents, 188096 keywords, and 27850059 keyword-document pairs.

7.2 Search Performance in Static Database

We first measure the search performance of our solutions for the static database. For this experiment, we take the first dataset as input, set up the encrypted database, and perform a series of conjunctive queries. We measure the time cost by the client and the server and the end-end search latency for each search query. Meanwhile, we measure the costed communication overheads.

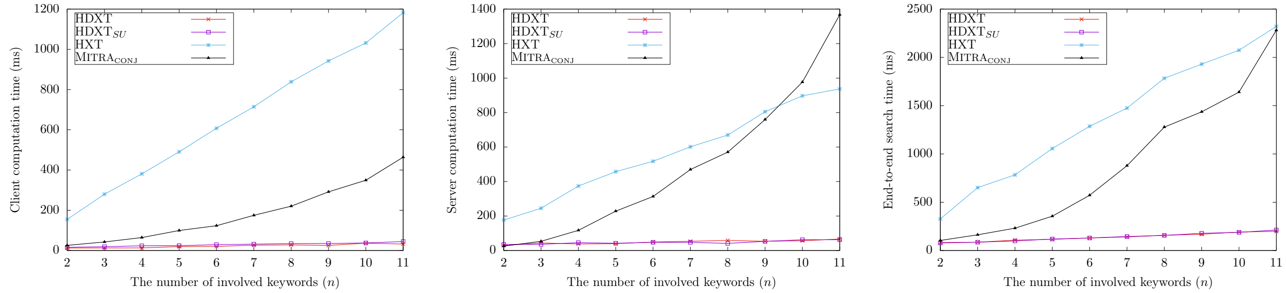
7.2.1 2-Conjunctions.

We start by testing the performance of conjunctions involving two keywords. We choose two terms v and a. The term v is variable with $|\text{DB}(v)|$ ranging from 1 to 20604. $|\text{DB}(a)|$ is fixed to 1096. When $|\text{DB}(v)| \leq \text{DB}(a)$, we perform the conjunction $v \wedge a$. $a \wedge v$ is searched when $|\text{DB}(v)| > \text{DB}(a)$. For our solutions, this can be easily done by checking the local counters produced by MITRA [13]. The search time for these conjunctions is described in Fig.7, and the communication overheads are presented in Fig.9(a). For HDXT, HDXT_{SU}, and HXT, when $|\text{DB}(v)| < 10^3$, the search time for $v \wedge a$ rises as $|\text{DB}(v)|$ increases, and the efficiency for $a \wedge v$ remains almost constant when $|\text{DB}(v)| > 10^3$. This result is consistent with the asymptotic complexity given in Table 1. For 2-conjunctions, the efficiency for the three schemes is only proportional to the number of documents matched by the *s-term*. In contrast, the search time for $\text{MITRA}_{\text{CONJ}}$ is linear with $|\text{DB}(v)| + |\text{DB}(a)|$. The results show that the 2-conjunctions in our solutions outperform $\text{MITRA}_{\text{CONJ}}$ and HXT in the respects of computational and communication overheads. Note that HXT costs



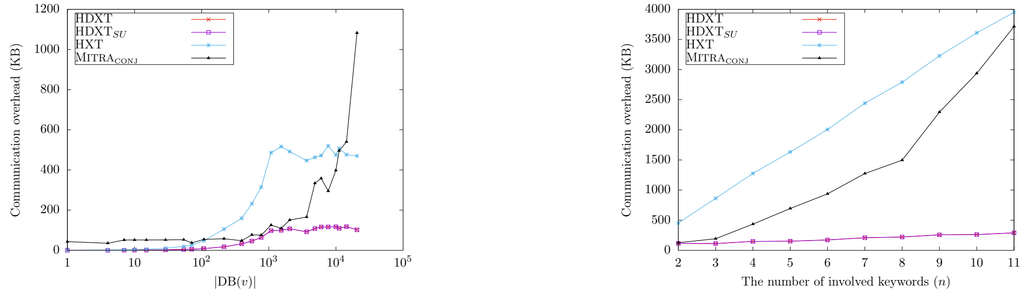
(a) Client (b) Server (c) End-to-end

Figure 7: Search Time of 2-Conjunctions



(a) Client (b) Server (c) End-to-end

Figure 8: Search Time of n-Conjunctions



(a) 2-Conjunctions (b) n-Conjunctions

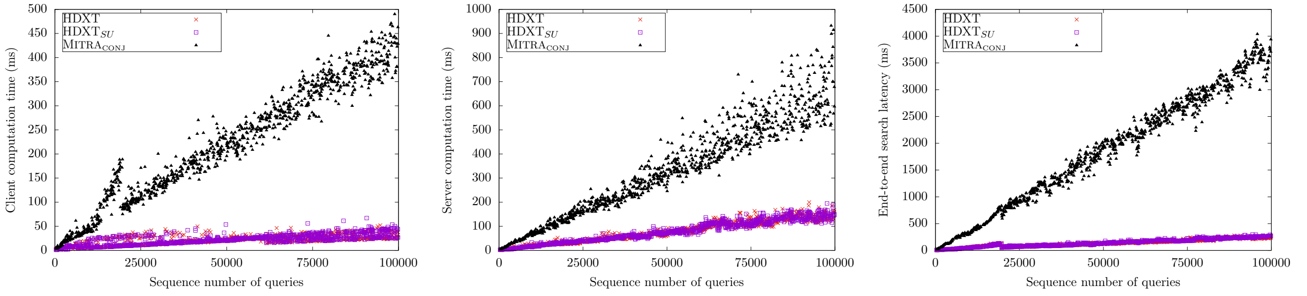
Figure 9: Communication Overheads of Conjunctions

relatively larger computation time because it involves a number of exponential operations for each search.

7.2.2 n-Conjunctions. We also test the performance for conjunctions of n ($2 \leq n \leq 11$) keywords. Here a conjunction is expressed as $a \wedge v_1 \wedge \cdots \wedge v_{n-1}$, where a is the *s-term*. Fig.8 presents the search time, and Fig.9(b) gives the communication overheads. The two figures clearly show that n-conjunctions in our two solutions perform better than MITRA$_{\text{CONJ}}$ and HXT in every respect. Moreover, we can see that the performance gap between HDXT and HXT and the gap between HDXT and MITRA$_{\text{CONJ}}$ become larger as n increases. In particular, when $n = 11$, HDXT is 10.7× and 10.5× faster than HXT and MITRA$_{\text{CONJ}}$, respectively. The communication overhead is 12.7× and 9.2× better than HXT and MITRA$_{\text{CONJ}}$, respectively.

7.3 Search Performance in Dynamic Database

This sub-section tests the search performance in the dynamic database for the dynamic solutions: HDXT, HDXT$_{SU}$, and MITRA$_{\text{CONJ}}$. Here we generate a sequence of queries that involve ten keywords $w_1, \cdots, w_{10}$. 99% of them are update queries, and 1% of them are conjunctions of the ten keywords. Among the update queries, 2%



(a) Client (b) Server (c) End-to-end

Figure 10: Search Time in Dynamic Database

of them edit pairs related to w_1, 10% edit the pairs related to w_i for $2 \le i \le 10$, and 1% delete pairs related to w_i for $4 \le i \le 10$.

Fig.10 shows the search time spent by every conjunction. We can see that the search performance of our two solutions is significantly better than MITRA$_{\text{CONJ}}$. For instance, the end-to-end search latency in our solutions is 13× better than that in MITRA$_{\text{CONJ}}$.

7.4 Update Performance

We use the sequence of update queries generated in Section 7.3 to evaluate the update performance. Specifically, we test the time cost by editing a keyword-document pair, the results of which are shown in Fig.11(a). Meanwhile, we compute the amortized update time per pair by first taking the total time it takes to update an increasing number of pairs and then dividing the obtained time by the number of updated pairs. Fig.11(b) shows the result. Fig.11(a) demonstrates that the update efficiency of HDXT and HDXT$_{SU}$ is close to that of MITRA$_{\text{CONJ}}$ and much better than FBDSSE-CQ, except for the query that incurs an eviction. HDXT and HDXT$_{SU}$ spend 5.8 and 1.6 hours for an eviction, respectively. Regarding the amortized efficiency, as shown in Fig.11(b), HDXT and HDXT$_{SU}$ are 8.2× and 32× better than that of FBDSSE-CQ, respectively. The amortized update performance of the three schemes is much weaker than MITRA$_{\text{CONJ}}$. Nevertheless, MITRA$_{\text{CONJ}}$ achieves quick updates at the cost of search efficiency and security.

7.5 Storage

We test the storage overheads for HDXT, HDXT$_{SU}$, HXT, MITRA$_{\text{CONJ}}$, and FBDSSE-CQ [57]. In the experiment for the server storage, to demonstrate that the server storage caused by our schemes is acceptable, we also test several other schemes proposed in recent years, including DIEX [25], IBTree [32], and CNFFilter [37]. Note that DIEX, IBTree, and CNFFilter do not achieve KPRP-hiding.

7.5.1 *Server Storage.*

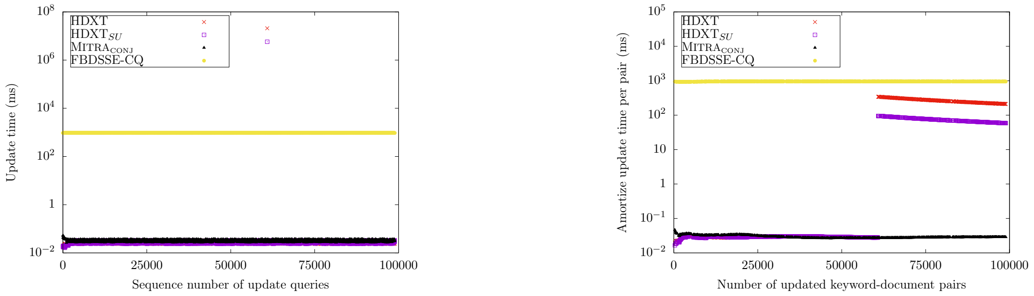
We encrypt the two datasets with the five schemes and show their storage overhead in Table 2. From the table, we can see that although the server storage required by HDXT and HDXT$_{SU}$ is larger than that needed by HXT and MITRA$_{\text{CONJ}}$, it is less than or comparable to some previous conjunctive SSE schemes. This is because there exists a trade-off between security, performance, and functionality for the design on conjunctive SSE. In order to improve security or functionality without sacrificing search efficiency, increasing the server storage moderately becomes a choice considering that the storage is becoming much cheaper.

7.5.2 *Client Storage.*

For static SSE, the client only keeps the secret keys, which commonly puts $O(1)$ storage overhead on the client. However, for DSSE, the client needs to store a state s to support updates securely. So here we just test the client storage required by the dynamic KPRP-hiding and forward secure solutions, which include HDXT, HDXT$_{SU}$, MITRA$_{\text{CONJ}}$[39], and FBDSSE-CQ. For MITRA$_{\text{CONJ}}$[39] and FBDSSE-CQ, we measure the size of the RocksDB database on the client after creating the encrypted database with the above datasets. Considering that the client keeps a cache in our two solutions, we generate an update sequence for HDXT and HDXT$_{SU}$ to fill the cache, before measuring their client storage. Table 3 presents the results. HDXT needs less client storage than FBDSSE-CQ. HDXT$_{SU}$ requires 13% more client storage space than FBDSSE-CQ. Note that the size of the client storage required by FBDSSE-CQ is the same as many previous forward secure SSE schemes, such as [7, 29, 30, 43].

8 RELATED WORK

SSE was first introduced by Song *et al.* [42] in 2000. It is a technique that allows slight leakages (such as search and access patterns) to ensure practicability. This motivates the research on leakage-abuse attacks [5, 10, 21, 23, 33, 34, 40, 41, 54] that exploit leakages to undermine security guarantees. In response, some literature develops leakage-suppression techniques [2, 13, 18, 22, 26, 27, 38, 46] to counteract the above attacks. However, these techniques have rather high overheads and focus on single-keyword searches. For example, Hoang *et al.* [22] hide response length for single-keyword queries, but their search complexity scales linearly with |D|. Chamani *et al.* [13] leveraged Path-ORAM to achieve ORION, which only reveals the response length. Nevertheless, ORAM brings impractical overhead and $O(\log N)$ rounds of interactions. The search performance could be improved by replacing Path-ORAM with more efficient ones, such as Root ORAM [46] that pays the price of reducing security to the level of differential privacy. However, this solution still suffers from $O(\log N)$ round complexity. As described in Section 1, these single-keyword schemes can be extended to securely process conjunctions, but their performance will become more unacceptable. As shown in Section 5, HDXT proposed in this paper achieves



(a) Single-update Time

(b) Amortized Update Time

Figure 11: Update Performance

Table 2: Comparison of Server Storage

$\|W\|$	$\|D\|$	N	Schemes							
			HDXT	HDXT_{SU}	HXT[31]	$\text{MITRA}_{\text{CONJ}}$[39]	DIEX[25]	IBTree[32]	CNFFilter[37]	FBDSSE-CQ[57]
6×10^4	2.3×10^4	8.4×10^6	44G	44G	8.6G	237M	187G	22G	258G	623G
1.9×10^5	8.6×10^4	2.8×10^7	498G	498G	28G	819M	692G	243G	1T	3.3T

Table 3: Comparison of Client Storage

$\|W\|$	$\|D\|$	N	Schemes			
			HDXT	HDXT_{SU}	$\text{MITRA}_{\text{CONJ}}$[39]	FBDSSE-CQ
6×10^4	2.3×10^4	8.4×10^6	1.6M	2.5M	1M	2.2M
1.9×10^5	8.6×10^4	2.8×10^7	4.2M	6.9M	2.4M	6.1M

a desirable trade-off between search efficiency and security. In the following, we review the existing conjunctive DSSE and mainly concern three crucial security properties: KPRP-hiding, forward privacy, and backward privacy.

Golle *et al.* [19] proposed the first conjunctive SSE scheme in 2004. Their scheme was extended in [3, 9] for better performance. However, they all suffer from linear search complexity.

In 2013, Cash *et al.* [12] achieved a nice trade-off between security and efficiency by designing OXT, yet OXT leaks KPRP. Afterward, HXT [31], BDXT [39], and ODXT [39] were proposed based on OXT. HXT [31] achieves KPRP-hiding, but only works for the static database. BDXT and ODXT gain forward and Type-II backward privacy, but do not hide KPRP.

In 2014, Pappas *et al.* [35] proposed Blind Seer. They adopt a tree-based index and use security computation to process searches. Blind Seer only reveals the search pattern, but it requires non-constant rounds of interactions. The schemes given in [24, 32, 50, 51] are also built on trees, with significant performance improvements. VBTree [50] also achieves forward privacy. However, three [32, 50, 51] of them leak the identifiers matched by every searched keyword, which is more severe than KPRP. Rphx [24] hides KPRP in the static setting, but it relies on hardware security provided by Intel SGX.

In 2017, Kamara and Moataz [25] proposed IEX by utilizing the inclusion-exclusion principle in the set theory. However, IEX leaks KPRP to the server. In [37], Patel *et al.* designed CNFFilter, a static scheme that reduces the leakages in IEX while ensuring efficiency. However, CNFFilter reveals the documents that contain both the first and the second keywords involved in a search.

Zuo *et al.* [57] utilize a bitmap index and symmetric homomorphic encryption to achieve FBDSSE-CQ. FBDSSE-CQ supports KPRP-hiding conjunctions, while reaching forward and Type-II backward privacy. However, their scheme suffers from linear search complexity and huge server storage.

Overall, there is no existing conjunctive DSSE schemes that achieve KPRP-hiding in sub-linear search efficiency, while ensuring forward and backward privacy.

9 CONCLUSION

In this work, we introduce a new cryptographic primitive: attribute-updatable hidden map encryption (AUHME), and design a secure AUHME construction. With AUHME as the primary tool, we propose HDXT, which is the first KPRP-hiding conjunctive DSSE solution with sub-linear search efficiency. Furthermore, HDXT simultaneously supports two crucial security properties: forward and backward privacy. The analysis and experiments show that the performance of HDXT is competitive compared with the previous schemes that do not have such strong security. In our future work, we aim to extend HDXT to process more complex searches and work for multi-client settings.

ACKNOWLEDGMENTS

Yuan and Russello acknowledge the MBIE-funded Stratus Research Programme (UOAX1910) for its support and inspiration for this research. All the authors thank the anonymous reviewers for their valuable comments and suggestions in improving the work.

REFERENCES

[1] Wikimedia dump service. https://dumps.wikimedia.org/enwiki/.

[2] Ghous Amjad, Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2023. Dynamic Volume-Hiding Encrypted Multi-Maps with Applications to Searchable Encryption. In *23rd Privacy Enhancing Technologies Symposium, PETS 2023, Lausanne, Switzerland.*

[3] Lucas Ballard, Seny Kamara, and Fabian Monrose. 2005. Achieving Efficient Conjunctive Keyword Searches over Encrypted Data. In *7th International Conference on Information and Communications Security, ICICS 2005, Beijing, China* (December 10-13). 414–426.

[4] Mihir Bellare, Roch Guérin, and Phillip Rogaway. 1995. XOR MACs: New Methods for Message Authentication Using Finite Pseudorandom Functions. In *15th Annual International Cryptology Conference, CRYPTO 1995, Santa Barbara, CA, USA* (August 27-31). 15–28.

[5] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2020. Revisiting Leakage Abuse Attacks. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, CA, USA* (February 23-26).

[6] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426.

[7] Raphaël Bost. 2016. Σοφος–Forward Secure Searchable Encryption. In *23rd ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, Vienna, Austria* (October 24-28). 1143–1154.

[8] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and backward private searchable encryption from constrained cryptographic primitives. In *24th ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA* (October 30-November 3). 1465–1482.

[9] Jin Wook Byun, Dong Hoon Lee, and Jongin Lim. 2006. Efficient Conjunctive Keyword Search on Encrypted Data Storage System. In *3rd European PKI Workshop: Theory and Practice, EuroPKI 2006, Turin, Italy* (June 19-20). 184–196.

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, Denver, CO, USA* (October 12-16). 668–679.

[11] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, and Michael Steiner. 2014. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, CA, USA* (February 23-26). 23–26.

[12] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *33rd Annual Cryptology Conference, CRYPTO 2013, Santa Barbara, CA, USA* (August 18-22). 353–373.

[13] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New Constructions for Forward and Backward Private Symmetric Searchable Encryption. In *25th ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada* (October 15-19). 1038–1055.

[14] Liqun Chen, Kaibin Huang, Mark Manulis, and Venkkatesh Sekar. 2021. Password-authenticated searchable encryption. *International Journal of Information Security* 20 (2021), 675–693.

[15] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA* (October 30 - November 3). 79–88.

[16] Wei Dai. Crypto++ Library 8.2. https://www.cryptopp.com.

[17] Facebook. Rocksdb 6.6.4. https://github.com/facebook/rocksdb/tree/v6.6.4.

[18] Marilyn George, Seny Kamara, and Tarik Moataz. 2021. Structured Encryption and Dynamic Leakage Suppression. In *40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2021, Zagreb, Croatia* (October 17-21). 370–396.

[19] Philippe Golle, Jessica Staddon, and Brent Waters. 2004. Secure Conjunctive Keyword Search over Encrypted Data. In *2nd International Conference on Applied Cryptography and Network Security, ACNS 2004, Yellow Mountain, China* (June 8-11). 31–45.

[20] Google. gRPC. https://github.com/grpc/grpc.

[21] Zichen Gui, Kenneth G. Paterson, and Sikhar Patranabis. 2022. Rethinking Searchable Symmetric Encryption. *https://eprint.iacr.org/2021/879* (2022).

[22] Thang Hoang and Attila A. Yavuz. 2021. A Secure Searchable Encryption Framework for Privacy-Critical Cloud Storage Services. *IEEE Transactions on Services Computing* 14, 6 (2021), 1675–1689.

[23] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, CA, USA* (February 5-8).

[24] Qin Jiang, Ee chien Chang, Yong Qi, Saiyu Qi, Pengfei Wu, and Jianfeng Wang. 2022. Rphx: Result Pattern Hiding Conjunctive Query Over Private Compressed Index Using Intel SGX. *IEEE Transactions on Information Forensics and Security* 17 (2022), 1053–1068.

[25] Seny Kamara and Tarik Moataz. 2017. Boolean Searchable Symmetric Encryption with Worst-Case Sub-linear Complexity. In *36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2017, Paris, France* (April 30–May 4). 94–124.

[26] Seny Kamara and Tarik Moataz. 2019. Computationally Volume-Hiding Structured Encryption. In *38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2019, Darmstadt, Germany* (May 19-23). 183–213.

[27] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. 2018. Structured Encryption and Leakage Suppression. In *38th Annual International Cryptology Conference, Crypto 2018, Santa Barbara, CA, USA* (August 19-23). 339–370.

[28] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *19th ACM conference on Computer and communications security, CCS 2012, Raleigh, NC, USA* (October 16-18). 965–976.

[29] Kee Sung Kim, Minkyu Kim, Dongsoo Lee, Je Hong Park, and Woo-Hwan Kim. 2017. Forward Secure Dynamic Searchable Symmetric Encryption with Efficient Updates. In *24th ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA* (October 30-November 3). 1449–1463.

[30] Russell W.F. Lai and Sherman S.M. Chow. 2017. Forward-secure searchable encryption on labeled bipartite graphs. In *15th International Conference on Applied Cryptography and Network Security, ACNS 2017, Kanazawa, Japan* (July 10-12). 478–497.

[31] Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K. Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shi-Feng Sun, Dongxi Liu, and Cong Zuo. 2018. Result Pattern Hiding Searchable Encryption for Conjunctive Queries. In *25th ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada* (October 15-19). 745–762.

[32] Rui Li and Alex X. Liu. 2017. Adaptively Secure Conjunctive Query Processing over Encrypted Data for Cloud Computing. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA* (April 19-22). 697–708.

[33] Simon Oya and Florian Kerschbaum. 2021. Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption. In *30th USENIX Security Symposium, USENIX Security 2021* (August 11-13). 127–142.

[34] Simon Oya and Florian Kerschbaum. 2022. IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA* (August 10-12). 2407–2424.

[35] Vasilis Pappa, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steven Bellovin. 2014. Blind Seer: A Scalable Private DBMS. In *35th IEEE Symposium on Security and Privacy, S&P 2014, Berkeley, CA, USA* (May 18-21). 359–374.

[36] Arash Partow. C++ Bloom Filter Library. http://www.partow.net/programming/bloomfilter/index.html.

[37] Sarvar Patel, Giuseppe Persiano, Joon Young Seo, and Kevin Yeo. 2021. Efficient Boolean Search over Encrypted Data with Reduced Leakage. In *27th International Conference on the Theory and Application of Cryptology and Information Security, ASIACRYPT 2021, Singapore* (December 6–10). 577–607.

[38] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing. In *26th ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK* (November 11-15). 79–93.

[39] Sikhar Patranabis and Debdeep Mukhopadhyay. 2021. Forward and Backward Private Conjunctive Searchable Symmetric Encryption. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021* (February 21-25).

[40] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. 2020. Practical Volume-Based Attacks on Encrypted Databases. In *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy* (September 7-11). 354–369.

[41] David Pouliot and Charles V. Wright. 2016. The Shadow Nemesis: Inference Attacks on Efficiently Deployable, Efficiently Searchable Encryption. In *23rd ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, Vienna, Austria* (October 24-28). 1341–1352.

[42] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *21st IEEE Symposium on Security and Privacy, S&P 2000, Berkeley, CA, USA* (May 14-17). 44–55.

[43] Xiangfu Song, Changyu Dong, Dandan Yuan, Qiuliang Xu, and Minghao Zhao. 2020. Forward private searchable symmetric encryption with optimized I/O efficiency. *IEEE Transactions on Dependable and Secure Computing* 17, 5 (2020), 912–927.

[44] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical Dynamic Searchable Encryption with Small Leakage. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, CA, USA* (February 23-26). 72–75.

[45] Shi-Feng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. 2018. Practical Backward-Secure Searchable Encryption from Symmetric Puncturable Encryption. In *25th ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada* (October 15-19). 763–780.

[46] Sameer Wagh, Paul Cuff, and Prateek Mittal. 2018. Differentially Private Oblivious RAM. In *18th Privacy Enhancing Technologies Symposium, PETS 2018, Barcelona, Spain* (July 24–27).

[47] Ding Wang and Ping Wang. 2018. Two Birds with One Stone: Two-Factor Authentication with Security Beyond Conventional Bound. *IEEE Transactions on Dependable and Secure Computing* 15, 4 (2018), 708–722.

[48] Jiafan Wang and Sherman S. M. Chow. 2022. Dynamic Searchable Symmetric Encryption Schemes Supporting Range Queries with Forward (and Backward) Security. In *22nd Privacy Enhancing Technologies Symposium, PETS 2022, Sydney, Australia* (July 11-15). 28–48.

[49] Qingxuan Wang, Ding Wang, Chi Cheng, and Debiao He. 2021. Quantum2FA: Efficient Quantum-Resistant Two-Factor Authentication Scheme for Mobile Devices. *IEEE Transactions on Dependable and Secure Computing* (2021).

[50] Zhiqiang Wu and Kenli Li. 2018. VBTree: forward secure conjunctive queries over encrypted data for cloud computing. *The International Journal on Very Large Data Bases* 28, 1 (2018), 25–46.

[51] Zhiqiang Wu, Kenli Li, Keqin Li, and Jin Wang. 2019. Fast Boolean Queries With Minimized Leakage for Encrypted Databases in Cloud Computing. *IEEE Access* 7 (2019), 49418–49431.

[52] Dandan Yuan. ConDSSESchemes. https://github.com/someoneapp/ConDSSESchemes.git.

[53] Dandan Yuan, Shujie Cui, and Giovanni Russello. 2022. We Can Make Mistakes: Fault-tolerant Forward Private Verifiable Dynamic Searchable Symmetric Encryption. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy* (June 6-10). 587–605.

[54] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *25th USENIX Security Symposium, USENIX Security 2016, Austin, TX, USA* (August 10-12). 707–720.

[55] Cong Zuo, Shi-Feng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. 2018. Dynamic Searchable Symmetric Encryption Schemes Supporting Range Queries with Forward (and Backward) Security. In *23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain* (September 3-7). 228–246.

[56] Cong Zuo, Shi-Feng Sun, Joseph K. Liu, Jun Shao, and Josef Pieprzyk. 2019. Dynamic Searchable Symmetric Encryption with Forward and Stronger Backward Privacy. In *24th European Symposium on Research in Computer Security, ESORICS 2019, Luxembourg* (September 23–27). 283–303.

[57] Cong Zuo, Shi-Feng Sun, Joseph K. Liu, Jun Shao, Josef Pieprzyk, and Guiyi Wei. 2003. Forward and Backward Private Dynamic Searchable Symmetric Encryption for Conjunctive Queries. *http://eprint.iacr.org/2020/1357* (2003).

A PROOF OF THEOREM 3.1

Proof. The construction for the ideal experiment is presented in Fig.12. This experiment AUHMEIDEAL$_{\mathcal{A},\mathcal{S}}(\lambda)$ could be obtained by gradually building the following three experiments.

$\mathbf{Exp}_0$: $\mathbf{Exp}_0$ is the real experiment AUHMEREAL$_{\mathcal{A}}(\lambda)$.

$\mathbf{Exp}_1$: To obtain $\mathbf{Exp}_1$, every call to the PRF $F(k, x)$ in $\mathbf{Exp}_0$ is replaced in the following way: if x is a new input, $\mathbf{Exp}_1$ chooses the output y uniformly at random from $\{0, 1\}^\lambda$ and inserts the pair (x, y) into a table $\mathcal{F}$, otherwise it outputs $\mathcal{F}[x]$. The ability to distinguish $\mathbf{Exp}_0$ and $\mathbf{Exp}_1$ could be reduced to that of breaking the security of the PRF.

$\mathbf{Exp}_2$: When $\phi^{\text{hme}}_{\mathbf{m_p}}(\mathbf{m_a})) = 0$ and $\beta = 1$, $\mathbf{Exp}_2$ selects d uniformly at random from $\{0, 1\}^\lambda$, instead of computing $d = H(r||xors)$ in $\mathbf{Exp}_1$. Since H is modeled as a random oracle, the two experiments might be distinguished only when $r||xors$ could be used as the input to H by the adversary, which is called the event *break* by us. r is randomly chosen but will be exposed to $\mathcal{A}$ in the query. When $\phi^{\text{hme}}_{\mathbf{m_p}}(\mathbf{m_a})) = 0$ and $\beta = 1$, following $\mathbf{Exp}_1$, $xors$ is indistinguishable from a random value. Therefore, for the adversary that makes α queries to H, the event *break* on $(\mathbf{m_p}, cnt)$ happens with less than $\alpha/2^\lambda$ probability. Assuming that there are a total of n^* distinct queries on $(\mathbf{m_p}, cnt)$ that satisfy $\phi^{\text{hme}}_{\mathbf{m_p}}(\mathbf{m_a})) = 0$ and $\beta = 1$, the chance of distinguishing $\mathbf{Exp}_1$ and $\mathbf{Exp}_2$ is less than $n^*\alpha/2^\lambda$

AUHMEIDEAL$_{\mathcal{A},\mathcal{S}}(\lambda)$:

(1) $\mathcal{A}(1^\lambda)$ outputs an attribute map $\mathbf{m_a} : \mathcal{K}_a \mapsto \{0, 1\}$. For $1 \le i \le |\mathbf{m_a}|$, $\mathcal{S}$ first selects $\ell_i \overset{\$}{\leftarrow} \{0, 1\}^\lambda$ and $v_i \overset{\$}{\leftarrow} \{0, 1\}^\lambda$. Then it sets $\mathcal{E}_1[i] = \ell_i$ and $\mathcal{E}_2[i] = v_i$ for $1 \le i \le |\mathbf{m_a}|$. After that, $\mathcal{S}$ sets $z = |\mathbf{m_a}|$ and $C_0 = \{(\ell_i, v_i)\}_{i=1}^{|\mathbf{m_a}|}$.

(2) $\mathcal{A}$ may adaptively makes ρ_1 queries.

★ **A Key Generation Query on $\mathbf{m_p}$:** Let Loc($\mathbf{m_p}$) = $\{\kappa_i\}_{i=1}^{l_1}$. $\mathcal{S}$(Loc($\mathbf{m_p}$), $\phi^{\text{hme}}_{\mathbf{m_p}}$($\mathbf{m_a}$)) generates the decryption key dk in the three steps: ❶ For $1 \le i \le l_1$, $\mathcal{S}$ sets $\ell_i = \mathcal{E}_1[\kappa_i]$; ❷ If $\phi^{\text{hme}}_{\mathbf{m_p}}(\mathbf{m_a}) = 1$, $\mathcal{S}$ first computes $xors = \oplus_{i=1}^{l_1} \mathcal{E}_2[\kappa_i]$, then selects $r \overset{\$}{\leftarrow} \{0, 1\}^\lambda$ and computes $d = H(r||xors)$. If $\phi^{\text{hme}}_{\mathbf{m_p}}(\mathbf{m_a}) = 0$, $\mathcal{S}$ chooses $r \overset{\$}{\leftarrow} \{0, 1\}^\lambda$ and $d \overset{\$}{\leftarrow} \{0, 1\}^\lambda$; ❸ $\mathcal{S}$ gives $\mathcal{A}$ the decryption key $dk = (\{\ell_i\}_{i=1}^{l_1}, r, d)$.

★ **An Update Query on** (op, k_u, v_u)**:** $\mathcal{S}$ initializes tok to an empty map.

If $\mathcal{L}^h_u(op, k_u, v_u) = add$, $\mathcal{S}$ works as follows: ❶ $\mathcal{S}$ first selects $\ell \overset{\$}{\leftarrow} \{0, 1\}^\lambda$ and $v \overset{\$}{\leftarrow} \{0, 1\}^\lambda$, and sets $tok[\ell] = v$; ❷ $\mathcal{S}$ sets $\mathcal{E}_1[z + 1] = \ell$ and $\mathcal{E}_2[z + 1] = v$. It updates z to $z + 1$; ❸ $\mathcal{S}$ gives $UTok = (add, tok)$ to $\mathcal{A}$.

If $\mathcal{L}^h_u(op, k_u, v_u) = 1$, ❶ For $1 \le i \le z$, $\mathcal{S}$ selects $u_i \overset{\$}{\leftarrow} \{0, 1\}^\lambda$ and sets $tok[\mathcal{E}_1[i]] = u_i$. ❷ For $1 \le i \le z$, $\mathcal{S}$ sets $\mathcal{E}_2[i] = \mathcal{E}_2[i] \oplus u_i$; ❸ $\mathcal{S}$ gives $UTok = (edit, tok)$ to $\mathcal{A}$.

(3) $\mathcal{S}$ sends C_0 to $\mathcal{A}$.

(4) $\mathcal{A}$ may make ρ_2 queries in an adaptive way, and each query is processed as in 2).

(5) Taking as input the view observed by $\mathcal{A}$ in the above operations, $\mathcal{A}$ outputs a bit b.

Figure 12: AUHMEIDEAL$_{\mathcal{A},\mathcal{S}}(\lambda)$

AUHMEIDEAL$_{\mathcal{A},\mathcal{S}}(\lambda)$: The ideal experiment is $\mathbf{Exp}_2$.

In conclusion, if F is a secure PRF and H is modeled as a random oracle, we can get that:

$$|\Pr[\text{AUHMEREAL}_{\mathcal{A}}(\lambda) = 1] - \Pr[\text{AUHMEIDEAL}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \le \text{Adv}^{prf}_{\mathcal{B}_1} + n^*\alpha/2^\lambda$$

where B_1 is an efficient adversary for PRF.

□

B PROOF OF THEOREM 5.1

```
Run $\mathcal{S}^{RHS}$.Setup($\mathcal{L}^{Stp}_{RHS}$(DB)), where the server obtains TMap
Client:
Γ, Ψ, $Z_1$, $Z_2$, ϒ ← empty maps
$\mathcal{S}^{HME}$ works as the step (1) in Fig.12, where it produces the vectors $\mathcal{E}_1$ and $\mathcal{E}_2$, the integer $z$, and the map $C_0$.
$\Gamma[t^{\triangleright}] \leftarrow \{1, \cdots, |W| \cdot |D|\}$
$t \leftarrow t^{\triangleright}$, XMap ← $C_0$
Send XMap to the server
```

Figure 13: $\mathcal{S}$**.Setup(**$\mathcal{L}^{Stp}_{RHS}$(DB), |W| · |D|**)**

Proof. In this section, we prove HDXT is adaptively secure with the leakage functions shown in Theorem 5.1 by constructing a simulator $\mathcal{S}$ for HDXT.

As shown in Section 4, HDXT only adopts two cryptographic primitives: RHS and AUHME. The simulator $\mathcal{S}$ for HDXT could be constructed by invoking the simulators for RHS and AUHME. We denote the simulator for RHS and AUHME as S^{RHS} and S^{HME}, respectively. The simulator $\mathcal{S}$ for the setup, update, and search protocols is presented in Fig.13, Fig.14, and Fig.15, respectively.

```
Parse $\mathcal{L}_{HDXT}^{Upd}$(DB, $op$, $in$) as ($\mathcal{L}_{RHS}^{Upd}$(DB, $op$, $in$), $\mathcal{L}_X$)
Run $\mathcal{S}^{RHS}$.Update($\mathcal{L}_{RHS}^{Upd}$( DB, $op$, $in$))

Client:
if $\mathcal{L}_X = (add, |\mathrm{W}|)$ then
    $UT \leftarrow$ empty map
    for $i = 1$ to $|\mathrm{W}|$ do
        Run $\mathcal{S}^{HME}(add)$ to process an update query, where the update token $UTok$ is generated, $\mathcal{E}_1$ and $\mathcal{E}_2$ is expanded, $z$ is updated to $z + 1$.
        $\Gamma[t] \leftarrow \Gamma[t] \cup \{z\}$
        $(add, ut) \leftarrow UTok$
        $UT \leftarrow UT \cup ut$
    end for
    $tok_x \leftarrow (add, UT)$
else if $\mathcal{L}_X = 1$ then
    Run $\mathcal{S}^{HME}(1)$ to process an update query, where the update token $UTok$ is generated and $\mathcal{E}_2$ is updated.
end if
if $UTok \neq \perp$ then
    $tok_x \leftarrow UTok$
    Send $tok_x$ to the server
end if
$t \leftarrow t + 1$
```

Figure 14: $\mathcal{S}$**.Update(**$\mathcal{L}_{HDXT}^{Upd}$(DB, op, in)**)**

We can directly get the simulator $\mathcal{S}$ for the setup protocol by invoking $\mathcal{S}^{RHS}$ and $\mathcal{S}^{HME}$. Notably, $\mathcal{S}$ creates five empty maps Γ, Ψ, Z_1, Z_2, and Υ, which are global variables in $\mathcal{S}$. We will detail these five variables later. $\mathcal{S}$ fills $\Gamma[t^{\blacktriangleright}]$ that records all the vector indices in $\mathcal{E}_1$.

In the simulator $\mathcal{S}$ for the update protocol, $\mathcal{S}$ first parses $\mathcal{L}_{HDXT}^{Upd}$(DB, op, in) to ($\mathcal{L}_{RHS}^{Upd}$(DB, op, in), $\mathcal{L}_X$)). $\mathcal{S}^{RHS}$ takes as input $\mathcal{L}_{RHS}^{Upd}$(DB, op, in) to simulate the process of updating TMap. When $\mathcal{L}_X = (add, |\mathrm{W}|)$, $\mathcal{S}$ invokes $S^{HME}(add)$ $|\mathrm{W}|$ times, where S^{HME} produces the token $UTok$. $\mathcal{S}$ gets tok_x from all the produced $UTok$, the process of which is the same as in the real game. Since fresh vector indices are added into $\mathcal{E}_1$ at this timestamp t, $\mathcal{S}$ stores these new indices into $\Gamma[t]$. If $\mathcal{L}_X = 1$, $S^{HME}(1)$ is run to generate the eviction token $UTok$. We can see that $\mathcal{S}$.Update is obtained just by replacing RHS.Update and HME.GenUpd with $\mathcal{S}^{RHS}$.Update and the update process in S^{HME}, respectively. Therefore, to distinguish the update protocols in the ideal and real games, $\mathcal{A}$ has to break the security of RHS or AUHME.

To get the simulator $\mathcal{S}$ for the search protocol, $\mathcal{S}^{RHS}$.Search is first run to simulate the process of searching for DB(q[1]). After that, the AUHME queries need to be simulated. In the real game, for each $id \in$ DB(q[1]), the client builds a map $I = \{(q[k]||id, 1)\}_{k=2}^{n}$ and calls HME.GenKey(msk, δ, I) to generate the decryption key dk for I. dk is inserted into a list DK. The entries of DK are randomly permuted before sending to the server. To simulate the process of producing dk, $\mathcal{S}$ needs to build Loc(I) for each $id \in$ DB(q[1]) and runs $S^{HME}(\mathrm{Loc}(I), \phi_I^{\mathrm{HME}}(\mathrm{DB}'))$. The function Loc is defined in Section 3.4. Every keyword-document concatenation $q[k]||id$ should match an unique vector index in $\mathcal{E}_1$. Loc(I) for id outputs the list $\{\epsilon_{k,id}\}_{k=2}^{n}$, where $\epsilon_{k,id}$ is the vector index matched by $q[k]||id$.

To simulate Loc(I) for each $id \in$ DB(q[1]), $\mathcal{S}$ fills or updates the global maps: Γ, Ψ, Z_1, Z_2, Υ as follows:

- $\Gamma[t]$ records the vector indices that were added to $\mathcal{E}_1$ at the timestamp t and have not been assigned to any keyword-document concatenation.
- For a conjunction q that occurs at t, $\Upsilon[t]$ is the number of documents that satisfy all the following three requirements: 1) belong to |DB(q[1])|; 2) added into the database at $t^{\blacktriangleright}$; 3) the document identifiers are not exposed to the adversary.
- Given a conjunction q that occurs at t, for each $id \in$ DB(q[1]), if id has been leaked, $\Psi[t, id, k]$ outputs the vector index matched by $q[k]||id$ for $2 \leq k \leq n$. If id is not leaked and was added into the database at the timestamp t_1 ($t_1 > t^{\blacktriangleright}$), the vector index matched by $q[k]||id$ is stored in $\Psi[t, t_1, k]$ for $2 \leq k \leq n$. For each identifier id that is not leaked and exists in the initial database, id could be denoted by any one in $\{*||i\}_{i=1}^{\Upsilon[t]}$ and the vector index matched by $q[k]||id$ could be any one in $\{\Psi[t, *||i, k]\}_{i=1}^{\Upsilon[t]}$.
- For any document identifier id that is leaked and was added into the database at t_1 ($t_1 > t^{\blacktriangleright}$), $Z_1[id]$ is set to t_1 and $Z_2[t_1]$ is set to id.

As shown in Line 3 - Line 30 in Fig.15, $\mathcal{S}$ first analyzes IP(q). It could get the vector indices matched by keyword-document concatenation that were already used by previous conjunctions and store them in $\mathfrak{L}$. $\mathfrak{L}[id, k]$ is the vector index of $q[k]||id$. Meanwhile, $\mathcal{S}$ obtains the set U, which stores all the document identifiers existing in IP(q). After analyzing IP(q), for each identifier $id \in U$, $\mathcal{S}$ builds the list Loc for id. For $2 \leq k \leq n$, if $\mathfrak{L}[id, k]$ is not empty, it is inserted into Loc. When $\mathfrak{L}[id, k]$ does not exist, it demonstrates that $q[k]||id$ has not been used by previous conjunctions. In this case, $\mathcal{S}$ first determines the timestamp t_1 that id was added and then selects an vector index ϵ from $\Gamma[t_1]$ uniformly at random. ϵ is used as the vector index of $q[k]||id$. After constructing Loc for id, $\mathcal{S}$ runs $S^{HME}(Loc, b)$ to generate the decryption key dk, where if $id \in$ DB(q), $b = 1$, otherwise $b = 0$. dk is inserted into the list DK.

For each entry $(t_1, id) \in$ TimeIds(DB(q)) that satisfies $id \notin U$ (t_1 is the timestamp that id was added), $\mathcal{S}$ builds Loc for id, by selecting the vector index ϵ from $\Gamma[t_1]$ and inserting ϵ into Loc. $(Loc, 1)$ is transferred to S^{HME} that then produces dk. dk is inserted into DK.

After processing the identifiers in $U \cup$ DB(q), $\mathcal{S}$ starts to process every document that satisfy all the following three conditions: 1) belongs to DB(q[1]); 2) does not exist in IP(q); 3) added after $t^{\blacktriangleright}$. The timestamps that these documents were added are stored in AddTimes(q[1]) $\setminus$ Pt. Pt is the set of the timestamps occurring in IP(q) and TimeIds(DB(q)). For each $t_1 \in$ AddTimes(q[1]) $\setminus$ Pt, $\mathcal{S}$ selects an vector index ϵ from $\Gamma[t_1]$ and inserts ϵ into Loc for $2 \leq k \leq n$. $\mathcal{S}^{HME}(Loc, 0)$ is run to generate dk, which is inserted into the list DK.

At last, $\mathcal{S}$ processes the documents that: 1) belong to DB(q[1]); 2) do not exist in IP(q); 3) exist in the initial database. Each vector index ϵ is selected from $\Gamma[t^{\blacktriangleright}]$ and inserted into Loc. $\mathcal{S}^{HME}(Loc, 0)$ is run to generate dk, which is inserted into DK. $\mathcal{S}$ randomly permutes entries of DK and sends DK to the server.

In the simulator $\mathcal{S}$ for the search protocol, when $q[k]||id$ has not been queried by previous conjunctions, $\mathcal{S}$ selects the vector index of $q[k]||id$ from $\Gamma[t_1]$ uniformly at random, where t_1 is the timestamp that id was added. This is the only difference with the real search protocol. Because the entries of DB$'$ are randomly permuted before calling HME.Enc in the real setup protocol and the entries of addition token are also randomly permuted after calling HME.GenUpd, $\mathcal{S}$ cannot distinguish the real and the ideal game.

In conclusion, we can get that:

```
Run $\mathcal{S}^{RHS}$.Search($\mathcal{L}^{Srch}_{RHS}$(DB, $q[1]$))
Client:
(IP($q[1], q[2]$), $\cdots$, IP($q[1], q[n]$)) $\leftarrow$ IP($q$)
$\mathfrak{L} \leftarrow$ empty map
$U \leftarrow$ empty set
for each $k = 2$ to $n$ do
    Sort the entries of IP($q[1], q[k]$) in ascending order according to the timestamp in each entry
    for each $(t_1, j, \Omega) \in$ IP($q[1], q[k]$) do
        for each $(t_2, id) \in \Omega$ do
            if $id \notin U$ then
                $U \leftarrow U \cup \{id\}$
            end if
            if $\Psi[t_1, id, j] \neq \perp$ then
                $\epsilon \leftarrow \Psi[t', id, j],\ \mathfrak{L}[id, k] \leftarrow \epsilon$
            else
                if $t_2 \neq t^{\triangleright}$ then
                    $\eta \leftarrow t_2,\ Z_1[id] \leftarrow t_2,\ Z_2[t_2] \leftarrow id$
                else if $t_2 = t^{\triangleright}$ then
                    $l \leftarrow \Upsilon[t_1],\ \eta \leftarrow *||l,\ \Upsilon[t_1] \leftarrow l - 1$
                end if
                $\epsilon \leftarrow \Psi[t_1, \eta, j],\ \mathfrak{L}[id, k] \leftarrow \epsilon$
                $j' = 2$
                while $\Psi[t_1, \eta, j'] \neq \perp$ do
                    $\Psi[t_1, id, j'] \leftarrow \Psi[t_1, \eta, j']$
                    Delete $\Psi[t_1, \eta, j']$
                    $j' \leftarrow j' + 1$
                end while
            end if
        end for
    end for
end for
for each $id \in U$ do
    $Loc \leftarrow$ empty list
    for $k = 2$ to $n$ do
        if $\mathfrak{L}[id, k] \neq \perp$ then
            $\epsilon \leftarrow \mathfrak{L}[id, k],\ Loc \leftarrow Loc \cup \{\epsilon\}$
        else
            if $Z_1[id] \neq \perp$ then
                $t_1 \leftarrow Z_1[id]$
            else
                $t_1 \leftarrow t^{\triangleright}$
            end if
            $\epsilon \xleftarrow{\$} \Gamma[t_1],\ \Gamma[t_1] \leftarrow \Gamma[t_1] \setminus \{\epsilon\}$
            $\Psi[t, id, k] \leftarrow \epsilon,\ Loc \leftarrow Loc \cup \{\epsilon\}$
        end if
    end for
    if $id \in$ DB($q$) then
        Run $\mathcal{S}^{HME}(Loc, 1)$ to process a key generation query, where the decryption key $dk$ is generated
    else
        Run $\mathcal{S}^{HME}(Loc, 0)$ to process a key generation query, where the decryption key $dk$ is generated
    end if
    $DK \leftarrow DK \cup \{dk\}$
end for
for each $(t_1, id) \in$ TimeIds(DB($q$)) s.t. $id \notin U$ do
    $Loc \leftarrow$ empty list
    if $t_1 \neq t^{\triangleright}$ then
        $Z_1[id] \leftarrow t_1,\ Z_2[t_1] \leftarrow id$
    end if
    for $k = 2$ to $n$ do
        $\epsilon \xleftarrow{\$} \Gamma[t_1],\ \Gamma[t_1] \leftarrow \Gamma[t_1] \setminus \{\epsilon\}$
        $\Psi[t, id, k] \leftarrow \epsilon,\ Loc \leftarrow Loc \cup \{\epsilon\}$
    end for
    Run $\mathcal{S}^{HME}(Loc, 1)$ to process a key generation query, where the decryption key $dk$ is generated
    $DK \leftarrow DK \cup \{dk\}$
end for
$Pt \leftarrow$ empty set
Take all the timestamps existing in TimeIds(DB($q$)) or IP($q$) and store them in $Pt$
for each $t_1 \in$ AddTimes($q[1]$) $\setminus Pt$ do
    $Loc \leftarrow$ empty list
    for $k = 2$ to $n$ do
        $\epsilon \xleftarrow{\$} \Gamma[t_1],\ \Gamma[t_1] \leftarrow \Gamma[t_1] \setminus \{\epsilon\}$
        if $Z_2[t_1] \neq \perp$ then
            $id \leftarrow Z_2[t_1],\ \Psi[t, id, k] \leftarrow \epsilon$
        else
            $\Psi[t, t_1, k] \leftarrow \epsilon$
        end if
        $Loc \leftarrow Loc \cup \{\epsilon\}$
    end for
    Run $\mathcal{S}^{HME}(Loc, 0)$ to process a key generation query, where the decryption key $dk$ is generated
    $DK \leftarrow DK \cup \{dk\}$
end for
$\Upsilon[t] \leftarrow |$DB($q[1]$)$| - |$AddTimes($q[1]$) $\cup Pt|$
for $i = 1$ to $\Upsilon[t]$ do
    $Loc \leftarrow$ empty list
    for $k = 2$ to $n$ do
        $\epsilon \xleftarrow{\$} \Gamma[t^{\triangleright}],\ \Gamma[t^{\triangleright}] \leftarrow \Gamma \setminus \{\epsilon\}$
        $\Psi[t, *||i, k] \leftarrow \epsilon,\ Loc \leftarrow Loc \cup \{\epsilon\}$
    end for
    Run $\mathcal{S}^{HME}(Loc, 0)$ to process a key generation query, where the decryption key $dk$ is generated
    $DK \leftarrow DK \cup \{dk\}$
end for
Randomly permute the entries of $DK$
Send $DK$ to the server
$t \leftarrow t + 1$
```

Figure 15: $\mathcal{S}$.Search($\mathcal{L}^{Srch}_{RHS}$(DB, $q[1]$), TimeIds(DB(q)), $|$DB($q[1]$)$|$, IP(q), AddTims($q[1]$))

$$|\Pr[\text{SSEReal}^{\text{HDXT}}_{\mathcal{A}}(\lambda) = 1] - \Pr[\text{SSEIdeal}^{\text{HDXT}}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda) = 1]| \leqslant \text{Adv}^{RHS}_{\mathcal{B}_2} + \text{Adv}^{AUHME}_{\mathcal{B}_3}$$

where $\mathcal{B}_2$ and $\mathcal{B}_3$ are efficient adversaries for $\mathcal{L}_{RHS}$-adaptively-secure RHS and selective-semantically secure AUHME, respectively. □

C BACKWARD PRIVACY DEFINITION

In Definition 2.4, we give the definition for backward private conjunctive DSSE. The definition also works for single-keyword DSSE as follows.

Definition C.1. ***(Backward Privacy of Single-keyword DSSE)*** *A $\mathcal{L}-adptively-secure$ DSSE scheme Σ =* ***{Setup, Search, Update}*** *is*

Type-I *backward private iff*

$$\mathcal{L}^{Updt}(\text{DB}, op, (id, \text{W}_{id})) = \mathcal{L}'(op, |\text{W}_{id}|)$$
$$\mathcal{L}^{Srch}(\text{DB}, w) = \mathcal{L}''(\text{TimeDB}(w), \pi(w))$$

Type-II *backward private iff*

$$\mathcal{L}^{Updt}(\text{DB}, op, (id, \text{W}_{id})) = \mathcal{L}'(op, \text{W}_{id})$$
$$\mathcal{L}^{Srch}(\text{DB}, w) = \mathcal{L}''(\text{TimeDB}(w), \text{Updates}(w), \pi^{\triangleright}(w))$$

where $\pi^{\triangleright}(w)$ is the the number of document identifiers matching w in $\text{DB}^{\triangleright}$, $\pi(w)$ is the sum of $\pi^{\triangleright}(w)$ and the number of updates related to w, $\mathcal{L}'$ and $\mathcal{L}''$ are stateless functions.

The above definition differs slightly from Bost *et al.*'s [8] in two aspects. First, TimeDB(w) in [8] captures DB(w) and the timestamps that these document identifiers are inserted into DB(w), while our TimeDB(w) records DB(w) and the timestamps that these identifiers are first added into the database (when they might not contain w). We argue that the exposed timestamps in our definition reveal nothing about the deletion information, so they will not influence backward privacy. Bost *et al.*'s TimeDB(w) is not suitable for the complex setting we consider, where a keyword-document pair could be inserted into the database again after it has been deleted. In this setting, there might exist multiple timestamps where an identifier id was added to DB(w), from which the server could infer when the previous deletions happened. For instance, if the server learns that a keyword-document pair (w, id) is inserted into the database in the two timestamps t_1, t_3, it could get that the pair was deleted once at the timestamp t_2. Second, for Type-II backward privacy, the search leakage in our definition captures $\pi^{\triangleright}(w)$. $\pi^{\triangleright}(w)$ is not included in Bost *et al.*'s definition just because they assume the initial database is empty.

D HDXT$_{SU}$ - SUBLINEAR UPDATES

In this section, we propose HDXT$_{SU}$, which aims to reduce the edit complexity of HDXT to be sub-linear.

The encrypted index of HDXT$_{SU}$ consists of TMap and XMap2. TMap is the same as that in HDXT. XMap2 is the new version of XMap. The pseudocodes of HDXT$_{SU}$ are shown in Fig.16, Fig.17, and Fig.18. HDXT$_{SU}$ adopts the pseudorandom function F and our AUHME construction. Every document has an independent AUHME instance, so in principle the client needs to store the master secret key and the state per AUHME instance. To reduce the client storage, the master secret key is pseudorandomly computed from

```
Run $(K_T, s_T, \text{TMap}) \leftarrow$
        RHS.Setup$(1^\lambda, \text{DB})$
Client:
$k_d \xleftarrow{\$} \{0,1\}^\lambda$
XMap2 $\leftarrow$ empty maps
for each $id \in \text{ID}$ do
    $\text{DB}'_{id} \leftarrow$ empty map
    for each keyword $w$ contained in $id$ do
        $\text{DB}'_{id}[w] \leftarrow 1$
    end for
    for each keyword $w$ not contained in $id$ do
        $\text{DB}'_{id}[w] \leftarrow 0$
    end for
    $k_{id1} \leftarrow F(k_d, id||0)$
    $k_{id2} \leftarrow F(k_d, id||1)$
    $k_{id3} \leftarrow F(k_d, id||2)$
    $(msk, \delta) \leftarrow$ HME.Setup$(1^\lambda)$
        where $msk$ is set to be $(k_{id1}, k_{id2}, k_{id3})$
    $\Delta[id] \leftarrow \delta$
    $\text{C}_{id} \leftarrow$ HME.Enc$(msk, \text{DB}'_{id})$
    Insert all entries of $\text{C}_{id}$ into XMap2
end for
Randomly permute entries of XMap2
Send XMap2 to the server
$K \leftarrow (K_T, k_d),\ tsiz \leftarrow 0$
EId $\leftarrow$ empty set
$s \leftarrow (s_T, tsiz, \text{EId}, \Delta)$
return $(K, s)$

Server:
return EDB=(TMap, XMap2)
```

Figure 16: HDXT$_{SU}$.Setup(1^λ, DB)

```
Run $(s_T; \text{TMap}) \leftarrow$ RHS.
    Update$(K_T, s_T, op, in; \text{TMap})$,
    where the client updates $s_T$ and the
    server updates TMap

Client:
$(id, \text{W}_{id}) \leftarrow in$
for each $w \in \text{W}_{id}$ do
    $(tok_x, tsiz, \text{EId}, \Delta) \leftarrow$ EditPair2(
            $k_d, tsiz, \text{EId}, \Delta, op, id, w$)
    if $tok_x \neq \perp$ then
        Send $tok_x$ to the server
    end if
end for
$s \leftarrow (s_T, tsiz, \text{EId}, \Delta)$
return $s$

Server:
XMap2 $\leftarrow$ HME.ApplyUpd$(tok_x,$
        XMap2)
return EDB = (TMap, XMap2)


EditPair2$(k_d, tsiz, \text{EId}, \Delta, op, id, w)$
if $op = edit^+$ then
    $b \leftarrow 1$
else if $op = edit^-$ then
    $b \leftarrow 0$
end if
EId $\leftarrow$ EId $\cup \{id\}$
if $tsiz + 1 \geq |\text{W}|$ then
    $UT \leftarrow$ empty map
    for $id' \in$ EId do
        $\delta_{id'} \leftarrow \Delta[id']$
        $(cnt, T, \perp, \perp) \leftarrow \Delta[id']$
        $k_{id1} \leftarrow F(k_d, id'||0)$
        $k_{id2} \leftarrow F(k_d, id'||1)$
        $k_{id3} \leftarrow F(k_d, id'||2)$
        $msk \leftarrow (k_{id1}, k_{id2}, k_{id_3})$
        $S \leftarrow$ empty set
        for each $w' \in$ W do
            $\ell \leftarrow F(k_{id1}, w')$
            $S \leftarrow S \cup \{\ell\}$
        end for
        $\delta_{id'} \leftarrow (cnt, T, 0, S)$
        if $id' = id$ then
            $(UTok, \delta_{id'}) \leftarrow$ HME.
                GenUpd$(msk, \delta_{id'}, edit, w, b)$)
        else
            $(UTok, \delta_{id'}) \leftarrow$ HME.
                GenUpd$(msk, \delta_{id'}, edit, \perp, \perp)$
        end if
        $(edit, ut) \leftarrow UTok$
        $UT \leftarrow UT \cup ut$
        $\Delta[id'] \leftarrow \delta_{id'}$
    end for
    Randomly permute entries of $UT$
    $tok_x \leftarrow (edit, UT)$
    $tsiz \leftarrow 0$, Clear EId
else
    $k_{id1} \leftarrow F(k_d, id||0)$
    $k_{id2} \leftarrow F(k_d, id||1)$
    $k_{id3} \leftarrow F(k_d, id||2)$
    $msk \leftarrow (k_{id1}, k_{id2}, k_{id_3})$
    $\delta_{id} \leftarrow \Delta[id]$
    Take cache $T$ from $\delta_{id}$
    $tsiz \leftarrow tsiz - |T|$
    $(tok_x, \delta_{id}) \leftarrow$ HME.GenUpd(
            $msk, \delta_{id}, edit, w, b$)
    Take cache $T$ from $\delta_{id}$
    $tsiz \leftarrow tsiz + |T|$
    $\Delta[id] \leftarrow \delta_{id}$
end if
return $(tok_x, tsiz, \text{EId}, \Delta)$
```

Figure 17: HDXT$_{SU}$.Update($K, s, edit, in$)

the corresponding document, instead of being randomly generated. Moreover, recall that the Query (or ApplyUpd) procedure in our AUHME construction uses the elements contain in the decryption key (or the update token) to find the entries that need to be operated on in the ciphertext. This implies that the two procedures can still processed correctly when the ciphertext is replaced with a superset of the ciphertext. In HDXT$_{SU}$, for reducing the leakages, the two procedures take XMap2 as the input.

```
Run DB$(w_1) \leftarrow$ RHS.Search(
    $K_T, s_T, w_1; \text{TMap}$), where the client
    receives DB$(w_1)$.

Client:
$R_1, DK' \leftarrow$ empty list
Insert DB$(w_1)$ into $R_1$ and Randomly
    permute the entries of $R_1$
for $j = 1$ to $|R_1|$ do
    $id \leftarrow R_1[j], I'_j \leftarrow$ empty map
    for $i = 2$ to $n$ do
        $I'_j[w_i] \leftarrow 1$
    end for
    $k_{id1} \leftarrow F(k_d, id||0)$
    $k_{id2} \leftarrow F(k_d, id||1)$
    $k_{id3} \leftarrow F(k_d, id||2)$
    $msk \leftarrow (k_{id1}, k_{id2}, k_{id3})$
    $dk \leftarrow$ HME.GenKey(
            $msk, \Delta[id], I'_j$)
    $DK' \leftarrow DK' \cup \{dk\}$
end for
Send $DK'$ to the server

Server:
$Pos \leftarrow$ empty set
for $j = 1$ to $|DK|$ do
    $r \leftarrow$ HME.Query$(DK[j], \text{XMap2})$
    if $r = 1$ then
        $Pos \leftarrow Pos \cup \{j\}$
    end if
end for
Send $Pos$ to the client

Client:
$R \leftarrow$ empty set
for each $j \in Pos$ do
    $R \leftarrow R \cup \{R_1[j]\}$
end for
return $R$
```

Figure 18: HDXT$_{SU}$.Search($K, s, w_1 \wedge \cdots \wedge w_n$)

In HDXT$_{SU}$, the client keeps the secret key $K = (K_T, k_d)$ and the state $s = (s_T, tsiz, \text{EId}, \Delta)$. The key k_d is used for deriving the master secret key adopted by every AUHME instance. $tsiz$ is the number of the edited keyword-identifier pairs since the last eviction. EId stores the identifiers of the edited documents since the last eviction. Δ maps every document id to the AUHME state corresponding to id. We specify that the maximum value of $tsiz$ is $|\text{W}|$.

- $(K, s; \text{EDB}) \leftarrow$ **HDXT$_{SU}$.Setup**$(\lambda, \text{DB}; \perp)$: As shown in Fig.16, the setup phase generates EDB = (TMap, XMap2). XMap2 stores $\{C_{id}\}_{id \in \text{ID}}$, where C_{id} is obtained by using an AUHME instance to encrypt DB'_{id}. The client uses k_d to derive the master secret key from id.
- $(s; \text{EDB}) \leftarrow$ **HDXT$_{SU}$.Update**$(K, s, op, in; \text{EDB})$: The client parses in to (id, W_{id}). TMap is updated as in HDXT. If $op = add$, to update XMap2, similar to HDXT, the client calls HME. GenUpd $|\text{W}|$ times to add the ciphertext of DB'_{id} into XMap2. When $op = del$, as in HDXT, only TMap is updated.
  Fig.17 shows the edit procedure. For each $w \in \text{W}_{id}$, the client calls **EditPair2**$(k_d, tsiz, \text{EId}, \Delta, op, id, w)$ to produce a token tok_x to update XMap2 and updates $(tsiz, \text{EId}, \Delta)$. The server uses tok_x to update XMap2.
  EditPair2 first inserts id into EId. Then it checks whether $tsiz + 1 \geq |\text{W}|$. If $tsiz + 1 < |\text{W}|$, HME.GenUpd is invoked to insert (w, b) into the cache T stored in $\Delta[id]$. $tsiz$ is updated based on the size of T. $(\perp, tsiz, \text{EId}, \Delta)$ is returned.
  If $tsiz + 1 \geq |\text{W}|$, **EditPair2** finds all the non-empty caches through EId and Δ. for each $id' \in \text{EId}$, the client forces the cache of id' to be evicted by setting the third parameter ζ in $\Delta[id']$ to 0. It calls HME.GenUpd to generate the eviction token $UTok$ for id'. All the produced $UTok$ are then merged into tok_x.
- $(\text{DB}(w_1 \wedge\ w_2, \cdots, \wedge w_n)$); EDB) $\leftarrow$ **HDXT$_{SU}$.Search** $(K, s, w_1 \wedge w_2, \cdots, \wedge w_n; \text{EDB})$: Within a conjunction, as in HDXT, the client gets DB(w_1) and inserts DB(w_1) into the list R_1 in random order. Then for each $id \in R_1$, the client builds the map I', where $I'[w_i]$ is set to 1 for $2 \leq i \leq n$. It queries whether I'

is a subset of DB'_{id} through the AUHME query. If the query returns 1, *id* is inserted into the search result *R*.

D.1 Security

This sub-section continue using the notations and functions defined in Section 5.1. For HDXT_{SU}, we claim that in addition to when the evictions occur, the documents involved in each eviction also could be obtained based on Q and $|\mathrm{W}|$. If an eviction occurs during (op, in), we use $\mathrm{Evids}(op, in)$ to denote the documents involved in the eviction, otherwise $\mathrm{Evids}(op, in)$ is empty. Below we define a function $\mathrm{EvP}(op, in)$ for an update query (op, in) and a function $\mathrm{TimeEv}(q[1])$ for a conjunctive query q.

If an eviction occurs within (op, in), $\mathrm{EvP}(op, in)$ outputs the eviction pattern, otherwise it outputs nothing. The eviction pattern includes: 1) when every document in $\mathrm{Evids}(op, in)$ (except for the ones existing in $\mathrm{DB}^{\triangleright}$) was added into the database; 2) the timestamps of the previous evictions that involve the documents that belong to $\mathrm{Evids}(op, in)$; 3) the timestamps of the previous conjunctions whose *s-term* matches at least one document that belongs to $\mathrm{Evids}(op, in)$. Formally, $\mathrm{EvP}(op, in) = \{t \mid \exists id \in \mathrm{Evids}(op, in)$ and $\mathrm{W}_{id} : (t, add, (id, \mathrm{W}_{id})) \in Q\} \cup \{t \mid \exists id \in \mathrm{Evids}(op, in)$ and $(op', in') : id \in \mathrm{Evids}(op', in')$ and $(t, op', in') \in Q\} \cup \{t \mid \exists id \in \mathrm{Evids}(op, in)$ and $q : id \in \mathrm{DB}(q[1])$ and $(t, q) \in Q\}$.

For a conjunctive query q, $\mathrm{TimeEv}(q[1])$ outputs the timestamps of the previous evictions that involve at least one document identifiers matching $q[1]$. $\mathrm{TimeEv}(q[1]) = \{t \mid \exists id \in \mathrm{DB}(q[1])$ and $(op, in) : id \in \mathrm{Evids}(op, in)$ and $(t, op, in) \in Q\}$.

Within an addition, deletion, or edit operation without an eviction, HDXT_{SU} has the same leakages as HDXT. During an eviction, the server learns which entries in XMap2 are accessed and could link this eviction to the previous queries, which exposes the eviction pattern. During a conjunction q, in addition to the leakages in HDXT, HDXT_{SU} also could associate q to previous evictions that access the same entries of XMap2, which is captured by $\mathrm{TimeEv}(q)$. Formally, we have the Theorem D.1.

THEOREM D.1. *If F is a secure PRF,* RHS *is $\mathcal{L}_{RHS}$-adaptively secure, and AUHME is selectively-semantically secure as defined in Section 3, HDXT_{SU} is $\mathcal{L}_{HDXT_{su}}$-adaptively secure where*

(1) $\mathcal{L}^{Stp}_{HDXT_{su}}(\mathrm{DB}) = (\mathcal{L}^{Stp}_{RHS}(\mathrm{DB}), |\mathrm{W}| \cdot |\mathrm{D}|)$

(2) $\mathcal{L}^{Upd}_{HDXT_{su}}(\mathrm{DB}, op, in) =$

$$\begin{cases} (\mathcal{L}^{Upd}_{RHS}(\mathrm{DB}, op, in), add, |\mathrm{W}|), & op = add \\ (\mathcal{L}^{Upd}_{RHS}(\mathrm{DB}, op, in), \mathrm{EvP}(op, in)) & op \neq add \end{cases}$$

(3) $\mathcal{L}^{Srch}_{HDXT_{su}}(\mathrm{DB}, q) = (\mathcal{L}^{Srch}_{RHS}(\mathrm{DB}, q[1]), \mathrm{TimIds}(\mathrm{DB}(q)),$

$\mathrm{DB}(q[1]), \mathrm{IP}(q), \mathrm{AddTims}(q[1]), \mathrm{TimeEv}(q[1]))$

Although HDXT_{SU} leaks more than HDXT, it does not reveal $\mathrm{DB}(w_i) \cap \mathrm{DB}(w_j)$ for all $2 \leq i < j \leq n$. Therefore, HDXT_{SU} does not break KPRP-hiding. For an update query, the server cannot learn any information about the updated keywords, which ensures forward privacy. However, HDXT_{SU} does not satisfy backward privacy because it exposes the edited documents within an eviction.

D.2 Performance Analysis

As in Section 5.2, we take MITRA [13] as an instantiation for RHS. The overheads caused by the setup, addition, deletion, and search protocols are the same as in HDXT. An edit query without evictions incur $O(1)$ computational complexity. If evictions occur, the produced overhead is $O(t \cdot |\mathrm{W}|)$, where t refers to the involved document identifiers in this eviction operation. Therefore, the amortized overhead for an edit query is $O(t)$.

The cost for the server storage is the same as in HDXT, which is $O(|\mathrm{W}| \cdot |\mathrm{D}|)$. As in HDXT, the client costs $O(|\mathrm{W}|(\log|\mathrm{D}| + \lambda))$ storage space for storing the state for MITRA and the caches. For HDXT_{SU}, the client additionally requires at most $O(|\mathrm{D}| \log(N/|\mathrm{W}|))$ bits for the counters in every AUHME instance. The experiment show that the additionally incured client storage is very small. As HDXT, HDXT_{SU} can process evictions in a streaming way to avoid taking up a lot of working client storage for the evictions.