The following publication H. Wu, Z. Peng, J. Xiao, L. Xue, C. Lin and S. -H. Chung, "HeX: Encrypted Rich Queries With Forward and Backward Privacy Using Trusted Hardware," in IEEE Transactions on Dependable and Secure Computing, vol. 22, no. 4, pp. 3751-3765, July-Aug. 2025 is available at https://doi.org/10.1109/TDSC.2025.3540958.

HeX: Encrypted Rich Queries with Forward and Backward Privacy Using Trusted Hardware

Haotian Wu, Zhe Peng, Jiang Xiao, Lei Xue, Chenhao Lin, Sai-Ho Chung

Abstract—Dynamic searchable symmetric encryption (DSSE) schemes empower data owners to outsource their encrypted data to clouds while retaining the ability to update or search on it. Despite a lot of efforts devoted in recent years, there are still several challenges that have not been well addressed. Firstly, the confidentiality of data might be compromised if forward privacy and backward privacy cannot be ensured. Secondly, only the traditional single keyword-file search has attracted tremendous attention, while other popular queries like Boolean queries and range queries are not fully investigated. Lastly, how to solve these problems on untrusted servers that may deviate from pre-defined protocols is also challenging. In this paper, aiming to tackle the above problems, we propose a novel DSSE scheme named HeX based on Trusted Execution Environment (TEE) that supports rich queries on untrusted servers while guaranteeing forward and backward privacy. We achieve strong forward and backward security by designing a deferred obfuscating read-write technique atop the bitmap index. We further extend the basic scheme to realize Boolean queries and range queries by reducing them to basic keyword queries. Strict theoretical analysis is conducted to prove the security of HeX, and extensive evaluations illustrate its efficiency and practicality.

Index Terms—Dynamic searchable symmetric encryption, forward privacy, backward privacy, Trusted Execution Environment, rich queries.

1 Introduction

Dynamic searchable symmetric encryption (DSSE) has become a popular trend for data owners to outsource their private data to cloud servers. It can not only protect sensitive information through encryption, but also reserve the ability to update and retrieve the data. Early efforts are devoted to the design of index building and search algorithms during the evolution from static searchable symmetric encryption (SSE) [1] to DSSE [2], [3]. Along with the development of DSSE, the additional leakage caused by the data update in DSSE, for example, forward privacy and backward privacy [4], has gained a lot of interest.

Forward and backward privacy depict the information leakage during the update and search operations of DSSE respectively. Forward privacy captures the association between the previously queried keywords and the currently updated data. The leakage of forward privacy will enable attackers to infer whether the newly updated data contains previously seen keywords. It was first introduced in [5] and got increasingly significant attention after a file-injection attack was proposed in [6]. This attack can reveal the client's queries by injecting very few files. As for backward privacy, it is proposed to limit the information about the matching files that have been deleted upon a keyword search. With the backward privacy leakage, attackers can get all matching files on a keyword even if some of the files have been deleted and are no longer valid. This notion is formally defined in [4] and categorized into three types according to the leakage level. The strongest level of backward privacy only leaks the number of updates on keywords. Over the last decade, many efforts have been made to design efficient constructions for forward and backward private DSSE schemes [7], [8], [9], [10]. Some works leverage the technique of oblivious RAM (ORAM) [11], [12], [13], [14] to build highly secure constructions of DSSE by completely hiding access patterns. But the prohibitive overhead renders these schemes inefficient and impractical. Other attempts utilize the Trusted Execution Environment (known as TEE or enclave), which is an isolated area guarded by trusted hardware, to achieve strong security and high efficiency [15], [16], [17]. However, their level of forward and backward privacy still has space for improvement. Ensuring strong forward and backward privacy has become a fundamental issue of building privacy-preserving DSSE schemes.

In terms of the query type, most traditional DSSE designs mainly focus on the single keyword search over files. Only a few works investigate rich queries like Boolean query (e.g., "return all documents containing keywords both 'forward' and 'backward'") and range query (e.g., "return all books whose page number is large than 100"), which are more ubiquitous in real-world applications [18], [19], [20]. Some designs for Boolean queries are limited to conjunctive

- *Haotian Wu and Sai-Ho Chung are with the Department of Industrial and Systems Engineering, The Hong Kong Polytechnic University, Hung Hom, Hong Kong.*
  E-mail: haotian.wu@connect.polyu.hk; nick.sh.chung@polyu.edu.hk.
- *Zhe Peng is with the Department of Industrial and Systems Engineering, The Hong Kong Polytechnic University, and The Hong Kong Polytechnic University Shenzhen Research Institute.*
  E-mail: jeffrey-zhe.peng@polyu.edu.hk.
- *Jiang Xiao is with the School of Computer Science and Technology, Huazhong University of Science and Technology, China.*
  E-mail: jiangxiao@hust.edu.cn.
- *Lei Xue is with the School of Cyber Science and Technology, Sun Yat-sen University, China.*
  E-mail: xuelei3@mail.sysu.edu.cn.
- *Chenhao Lin is with the Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, China.*
  E-mail: linchenhao@xjtu.edu.cn.

Manuscript received October 3, 2023. (Corresponding author: Zhe Peng.)

queries [21] or inefficient for negated queries [22]. On the other side, several studies [23], [24], [25] enable DSSE to support range queries, but fail to achieve both forward and backward privacy. Efficient Boolean queries and range queries usually entail corresponding complicated indexes, which leads to weaker privacy due to their additional leakage. Hence, how to support rich queries while maintaining strong security in DSSE is still challenging to explore.

Another issue that has always been overlooked about DSSE is that many works are conducted based on the assumption of the semi-honest server setting. However, this assumption may be invalid in practical scenarios where untrusted servers can return non-conforming data. To remedy this issue, some verifiable DSSE schemes have been proposed [26], [27], [28], [29], [30]. Some of these designs rely on the construction of authenticated data structure (ADS) [26], [27], or employ the blockchain technology [31], [32], [33] to serve the result verification [28], [29], [30]. These underlying verification techniques will incur extra overhead and lead to low efficiency of encrypted search.

We conclude the problem to solve in this paper as: *how to support rich queries over encrypted data on untrusted servers while ensuring forward and backward privacy?* We tackle the problem in this work by proposing a novel DSSE scheme named HeX based on TEE. The biggest advantage of introducing the enclave is that it can relieve the burden of the client by securely interacting with the malicious server on his behalf. Nevertheless, it is still demanding to achieve all goals with such a limited memory size (128MB for Intel SGX) in the enclave. First of all, we adopt the bitmap, which is a vector of bits, as the underlying index form for keywords. One reason is that the bitmap can easily support arbitrary Boolean queries due to its intrinsic bitwise operations. Another advantage is that it can upgrade the search efficiency since only one index entry will be accessed. Additionally, it can record the deletion by setting the bit to 0 without deleting any index on the server. This is very useful when eliminating the leakage of which deletion cancels which insertion since the bitmap is encrypted on servers. However, the usage of bitmap will compromise the forward privacy since all updates will point to the same index entry. Hence, we design a deferred obfuscating technique atop the enclave to break the association between the read index and the index of real needs. In order to support range queries, we employ the idea of an encryption scheme named SORE to transform a range query into a disjunctive query of several keyword queries. Finally, the experiments show that the search efficiency of HeX is up to two orders of magnitude higher than existing designs. In general, our contributions can be summarized as follows:

- We present the basic protocols of our HeX scheme that provide single keyword-file queries based on the enclave. It achieves strong forward and backward privacy by obfuscating the read and write of accessed index entries.
- We extend the basic scheme to support both Boolean queries and range queries for the first time. The Boolean query is converted into bitwise operations of bitmap indexes while the range query is transformed into a disjunctive query of several keyword searches.
- We give formal definitions of leakage and conduct rigorous theoretical analysis on the security of our HeX, including the basic scheme and its extension.
- We implement the prototype of HeX and conduct extensive evaluations. The results demonstrate the feasibility and high search efficiency.

The remainder of this paper is organized as follows. We first review related work in Section 2 and introduce background knowledge in Section 3. We then present the system model in Section 4 and elaborate on our design in Section 5. We further perform security analysis in Section 6 and give the evaluation results in Section 7. Section 8 finally concludes our paper.

2 Related Work

2.1 Forward and Backward Private DSSE without TEE

Forward and backward privacy are proposed to profile the extra leakage when static SSE evolves to DSSE [5]. Bost [7] formally defines the forward privacy and gives an efficient instantiation of forward private SSE scheme. A follow-up work [4] comprehensively studies the notion of backward privacy and gives formal definitions for different levels of backward privacy according to their leakage extent. Type-III backward privacy leaks which deletion withdraws which insertion, and Type-II reveals the update time of keywords. Type-I backward privacy is the strongest notion and only tells the total number of updates. Typical Type-III backward private schemes include $\text{Diana}_{\text{del}}$, Janus [4] and Horus [37], while representative schemes of Type-II backward privacy are Fides [4] and Mitra [37]. Most Type-I schemes like Moneta [4] and Orion [37] rely on the ORAM technique which will incur prohibitive overhead. In [34], Xu et al. propose a practical DSSE scheme named ROSE that guarantees robustness, forward and backward privacy simultaneously. It can ensure the correctness of protocols in the case of misoperations. A recent work [35] designs a DSSE scheme with enhanced backward privacy by employing the k-anonymity method. Chen et al. [38] propose a novel scheme called Bamboo to thwart the post-compromise security of DSSE. It can not only achieve forward and backward privacy, but also guarantee data security even after the secret key is compromised.

2.2 Forward and Backward Private DSSE with TEE

Trusted Execution Environment (TEE), also known as enclave, such as Intel SGX, is capable of offering trust from the level of hardware. The memory (limited to 128MB) and computation in the enclave are isolated from external operating systems and softwares, thus guaranteeing the integrity and confidentiality of the enclave's data. The codes in the enclave can only be accessed and modified by the authorized user. He needs to establish a secure channel to the enclave and verify the target code is running as intended via remote attestation [39]. The enclave enables the client to securely execute programs even if the server is untrusted.

In recent years, TEE has been introduced to DSSE to achieve stronger security and higher efficiency. Amjad et al. [15] take the first step to give three constructions of forward and backward private DSSE schemes based on SGX,

TABLE 1: Comparison with representative DSSE schemes.

Schemes		Backward Privacy[a]	Forward Privacy[b]	Query Types		Leakage Protection		Against Malicious Server
				Boolean Query	Range Query	Access Pattern	Search Pattern	
Non-TEE Designs	ROSE [34]	Type-III	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$
	Fides [4]	Type-II	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$
	Zheng-Lu [35]	Type-I$^-$	$\surd$	$\surd$	$\times$	$\times$	$\surd$	$\times$
TEE-based Designs	HybrIDX [25]	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$	$\times$
	Bunker-A [15]	Type-III	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$
	SGX-SE1/2 [16]	Type-II	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$
	Bunker-B [15]	Type-II	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$
	BISEN [36]	Type-II	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\surd$
	Fort [15]	Type-I	$\surd$	$\times$	$\times$	$\surd$	$\surd$	$\times$
	Maiden [17]	Type-I	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$
	HeX	Type-0	$\surd^+$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

[a] Type-I$^-$ means that the scheme leaks less information than Type-I backward privacy. Type-0 is a new type of backward privacy that we define in Definition 5, which leaks the least information.
[b] $\surd^+$ indicates that it not only ensures the traditional forward privacy, but also guarantees strong forward search privacy defined in Definition 2.

including Bunker-A (Type-III), Bunker-B (Type-II) and Fort (Type-I). Later, Vo et al. [16] propose two efficient schemes of Type-II backward privacy schemes named SGX-SE1 and SGX-SE2. They both maintain the keyword states and file deletion in the enclave and avoid deleting the index on the server. SGX-SE2 optimizes SGX-SE1 by utilizing Bloom filter to directly judge the existence instead of reading the files. The two schemes show high efficiency in terms of update and search since they avoid the re-encryption operations as Bunker-B does. Their follow-up work [17] called Maiden further improves the level of backward privacy to Type-I by moving the counter state from the server to the enclave. However, these schemes are all limited to single keyword-file search. Ren et al. [25] design a new hybrid index named HybrIDX that can support encrypted range queries in a volume-hiding manner. It also obfuscates the access pattern by integrating a bulk refresh mechanism. Different from the semi-honest setting in the above schemes, Ferreira et al. assume the server is untrusted and propose BISEN [36] to ensure the data integrity returned by the server. It achieves forward and Type-II backward privacy with the support for Boolean queries. However, it needs to retrieve all related index (including deletion ones) for a keyword search, which degrades the search efficiency especially when the deletion percentage is high.

2.3 DSSE with Boolean Queries and Range Queries

Most prior works focus on the single keyword-file search, and only a few schemes work on Boolean queries or range queries. Yuan et al. [22] enable multi-client Boolean search over encrypted database using a distributed index framework. A novel DSSE scheme named ODXT is proposed in [21] to support conjunctive search on arbitrarily structured databases. It only needs a single round of communication for all operations and achieves both forward and backward privacy. Guo et al. [24] provide both exact-match and range-match queries over encrypted attribute-value data by designing a novel enhanced order-revealing encryption scheme. Zuo et al. [40] design a DSSE scheme using a refined binary tree for range queries with a new backward privacy named Type-R. There are few works that can support both Boolean queries and range queries while keeping forward and backward private.

Table 1 presents representative DSSE schemes achieving forward or backward privacy. We can see that our HeX achieves both stronger forward and backward privacy than existing schemes while supporting Boolean queries and range queries besides traditional single keyword-file search. It can also counter malicious servers who may deliberately deviate from the protocols.

3 Preliminaries

In this section, we introduce some preliminaries related to the proposed design. Table 2 lists the notations that are often used through our work.

3.1 Dynamic Searchable Symmetric Encryption (SSE)

A typical dynamic SSE scheme $\Sigma = \{Setup, Update, Search\}$ usually consists of three protocols between a client and a server.

- $Setup(\lambda, \mathtt{DB})$ protocol takes the security parameter λ and the initial database $\mathtt{DB}$ as input and outputs $(K, \sigma, \mathtt{EDB})$, where K is the secret key, σ is the client's local state, and $\mathtt{EDB}$ is the encrypted database maintained by the server.
- $Update(K, op, in, \sigma; \mathtt{EDB})$ protocol allows the client to update the plaintext database, thus modifying the encrypted database $\mathtt{EDB}$ on the server. The operation op can be either add or del, and in contains the keyword w and the file identifier id to be updated.
- $Search(K, w, \sigma; \mathtt{EDB})$ protocol enables the client to search a keyword w over $\mathtt{EDB}$ by generating query tokens using K and σ. The server will return the matching file identifiers to the client.

TABLE 2: Notations.

Symbol	Meaning
K	The secret key
σ	The local state
id	The unique identifier
w	The keyword
W	The keyword set
$\mathtt{DB} = \langle id, w \rangle$	The plaintext database of id, w pairs
l	The label index of the encrypted index
c	The cipher value of the encrypted index
$\mathtt{EDB} = \langle l, c \rangle$	The encrypted database of l, c pairs
op	The operation of add or del
t	The timestamp of an operation
$\mathcal{L}$	The leakage function
$\perp$	An empty set
KVS	The keyword version state map in the enclave
CK	The cached keyword map in the enclave
$\xleftarrow{\$}$	Randomly generating
$\xleftarrow[k]{\$}$	Randomly sampling k elements
λ	The security parameter

3.2 Forward Privacy

Forward privacy guarantees the updating keyword-document pair during the update operation cannot be related to any keyword that has been searched before. It was first informally defined in [5] and we follow the definition in [4].

Definition 1. *An $\mathcal{L}$-adaptively secure SSE scheme ensures forward privacy iff the update leakage function $\mathcal{L}^{update}$ can be written as:*

$$\mathcal{L}^{update}(op, w, id) = \mathcal{L}'(op, id),$$

where op, w, id are the operation, the keyword and the document ID to be updated respectively; and $\mathcal{L}'$ is stateless.

The above definition of forward privacy focuses on the leakage during the data update, so it can also be regarded as forward update privacy. A stronger security notion named forward search privacy, which further reduces the leakage during the search, was proposed in [8]. It blocks the association between the former search tokens and the current search over newly updated data. Formally, it is defined as follows:

Definition 2. *An $\mathcal{L}$-adaptively secure SSE scheme achieves forward search privacy iff the search leakage function $\mathcal{L}^{search}$ can be written as:*

$$\mathcal{L}^{search}(w) = \mathcal{L}'(\perp),$$

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

3.3 Backward Privacy

Backward privacy aims to ensure the search query on keyword w will not reveal any deleted document regarding w. Since the information of deletion can also be tracked during update, three types of backward privacy with different leakage functions were introduced in [4]. Type-I backward privacy is the strongest notion and it only has extra leakage of insertion pattern. In other words, besides the existing documents matching w and their insertion timestamps, it reveals the total number of updates on w. Type-II backward privacy additionally leaks timestamps of all updates regarding w and Type-III backward privacy further reveals which deletion update withdrew which insertion update. Here we borrow the definition of Type-I backward privacy. Let $\mathsf{TimeDB}(w)$ denote the list of all documents currently matching w along with their corresponding insertion timestamps. Formally, given a query list Q, $\mathsf{TimeDB}(w) = \{(t, id) \mid (t, \mathsf{add}, (w, id)) \in Q \text{ and } \forall t', (t', \mathsf{del}, (w, id)) \notin Q\}$, where t is the timestamp of the update operation of (w, id). Then we have the following definition.

Definition 3. *An $\mathcal{L}$-adaptively secure SSE scheme guarantees Type-I backward privacy iff the search and update leakage function $\mathcal{L}^{search}$, $\mathcal{L}^{update}$ can be written as:*

$$\mathcal{L}^{update}(op, w, id) = \mathcal{L}'(op),$$
$$\mathcal{L}^{search}(w) = \mathcal{L}''(\mathsf{TimeDB}(w), a_w),$$

where a_w is the number of all inserted entries matching w; and $\mathcal{L}'$ and $\mathcal{L}''$ are stateless.

3.4 Pattern Leakage

Access Pattern. The access pattern indicates the search result and accessed entries during a single query process. The leakage function of access pattern $\mathcal{L}^{ap}$ arising from a query q on w can be written as:

$$\mathcal{L}^{ap}(w) = \{op, t, \langle l, c \rangle \mid (w, \langle l, c \rangle) \in q\},$$

where $\langle l, c \rangle$ is the matching label-cipher pairs, op denotes its operation of update or search and t is the timestamp when it is updated. Since the label-cipher pairs are usually encrypted, the amount of matching pairs, i.e., volume pattern [25], becomes a more informative leakage for attackers to leverage.

Search Patterns. The search pattern captures the repeated queries pertaining to the same keyword among a list of queries Q. Specifically, the leakage function of search pattern $\mathcal{L}^{sp}$ on w can be written as:

$$\mathcal{L}^{sp}(w) = \{t \mid (t, w) \in Q\},$$

where t is the timestamp when the query on w is issued [4].

3.5 Succinct Order-Revealing Encryption (SORE) scheme

The SORE scheme was proposed in [41] to convert a range query to an exact matching query. It consists of three protocols, i.e., $Encrypt, Token, Compare$.

- $Encrypt(k, v)$ protocol takes the secret key k and the plaintext value v as input, and outputs ciphertexts. Specifically, for each bit index $i \in [1, b]$, where b is the number of v's bits, it will compute a tuple $ct_i \leftarrow v_{|i-1} \| \bar{v}_i \| cmp(\bar{v}_i, v_i)$. Here v_i is the ith bit of the v, $v_{|i-1}$ denotes the bits from 1 to $i-1$, $\bar{v}_i$ represents the opposite bit of v_i,

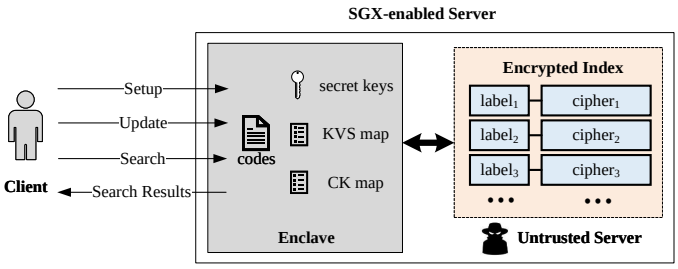


Fig. 1: System architecture.

$cmp(\bar{v}_i, v_i) \in \{">", "<"\}$ denotes the comparison result between $\bar{v}_i$ and v_i, and $\|$ means the concatenation. Finally, it shuffles all tuples and outputs their pseudo-random function (PRF) values as ciphertexts $ct = \{F_k(ct_1), F_k(ct_2), \cdots, F_k(ct_b)\}$.

- $Token(k, v, oc)$ protocol takes the queried value v and the order condition $oc \in \{">", "<"\}$ as input, and returns corresponding query tokens. It generates a tuple $tk_i \leftarrow v_{|i-1}\|v_i\|oc$ for each bit. At last, it shuffles all tuples and outputs their PRF values as tokens $tk = \{F_k(tk_1), F_k(tk_2), \cdots, F_k(tk_b)\}$.
- $Compare(ct, tk)$ protocol takes the ciphertexts ct and the query tokens tk as input, and returns whether they have one tuple in common. If the common tuple exists, ct matches tk and the algorithm outputs `True`. Otherwise, it outputs `False`.

4 System Model

4.1 System Architecture

Figure 1 depicts the system framework in our design which consists of two parties, i.e., a *client* and an SGX-enabled *server*. The client owns a plaintext database and outsources its encrypted version to the cloud server. He can later update the encrypted database or search over it via corresponding protocols. Our system only works under the *single-client* setting because the secret keys belong to only one client. We plan to support multiple clients by incorporating access control algorithms in future work. The server contains the trusted enclave and the untrusted host part. The client can establish a secure channel with the enclave and authenticate the codes of protocol programs via remote attestation. The enclave can also store some secret keys and local states so that it can represent the client to interact with the untrusted part of the server. Since the untrusted server is more powerful in terms of storage and memory than the enclave, it will host massive encrypted index for future operations. In essence, the index is a map of label-cipher pairs whose content is in an encrypted form. It is noted that this paper focuses on the index building and the search for file IDs, thus skipping the storage and retrieval of actual file documents like many other works [17], [34], [35], [38].

4.2 Threat Model

We consider the client and the enclave within the server to be honest, which means they will faithfully execute the default protocols. In addition, the secure communication channel (e.g., TLS) between the client and the enclave is secure, preventing potential attacks such as Man-in-the-Middle (MitM) attacks, session hijacking attacks, replay attacks, etc. But at the same time, we assume the server outside the enclave is untrusted, which is more strict than the semi-honest setting in many existing works [15], [16], [17]. First, the untrusted server may deviate from predefined protocols. The deviation may be due to the server's intentional response of incorrect or incomplete results, or unintentional consequences of software bugs and hardware failure. Therefore, the enclave is required to be capable of detecting incorrect and incomplete results from the untrusted server. Meanwhile, the server may also try to learn information by observing the addresses and data of read/write requests from the enclave.

4.3 Design Goals

In this paper, we aim to provide strong forward and backward private DSSE for rich queries on untrusted servers. Specifically, we have the following two design goals:

- We need to achieve forward search privacy and Type-0 backward privacy as defined in Definition 2 and Definition 5 respectively even when the server is untrusted.
- We want to provide rich queries, including Boolean queries and range queries, with high efficiency.

5 Our Proposed Scheme: HeX

In this section, we elaborate on the detailed design of HeX, including the underlying obfuscating techniques, the basic protocols for single keyword-file search and the extension for rich queries.

5.1 Design Rationale

The first point we consider is how to achieve strong backward privacy, which essentially entails the need to hide deletion operations. Some schemes choose to embed the deletion symbol into the update tokens and treat a deletion as an insertion [4], [15]. The deleted file IDs will be excluded during subsequent data searches. This method apparently degrades the search efficiency due to the processing of deletion. Like some other schemes [35], [42], here we adopt the bitmap index where a keyword owns a n-bit string recording whether each file ID contains itself for all n files. It not only conceals the information of which deletion cancels which insertion, i.e., the extra leakage in Type-III backward privacy, but also hides the volume pattern due to the constant size of the returned results. In addition, the bitmap intrinsically supports Boolean queries via its bit operations. Moreover, it can speed up the data search compared to the traditional traversal way since only one index entry needs to be processed.

Nonetheless, putting all existence marks of a keyword into one bitmap brings about other problems. First, the forward privacy cannot be ensured since all update and search requests will target the same index. Moreover, the number of updates and all update timestamps of each keyword are still revealed, which makes the level of backward privacy stop at Type-II backward privacy. Therefore, an obfuscating

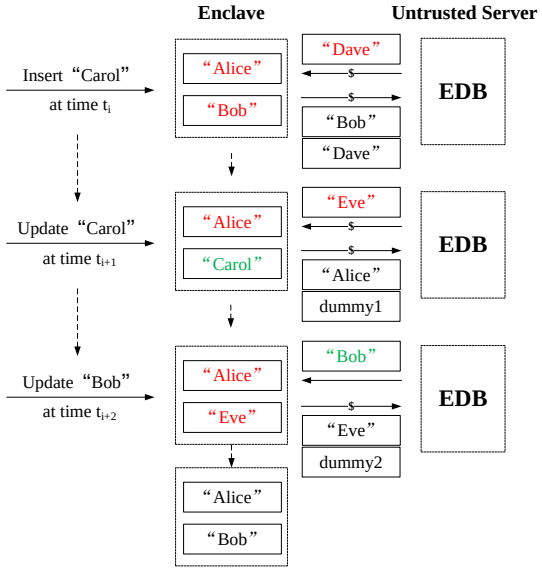


Fig. 2: Deferred obfuscating read-write during update.

technique is strongly required to disrupt the association between the accessed index and the actual index. In this way, the target keywords and the operation time of all searches and updates will no longer be deterministic as in conventional schemes, thus guaranteeing the forward privacy and stronger backward privacy.

5.2 Deferred Obfuscating Read-Write Techniques

An obfuscating technique based on the k-anonymity method was proposed in [35] to hide the search pattern. However, since it writes back the read entry immediately after the processing, it will reveal the written time the next time this entry is read. Another attempt is made to obfuscate the access pattern by the enclave cache and a bulk refresh mechanism in [25]. This approach makes good use of the enclave, but still leaks whether the accessed entries exist in the cache. Different from the above works, we defer the write of the currently accessed index entry to any possible time afterward by caching it in the enclave. Specifically, the cached keyword will be probabilistically selected for flushing out only when it is not requested during the subsequent operations. Therefore, the requested keyword can be largely irrelevant to the operated index entry. To further hide whether the accessed keyword is cached, we will still read a random entry outside the enclave even when the keyword is cached. Similarly, during the update, the number of written entries needs to keep constant by padding regardless of the operation type, so the read and write patterns can remain the same for data insertion or update. We illustrate our deferred obfuscating read-write techniques during the update and search in Figure 2 and Figure 3 respectively. To ease the understanding, we set the cache size to 2 in the descriptions. But in practice, we can enlarge the cache to upgrade the obfuscation according to the security parameter λ.

As shown in Figure 2, there is a constant-size cache in the enclave and an EDB containing most index entries outside the enclave. At time t_i, only keywords 'Alice' and 'Bob' are cached, but an insertion request for 'Carol' comes. Since

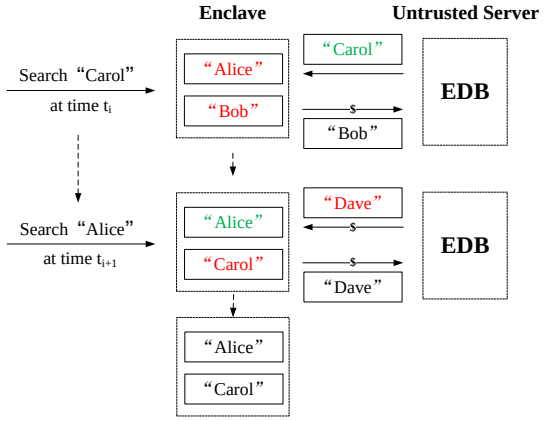


Fig. 3: Deferred obfuscating read-write during search.

Algorithm 1 Setup

Input: Security parameter λ; initial database DB. **Client**:

1: Generate a PRF key $K \xleftarrow{\$} \{0,1\}^\lambda$;
2: Establish a secure channel to the enclave and validate the program codes via remote attestation;
3: Send K and DB to the enclave;

Enclave:

4: Initialize an empty keyword version state map KVS, an empty cached keyword map CK and a temporary encrypted index map EDB.
5: $w_1, w_2 \xleftarrow[2]{\$} \{w\}$; ▷ random keywords for cache
6: **for each** w **do**
7: $vn_w \leftarrow 0$; $bm_w \leftarrow 0^D$;
8: **for each** $id \in \mathtt{DB}(w)$ **do**
9: $bm_{w|id} \leftarrow 1$;
10: **end for**
11: **if** $w \in \{w_1, w_2\}$ **then**
12: $\mathtt{CK}.Put\,(w, bm_w)$;
13: $\mathtt{KVS}.Put\,(w, vn_w)$;
14: **else**
15: $label \leftarrow F\,(K, w\|vn_w + 1\|0)$;
16: $eKey \leftarrow F\,(K, w\|vn_w + 1\|1)$;
17: $cipher \leftarrow Enc\,(eKey, bm_w)$;
18: $\mathtt{EDB}.Put\,(label, cipher)$;
19: $\mathtt{KVS}.Put\,(w, vn_w + 1)$;
20: **end if**
21: **end for**
22: Send EDB to the server and then delete EDB;

Server:

23: Store EDB.

'Carol' is inserted for the first time, the protocol produces a new bitmap for it and stores it in the cache. Although the protocol does not need any data in EDB, it still needs to pretend to read from the untrusted server. Specifically, the enclave will retrieve a random keyword, namely 'Dave', together with its bitmap from EDB and put it into the cache. Here $ on the arrow means the related item(s) is(are) randomly selected. The keyword marked in red denotes that it mismatches the requested keyword, while the green means matching.

Now the cache has three keywords excluding the ac-

Algorithm 2 Update

Input: PRF key K; op, w, id to update; keyword version state KVS; cached keywords CK; encrypted database EDB.

Client:

Send op, w, id to the enclave;

Enclave: ▷ obfuscated read

if $w \in \texttt{CK}.keys()$ **then** ▷ cached
  $w_{in} \xleftarrow[1]{\$} \texttt{KVS}.keys() \setminus \texttt{CK}.keys()$;
  $vn_{w_{in}} \leftarrow \texttt{KVS}.Get(w_{in})$;
else if $w \in \texttt{KVS}.keys()$ **then** ▷ exists but not cached
  $w_{in} \leftarrow w$; $vn_{w_{in}} \leftarrow \texttt{KVS}.Get(w_{in})$;
else ▷ first update
  if $op ==$ del **then return** error;
  end if
  $w_{in} \xleftarrow[1]{\$} \texttt{KVS}.keys() \setminus \texttt{CK}.keys()$;
  $vn_{w_{in}} \leftarrow \texttt{KVS}.Get(w_{in})$;
end if
$label \leftarrow F(K, w_{in} \| vn_{w_{in}} \| 0)$;
$eKey \leftarrow F(K, w_{in} \| vn_{w_{in}} \| 1)$;
Send $label$ to the server;

Server:

$cipher \leftarrow \texttt{EDB}.Pop(label)$;
Send $cipher$ to the enclave;

Enclave:

if $cipher == \perp$ **or** $Dec(eKey, cipher) == \perp$ **then**
  Abort; ▷ verification fails
end if
$bm_{in} \leftarrow Dec(eKey, cipher)$;
if $w \in \texttt{CK}.keys()$ **then** ▷ cached
  $\texttt{CK}.Put(w_{in}, bm_{in})$;
  $bm_w \leftarrow \texttt{CK}.Get(w)$;
  UpdateBitmap (bm_w);
  $\texttt{CK}.Put(w, bm_w)$;
  $\texttt{CK}.Put(dummy, \perp)$; $\texttt{KVS}.Put(dummy, 0)$;
else if $w \in \texttt{KVS}.keys()$ **then** ▷ exists but not cached
  UpdateBitmap (bm_{in});
  $\texttt{CK}.Put(w, bm_{in})$;
  $\texttt{CK}.Put(dummy, \perp)$; $\texttt{KVS}.Put(dummy, 0)$;
else ▷ first update
  $\texttt{CK}.Put(w_{in}, bm_{in})$;
  $vn_w \leftarrow 0$; $bm_w \leftarrow 0^D$;
  UpdateBitmap (bm_w);
  $\texttt{CK}.Put(w, bm_w)$; $\texttt{KVS}.Put(w, vn_w)$;
end if
$\texttt{EDB}_{out} \leftarrow \phi$;
$w_{out1}, w_{out2} \xleftarrow[2]{\$} \texttt{CK}.keys() \setminus \{w\}$;
for each w_{out} **do** ▷ obfuscated write
  $vn_{w_{out}} \leftarrow \texttt{KVS}.Get(w_{out})$;
  $label \leftarrow F(K, w_{out} \| vn_{w_{out}} + 1 \| 0)$;
  $eKey \leftarrow F(K, w_{out} \| vn_{w_{out}} + 1 \| 1)$;
  $bm_{w_{out}} \leftarrow \texttt{CK}.Pop(w_{out})$;
  $cipher \leftarrow Enc(eKey, bm_{w_{out}})$;
  $\texttt{EDB}_{out}.Put(label, cipher)$;
  $\texttt{KVS}.Put(w_{out}, vn_{w_{out}} + 1)$;
end for
Send $\texttt{EDB}_{out}$ to the server;

Server:

Add $\texttt{EDB}_{out}$ into EDB;

cessed 'Carol', i.e., 'Alice', 'Bob', and 'Dave', among which it will randomly choose two to remove and write back to EDB after re-encryption. Hence, at the next timestamp t_{i+1}, only 'Alice' and 'Carol' are cached while an update request on 'Carol' arrives. Because 'Carol' is already for update, besides modifying the bitmap of 'Carol', we also need to randomly pick one keyword in the local state and read its information from EDB into the cache. To mimic the extra two keywords generated in the case of the first insertion at time t_i, we pad the enclave with a dummy keyword along with a dummy bitmap. Finally, the enclave will get two extra items, i.e., 'Eve' and dummy1, and then randomly write back two out of the cache. At time t_{i+2}, the cached keywords become 'Alice' and 'Eve', and the client requests to update 'Bob'. Since 'Bob' is not cached, we will retrieve its bitmap from EDB and update it. Similarly, one dummy entry will be padded and two entries will be transferred back to the untrusted server.

The obfuscating mechanism during the search in Figure 3 resembles the update operation above. Since the search operation will not produce extra entries like that during the update, it only needs to read one keyword in and write one out without dummy padding. If the queried keyword is cached, then a random keyword will be read, otherwise the keyword itself is retrieved. Detailed pseudo-codes of our obfuscating techniques are presented in the following subsection.

5.3 Basic Protocols

We denote the security parameter as λ and the PRF key as K. Let n be the total number of files and id be a file identifier where $id = 1, 2, \cdots, n$, then a keyword w's bitmap can be denoted as $bm_w = b_{w,1} b_{w,2} \cdots b_{w,n}$, where $b_{w,i} \in \{0, 1\}$ represents whether w exists in the file with ID i. 1 means existence and 0 is non-existence.

Setup. Algorithm 1 shows the Setup protocol to bootstrap HeX. The client will establish a secure channel to the enclave, authenticate it using remote attestation, and share his PRF key with it. The enclave also needs to build an initial encrypted database EDB based on the client's plaintext database DB. The initial EDB will finally be transferred from the enclave to the server. In specific, the algorithm produces a bitmap bm_w for each keyword w and sets its value according to the IDs of files containing it. Additionally, each w also has his version number vn_w whose initial value is 0. If w is chosen as the cached keyword, it will be stored into the cached keyword map CK together with its bm_w. The keyword w and its vn_w are put into the keyword version state map KVS since it will not be written to the encrypted index. If w will not be cached, it needs to generate a $< label, cipher >$ pair which will be later put in EDB and sent to the untrusted server for storage. The

Algorithm 3 UpdateBitmap

Input: op, w, id to update; w's bitmap bm_w.
Output: Updated bitmap bm_w.

1: **if** $op ==$ add **then**
2: **if** $bm_{w|id} == 0$ **then** $bm_{w|id} \leftarrow 1$
3: **else return** error;
4: **end if**
5: **else** ▷ delete
6: **if** $bm_{w|id} == 1$ **then** $bm_{w|id} \leftarrow 0$
7: **else return** error;
8: **end if**
9: **end if**

$label$ is computed via the PRF on $w\|vn_w + 1\|0$ with key K, where $\|$ is the concatenation symbol. A one-time symmetric encryption key $eKey$ is also calculated using the PRF on $w\|vn_w + 1\|1$. The value of $cipher$ is the bm_w encrypted by $eKey$. Finally, vn_w is incremented and recorded in KVS.

Update. As shown in Algorithm 2, the client will send the operator $op \in \{\mathsf{add}, \mathsf{del}\}$ along with w, id at the beginning of Update protocol. The enclave follows the deferred obfuscating technique for update as described in Section 5.2, and generates a keyword w_{in} that needs to be read in. The ciphertext of w_{in} will be retrieved by the server using its label. Note that the matching label-cipher pair will be removed by the server after the retrieval. Since the server is regarded as untrusted, the returned $cipher$ should be validated before processing (see Line 18-19). If $cipher$ is empty or the decrypted result via $eKey$ is empty, then the server is considered to return incomplete or incorrect data thus failing the verification.

After the verification, the enclave will update the bitmap for the requested w using UpdateBitmap in Algorithm 3. The main purpose of this algorithm is to prevent duplicate insertion and duplicate deletion by reporting errors. If the requested operation is valid, the corresponding bit of the bitmap will be set to the desired value. Eventually, two keywords will be randomly selected and written back into the server's EDB.

Search. As presented in Algorithm 4, the Search protocol is similar to Update except for the write-back part and the case where the requested keyword has not been seen before. After decrypting the ciphertexts from the server, the client can readily obtain the search result via the bitmap bm_w. The write-back step follows the description of the obfuscating scheme for search in Section 5.2, where only one entry needs to be put back. The bitmap update and dummy padding operations in Update are no longer required.

Moreover, during the data search, if an unseen keyword is received (see Line 7), the enclave can easily discover its non-existence by checking the keyword version state KVS. Instead of reporting the exception and aborting, we choose to select a random dummy keyword used by the obfuscating scheme to proceed with the search as normal. The client will still get an empty search result eventually. This trick can prevent the server from learning whether the sent keyword exists in DB, which is important to the extension for range queries in the next subsection.

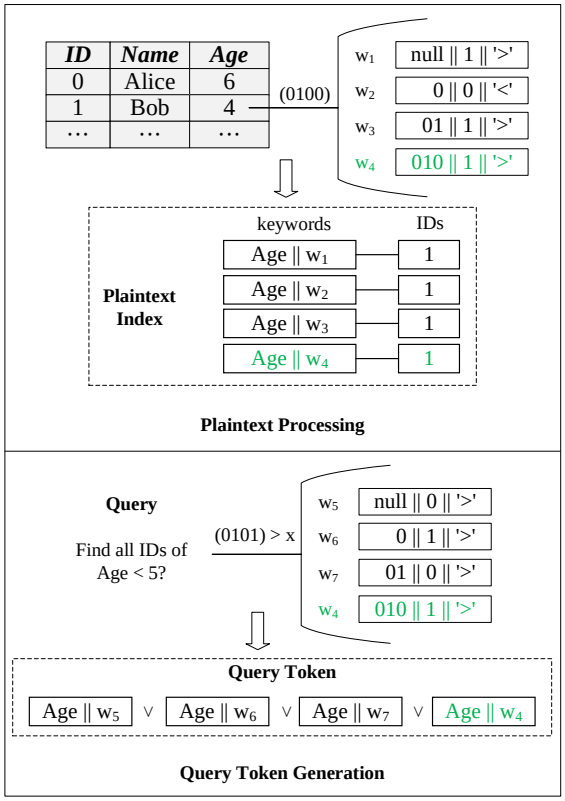


Fig. 4: A range query example.

5.4 Extensions for Rich Queries

Boolean Queries. Since the bitmap records the information of whether the keyword exists in each file, our HeX can be easily extended to support Boolean queries, i.e., conjunction, disjunction and negation, by applying bitwise 'AND', 'OR' and 'NOT' to corresponding bitmaps. For instance, suppose there is a Boolean query with the formula $\psi = w_1 \vee w_2 \wedge \neg w_3$. We first need to invoke the basic Search protocol for all keywords w_1, w_2, w_3 and get their bitmaps, denoted as bm_1, bm_2, bm_3 respectively. Then the query result of ψ will be the corresponding bitwise operations of these three bitmaps, i.e., $Q(\psi) = bm_1|bm_2\&\sim bm_3$.

Range Queries. Range queries are usually required over multiple-attribute data with numerical values. Figure 4 shows an example of a range query. In the table, each entry has a unique ID, a string attribute 'Name' and a numerical attribute 'Age'. To enable range query on the 'Age' attribute, we first employ the core idea of the SORE scheme to slice the number into several tuples. Taking the second entry as an instance, we generate four plaintext tuples w_1, w_2, w_3, w_4 for the number 4. Then we concatenate the attribute name with the tuples, and directly use them as the keywords in the basic protocols. The calculation of PRF in the original SORE scheme is omitted here since our basic protocols have already provided it.

Suppose now we have a range query to find all IDs whose 'Age' is smaller than 5. We can follow the SORE scheme to produce four tuples as well, i.e., w_5, w_6, w_7, w_4. It is noted that w_4 is a common tuple shared by the query and the plaintext, which means the plaintext entry 4 is one of the answers. At last, the range query will become a

Algorithm 4 Search

Input: PRF key K; op, w, id to search; keyword version state KVS; cached keywords CK; encrypted database EDB.

Client:

Send op, w, id to the enclave;

Enclave: ▷ obfuscated read

if $w \in \mathtt{CK}.keys()$ **then** ▷ cached
  $w_{in} \xleftarrow[1]{\$} \mathtt{KVS}.keys() \setminus \mathtt{CK}.keys()$;
  $vn_{w_{in}} \leftarrow \mathtt{KVS}.Get(w_{in})$;
else if $w \in \mathtt{KVS}.keys()$ **then** ▷ exists but not cached
  $w_{in} \leftarrow w$; $vn_{w_{in}} \leftarrow \mathtt{KVS}.Get(w_{in})$;
else ▷ not exists
  $w_{in} \leftarrow dummy$;
end if
$label \leftarrow F(K, w_{in} \| vn_{w_{in}} \| 0)$;
$eKey \leftarrow F(K, w_{in} \| vn_{w_{in}} \| 1)$;
Send $label$ to the server;

Server:

$cipher \leftarrow \mathtt{EDB}.Pop(label)$;
Send $cipher$ to the enclave;

Enclave:

if $cipher == \perp$ **or** $Dec(eKey, cipher) == \perp$ **then**
  Abort; ▷ verification fails
end if
if $w \in \mathtt{CK}.keys()$ **then** ▷ cached
  $bm_{in} \leftarrow Dec(eKey, cipher)$;
  $\mathtt{CK}.Put(w_{in}, bm_{in})$;
  $bm_w \leftarrow \mathtt{CK}.Get(w)$;
  $w_{out} \xleftarrow[1]{\$} \mathtt{CK}.keys() \setminus \{w\}$;
else if $w \in \mathtt{KVS}.keys()$ **then** ▷ exists but not cached
  $bm_w \leftarrow Dec(eKey, cipher)$;
  $\mathtt{CK}.Put(w, bm_w)$;
  $w_{out} \xleftarrow[1]{\$} \mathtt{CK}.keys() \setminus \{w\}$;
end if
Send bm_w to the client;
$\mathtt{EDB}_{out} \leftarrow \phi$; ▷ obfuscated write
$vn_{w_{out}} \leftarrow \mathtt{KVS}.Get(w_{out})$;
$label \leftarrow F(K, w_{out} \| vn_{w_{out}} + 1 \| 0)$;
$eKey \leftarrow F(K, w_{out} \| vn_{w_{out}} + 1 \| 1)$;
$bm_{w_{out}} \leftarrow \mathtt{CK}.Pop(w_{out})$;
$cipher \leftarrow Enc(eKey, bm_{w_{out}})$;
$\mathtt{EDB}_{out}.Put(label, cipher)$;
$\mathtt{KVS}.Put(w_{out}, vn_{w_{out}} + 1)$;
Send $\mathtt{EDB}_{out}$ to the server;

Client:

Extract ids using bm_w;

Server:

Add $\mathtt{EDB}_{out}$ into EDB;

Boolean query finding the disjunction of four keywords, i.e., 'Age$\|w_5$', 'Age$\|w_6$', 'Age$\|w_7$', 'Age$\|w_4$'. Some more complex range queries like finding x that satisfies $1 < x < 6$, can be readily solved by the conjunction of $1 < x$ and $6 > x$.

It is worth noting that some query tuples generated by the SORE scheme do not exist in the plaintext tuples, e.g., w_5 and w_7 will never appear in the plaintexts in the above example. This is because during the generation of plaintext tuples, the second concatenated element '0' is never followed by '>', only by '<'. Therefore, if we skip the search when facing an unseen keyword, then the server can directly get how many valid tuples are produced by the query, which is an additional leakage that may be exploited by attackers. For instance, still in the example, given a valid tuple w_4 from the query, it can be inferred that there is at least one '1' in the binary form of the queried value.

6 Security and Complexity Analysis

In this section, we analyze the security and the complexity of our proposed HeX, including the basic protocols and extended schemes for rich queries.

6.1 Correctness of Data Verification

The correctness verification lies in the enclave's data retrieval from the server during the update (Line 18-19 in Algorithm 2) and search (Line 15-16 in Algorithm 4). We give the formal theorem as follows:

Theorem 1. HeX *guarantees the correctness of data loaded into the enclave if F is a secure PRF, and (Enc, Dec) is CPA-secure.*

Proof. We consider three situations that may occur when the malicious server wants to undermine the data correctness. In all these cases, our scheme can detect and report the cheating behavior of the server. 1) If the server returns an empty cipher to the enclave, the enclave can easily tell because the requested keyword must exist in EDB if it is in its keyword version state KVS. 2) If the server returns a forged cipher or a stale cipher that is no longer valid, the enclave will not be able to decrypt it using the latest encryption key, and the verification fails. 3) If the server returns a cipher that can be successfully decrypted but actually encrypted from another plaintext, it must break the security of F or (Enc, Dec) because the cipher is produced only based on them. This contradicts the assumption that F is a secure PRF and (Enc, Dec) is CPA-secure. □

6.2 Security of Basic Protocols

We first give the formal definitions of leakage functions generated by our protocols and further prove the desired security notions. After the enclave establishes the encrypted label-cipher pairs, we have the leakage function of Setup defined as follows:

$$\mathcal{L}^{setup}(\mathtt{DB}) = \mathcal{L}'(|l|, |c|, |\mathtt{EDB}|),$$

where $|l|, |c|$ are the lengths of $label$ and $cipher$ in each index, and $|\mathtt{EDB}|$ is the number of encrypted index. Then we give the leakage functions of our Update and Search protocols, i.e., $\mathcal{L}^{update}$, $\mathcal{L}^{search}$, as follows:

$$\mathcal{L}^{update}(op, w, id) = \mathcal{L}'(\perp),$$
$$\mathcal{L}^{search}(w) = \mathcal{L}''(\perp).$$

In other words, the update and search operations in our scheme will not leak anything. Based on the above leakages and the simulation proof technique, we present the following security definition and theorem:

Definition 4. *Let* HeX $=$ (Setup, Update, Search) *be our SSE scheme, and let* $\mathcal{L}^{setup}$, $\mathcal{L}^{update}$ *and* $\mathcal{L}^{search}$ *be the leakage functions. For a probabilistic polynomial-time (PPT) adversary* $\mathcal{A}$ *and a PPT simulator* $\mathcal{S}$*, we define two games* $\mathbf{Real}_{\mathcal{A}}(\lambda)$ *and* $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ *as follows:*

$\mathbf{Real}_{\mathcal{A}}(\lambda)$*:* $\mathcal{A}$ *chooses a dataset* DB *and asks the client to build encrypted index* EDB *via* Setup*. Next,* $\mathcal{A}$ *repeatedly sends a polynomial number of requests for data update or search by running* Update *or* Search *protocols. Eventually,* $\mathcal{A}$ *observes the transcripts produced by two protocols and returns a bit as the output of the game.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$*:* $\mathcal{A}$ *chooses a dataset* DB*, and* $\mathcal{S}$ *builds simulated index based on the given leakage* $\mathcal{L}^{setup}$*. Next,* $\mathcal{A}$ *repeatedly sends the same requests for data update or search as in the real game. In response to these requests,* $\mathcal{S}$ *performs the update or search operations based on the predefined leakage* $\mathcal{L}^{Update}$ *or* $\mathcal{L}^{search}$*. Finally,* $\mathcal{A}$ *observes the simulated transcripts generated by* $\mathcal{S}$ *and returns a bit as the output of the game.*

We say HeX *is adaptively secure with (*$\mathcal{L}^{setup}$, $\mathcal{L}^{update}$, $\mathcal{L}^{search}$*) leakages if for all PPT adversaries* $\mathcal{A}$*, there exists a PPT simulator* $\mathcal{S}$ *such that:* $\Pr[\mathbf{Real}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda) = 1] \leq \mathrm{negl}(\lambda)$*, where* $\mathrm{negl}(\lambda)$ *denotes a negligible function in* λ*.*

Theorem 2. *Assuming all data and code execution in the enclave and the communication between the client and the enclave are secure,* HeX *is adaptively secure with (*$\mathcal{L}^{setup}$, $\mathcal{L}^{update}$, $\mathcal{L}^{search}$*) if* F *is a secure PRF, and* (Enc, Dec) *is CPA-secure.*

Proof. We now prove the above theorem by sketching the execution of a simulator $\mathcal{S}$ that makes all adversaries distinguish $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ at a negligible probability. It can be described in terms of the following three operations:

- At the setup phase, $\mathcal{S}$ generates the simulated index using $\mathcal{L}^{setup}$ and gives them to the server. Specifically, $\mathcal{S}$ produces a simulated index map, denoted as $\mathtt{EDB}'$, which has the size of $|\mathtt{EDB}|$ and each pair inside has the length of $\langle |l|, |c| \rangle$. Due to the pseudorandomness of PRFs and the semantic security of symmetric encryption, $\mathcal{A}$ cannot distinguish $\mathtt{EDB}'$ from the real EDB.
- Given a data update request of (op, w, id), $\mathcal{S}$ randomly selects and removes an index entry in $\mathtt{EDB}'$ and sends its label to the server. After receiving the cipher from the server, $\mathcal{S}$ randomly generates two new index entries and returns them to the server. Finally, $\mathcal{S}$ tracks the state of the newly updated $\mathtt{EDB}'$ in the server. Owing to the secure PRF and encryption scheme adopted by the index generation of $\mathbf{Real}_{\mathcal{A}}(\lambda)$, the real transcripts are indistinguishable from the simulated ones in $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$.
- Given a data search request on w, $\mathcal{S}$ randomly chooses and deletes an index entry from $\mathtt{EDB}'$ and gives its label to the server. After getting the corresponding cipher, $\mathcal{S}$ generates one random index entry and delivers it to the server. Eventually, $\mathcal{S}$ keeps the newest $\mathtt{EDB}'$ as the server does. Due to the same reason in the update process, the real transcripts are also indistinguishable from the simulated ones.

Since $\mathcal{A}$ cannot distinguish the real transcripts in $\mathbf{Real}_{\mathcal{A}}(\lambda)$ from the simulated transcripts in $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$, our scheme HeX is adaptively secure with ($\mathcal{L}^{setup}$, $\mathcal{L}^{update}$, $\mathcal{L}^{search}$).

□

We note that adversaries $\mathcal{A}$ will see the read index entry and its timestamp during data update and search, but we do not include this in the corresponding leakage functions as [35] does. This is because the update timestamp of the read index entry is also random in our scheme thanks to the deferred obfuscating read-write techniques. This entry can be any index from the original version at Setup or new versions during Update and Search at any timestamp before the current operation. This ensures that $\mathcal{L}^{update}$ and $\mathcal{L}^{search}$ are both empty, and thus hiding the access pattern and search pattern.

Theorem 3. *Our SSE scheme* HeX *achieves both forward update privacy and forward search privacy.*

Proof. Since our $\mathcal{L}^{update}$ and $\mathcal{L}^{search}$ are both empty, which satisfy the definitions of forward update privacy and forward search privacy in Definition 1 and Definition 2 respectively, the above theorem apparently holds. □

Since our scheme achieves stronger security than Type-I backward privacy in Definition 3, which is the strongest among existing notions, we define a new backward privacy notion named *Type-0 backward privacy* as follows.

Definition 5. *An* $\mathcal{L}$*-adaptively secure SSE scheme guarantees Type-0 backward privacy iff the search and update leakage function* $\mathcal{L}^{search}$, $\mathcal{L}^{update}$ *can be written as:*

$$\mathcal{L}^{update}(op, w, id) = \mathcal{L}'(\perp),$$
$$\mathcal{L}^{search}(w) = \mathcal{L}''(\perp),$$

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

Theorem 4. *Our SSE scheme* HeX *achieves Type-0 backward privacy.*

Proof. Based on our $\mathcal{L}^{update}$ and $\mathcal{L}^{search}$ functions and the definition of Type-0 backward privacy in Definition 5, the above theorem is clearly correct. □

6.3 Security of Extended Queries

Boolean Queries. The leakage of the Boolean query in HeX is the number of keywords in the Boolean formula. Here we assume all keywords indeed exist in DB. Let ψ_W denote the set of keywords in a Boolean query ψ, and then formally we have the following leakage function:

$$\mathcal{L}^{search}_{Boolean}(\psi) = \mathcal{L}'(|\psi_W|),$$

where $\mathcal{L}'$ is stateless. It is worth noting that the specific operators within the formula will not be leaked since the formula and its calculations are all inside the enclave. Only a small modification is required based on the basic proof to prove the security of Boolean queries formally. The ideal game $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ and the real game $\mathbf{Real}_{\mathcal{A}}(\lambda)$ need to conduct the search for $|\psi_W|$ times and $\mathcal{A}$ still cannot distinguish them.

Range Queries. The security analysis of the range query is almost the same as the Boolean query since a range query is finally converted into a Boolean query consisting of a number of keywords. Although the query tuples produced by a range query may not exist in the database, we still mimic the search for them. Hence, the number of keywords involved in the search is constant and equals the bit number of value, denoted as $|v|$. Formally, we have the following leakage function for the range query:

$$\mathcal{L}^{search}_{range}(Q) = \mathcal{L}'(|v|).$$

The bit number $|v|$ here is similar to the size of the keyword set $|\psi_W|$ in the Boolean query. The formal proof can also be achieved by letting the games run the search for $|v|$ times.

6.4 Complexity Analysis

For a deeper understanding of the efficiency of our scheme, we also present the complexity analysis in terms of time, communication and space. As mentioned, the operations of data insertion and data update are executed in the same manner, so the time complexity of these operations is $O(n)$, where n is the total number of files. Their communication complexity is also $O(n)$ since the transferred bitmap is n bits long. The enclave holds a cache maintaining λ index entries and a keyword version state consisting of $|W|$ items, so the space complexity of the enclave is $O(\lambda n + |W|)$. Outside the enclave, the server needs to store $|W|$ index entries, thus the space cost of the server is $O(|W|n)$. With respect to the data search, a basic single keyword-file search still needs $O(n)$ time and communication. However, it is concretely efficient because each file is represented by one bit and the time involved is the decryption of the bitmap instead of retrieving the same number of index entries. As for the Boolean queries and range queries, their corresponding complexity needs to multiply $|\psi_W|$ and $|v|$, respectively.

6.5 More Discussion

Side-channel Attacks towards TEEs. In the threat model, we regard the enclave as a trusted party, which means it will not actively leak any information about its memory and execution. However, malicious servers can still learn some information by observing the enclave memory access [43] or control flow branches [44]. Despite the above vulnerabilities, we still skip these side-channel attacks towards the TEE in this paper, since they have been well addressed by other works like [45], [46]. Their oblivious primitives can be directly equipped to enhance our system.

Leakage-Abuse Attacks. Some recent studies investigate the vulnerabilities to leakage-abuse attacks (LLAs) in existing searchable encryption schemes [47], [48], [49]. Although the forward and backward privacy can shrink the leakages to a constrained range, they are still insufficient to prevent potential LLAs. By exploiting the leakage profiles, e.g., the response length and the co-occurrence pattern, LLAs enable attackers with background knowledge to recover the queries or reconstruct the plaintext database. Fortunately, our system can withstand these attacks because the deterministic leakages that LLAs can utilize are eliminated by the obfuscating schemes. During each operation, the accessed index entry can belong to any keyword so that LLAs can barely infer from the observed associations.

TABLE 3: Statistics of datasets

Name	# of keywords	# of files	# of keyword-file pairs
SD	1,000	30,000	356,833
Enron	10,960	10,000	523,492

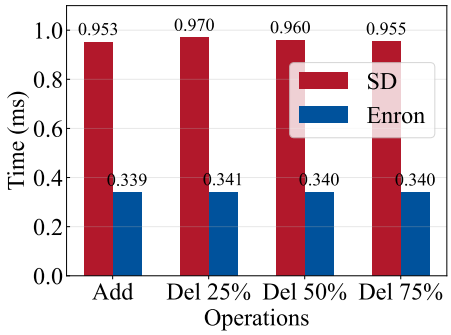


Fig. 5: Average Time Cost of Add or Delete one Keyword-File pair in HeX.

7 Implementation and Evaluation

We implement our HeX scheme based on Intel SGX SDK and C++17, and compare it with two state-of-the-art TEE-based SSE schemes, i.e., Maiden [17] and SGX-SE2 [16]. BISEN [36] is also a Type-II backward private scheme based on TEE but lacks the deletion processing as SGX-SE2 does. Thus, we only contrast BISEN's performance in terms of Boolean queries with our scheme. To better understand the performance of range queries, we also implement HybrIDX, which is a state-of-the-art DSSE scheme designed atop TEE for volume-hiding range queries. The codes of our implementation and evaluation are open-source and available on GitHub[1]. We adopt AES-128 as the underlying symmetric encryption and CMAC-128 for the pseudo-random function. The size of cached keywords in our HeX scheme is set to 10. In the performance evaluation of single keyword-file queries and Boolean queries, we select two datasets, including a synthetic dataset (SD) and a portion of the Enron email dataset[2]. The synthetic data is generated out of frequent English words[3] following Zipf's law distribution. The number details of the two datasets are listed in Table 3. As for the evaluation of the range query, we use 10,000 randomly generated numbers of 8, 16, 24 and 32 bits respectively. All these plaintext numbers follow the normal distribution within the

1. https://github.com/tripleday/HeX
2. https://www.cs.cmu.edu/~enron/
3. https://norvig.com/ngrams/

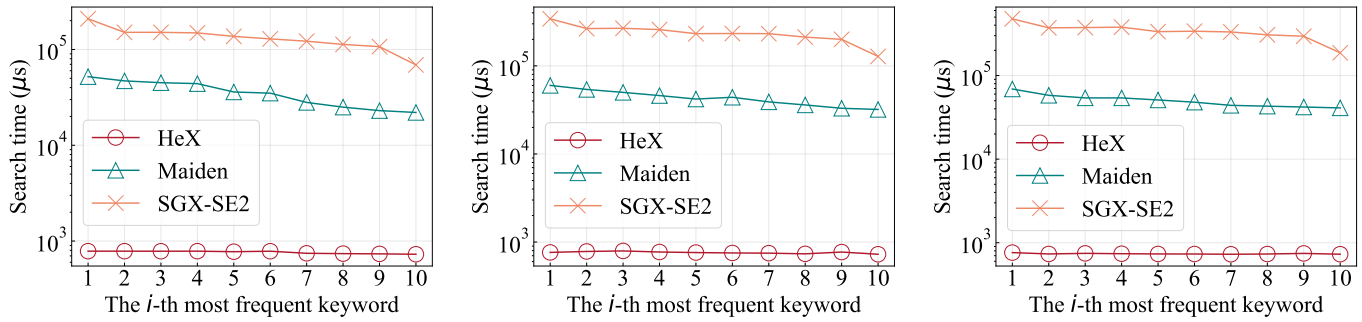


Fig. 6: Search Delay on SD.

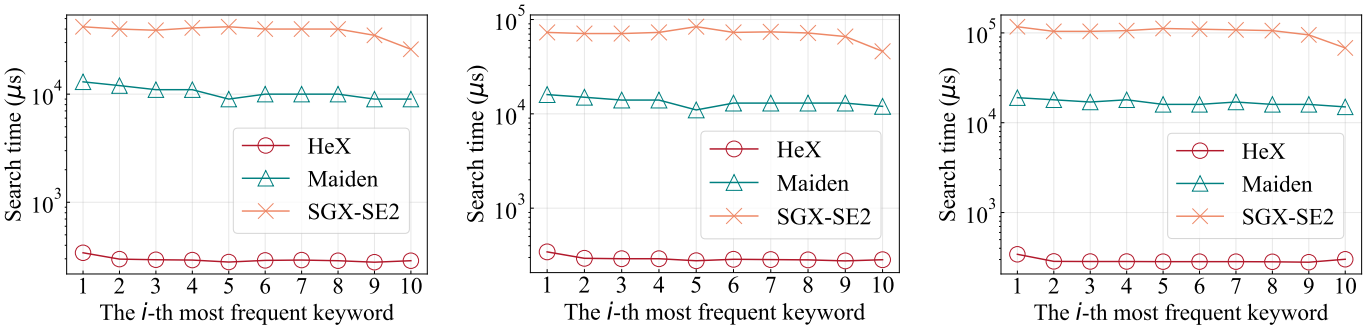


Fig. 7: Search Delay on Enron.

value field. We also randomly produce 1,000 range queries and average the search time as the result. The evaluation is conducted on an SGX-enabled cloud server equipped with a duo-core Intel Xeon Platinum 8374B 2.7GHz CPU, 16GB RAM and Ubuntu Server 18.04 LTS 64-bit OS.

7.1 Performance of Single Keyword-File Queries

Add and Delete Operations. To evaluate the time cost of insertion and deletion, we first insert all keywords into the server and delete a certain percentage of them after the insertion. Figure 5 records the average time cost it takes to add or delete a keyword-file pair. We can see that the time cost to add a pair is very close to that of deleting one in both datasets, because the process of deletion is almost the same as the insertion except for the bitmap update. The average time cost of SD is higher than that of Enron even though SD's number of keyword-file pairs is smaller than Enron's. This is because the key factor affecting the insertion and deletion efficiency in HeX is the number of files since the cipher length depends on the total file amount.

Search Delay. We evaluate the search delay of the most 10 frequent keywords over two datasets after deleting a portion of files. As shown in Figure 6, the search delay of our HeX on SD is around 0.76ms for all keywords under the three circumstances. On the contrary, the time costs of Maiden and SGX-SE2 at 25% deletion are much higher where the search of the most frequent keyword takes 52ms and 209ms respectively in Figure 6a. When the deletion portion increases to 50% in Figure 6b, the search time of the most frequent keyword in Maiden and SGX-SE2 grows to 60ms and 342ms accordingly. The same costs become 69ms and 477ms when it comes to 75% deletion in Figure 6c. The higher the percentage of deleted files is, the more time it will take for the same keyword even though the number of matching files decreases. This is because Maiden and SGX-SE2 need to process the deletion information during data search instead of data update. In addition, in Maiden and SGX-SE2, it costs more search time as the keyword's frequency increases since it needs more time to retrieve all matching index entries. In contrast, the time in HeX remains almost the same for any keyword because the calculation involved is irrelevant to the number of matched files.

The search efficiency on the Enron dataset is shown in Figure 7. We can see all schemes achieve lower search delays than those on SD due to the smaller number of files. For the most frequent keyword at 25% deletion in Figure 7a, our HeX takes 0.34ms, while Maiden and SGX-SE2 need 13ms and 42ms respectively. When the deletion portion rises to 50% in Figure 7b and 75% in Figure 7c, Maiden costs 16ms and 19ms, and SGX-SE2 needs 73ms and 117ms respectively. However, our HeX always remains around 0.3ms no matter how the deletion percentage changes.

Memory Consumption. Figure 8 depicts the memory occupation of the server and the enclave in HeX over the two datasets. The increase of the memory size in both the server and the enclave is approximately linear to the percentage

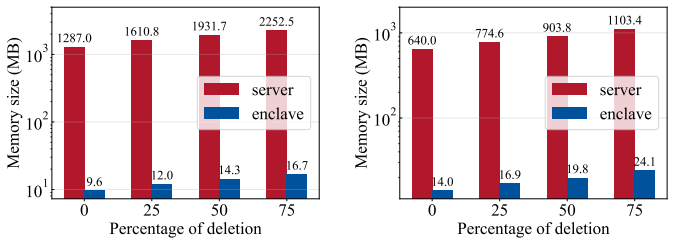


Fig. 8: Memory Consumption.

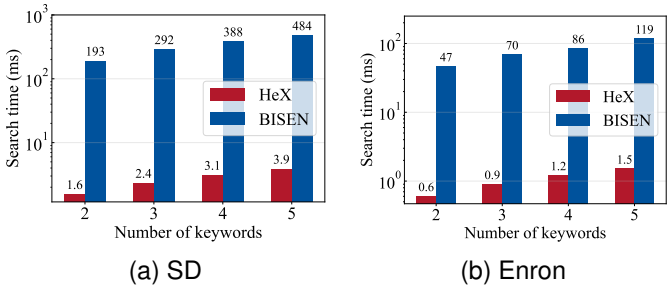


Fig. 9: Search Time of Boolean Queries.

of deleted files in both cases. This is because HeX processes a data deletion by faking as a new data insertion into the database instead of deleting any records. On the server side, one more encrypted index entry will be added when a data update comes regardless of the update type. On the other hand, the enclave will insert one more item into the map of keyword version number state. Moreover, the server memory cost of the Enron dataset in Figure 8b is about half of that of SD in Figure 8a, for example, 774.6MB on Enron versus 1610.8MB on SD when deleting 25%. This indicates that the server memory consumption is more relevant to the number of files and keyword-file pairs than keywords. It is because the number of files directly determines the length of the index cipher while the number of keyword-file pairs dominates the amount of index. Note that although we only build one index entry of each keyword, the number of keywords cannot decide the size of index as each keyword-file insertion actually brings one more entry. In contrast, the enclave memory used by Enron is more than SD, e.g., 24.1MB on Enron versus 16.7MB on SD when deleting 75%, because it is only affected by the number of keyword-file pairs.

7.2 Performance of Boolean Queries

To evaluate the performance of Boolean queries, we conduct a Boolean query containing several keywords for HeX and BISEN. The queried keywords are randomly sampled from the most 10 frequent keywords in the dataset. Figure 9 gives the average search time of the two schemes on both SD and Enron datasets. The number of keywords increases from 2 to 5. As shown in Figure 9a, both HeX and BISEN need the time that is proportional to the number of keywords. Under the same setting, the search of HeX is more than 120× faster than BISEN. This huge advantage is narrowed to around 70× when it comes to the Enron dataset, as depicted in Figure 9b. Although there are far more keywords in Enron than in the SD dataset, the search time in Enron is smaller than SD. This is because Enron has fewer files so that HeX constructs a shorter bitmap for each keyword and BISEN needs to retrieve fewer index entries.

7.3 Performance of Range Queries

Search Time. We present the average time cost of conducting a range query using HeX and HybrIDX in Figure 10a. Under all four bit settings, the search time of our HeX is much smaller than that HybrIDX, especially achieving up to

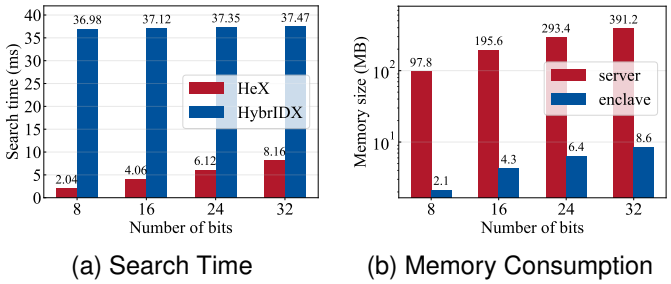


Fig. 10: Performance of Range Queries.

18× faster when the bit number is 8. This is because HeX only needs to retrieve 8 keyword records while HybrIDX has to read all qualified number records. It can also explain why the bit setting has a very minor impact on the search time for HybrIDX (around 37ms) since the average number of answers is dominated by the total amount of records. As the number of bits gets larger, the search time of HeX grows proportionately because the number of keywords transformed from the range query is equal to the bits.

Memory Consumption. Figure 10b shows the memory cost of the range query in HeX. We can see that the memory occupation of both the server and the enclave is almost linear to the bit number. The reason behind this is that the number of tuples generated by a b-bit value is exactly b, thus the number of keywords linearly increases as well. Compared with the memory statistics of a single keyword-file search on the Enron dataset in Figure 8b, where the amount of files is also 10,000, it costs much less memory in the range query than that in the keyword query. This is because the average number of keywords that a file contains in the Enron dataset is about 52, which is larger than the bit numbers in all settings.

8 Conclusion and Future Work

In this paper, we propose a new DSSE scheme called HeX to enable rich encrypted queries on untrusted servers with strong forward and backward privacy for the first time. We reduce the leakage by leveraging TEE and deferred obfuscating techniques during data updates and searches. By extending the basic scheme, we further support Boolean queries and range queries over multiple-attribute data. Our theoretical analysis validates that our HeX achieves high forward and backward security by obfuscating the access pattern and search pattern. Extensive evaluations show that

the search efficiency of HeX is much higher than state-of-the-art designs.

Our HeX achieves strong security and high search efficiency at the cost of insertion latency and memory occupation. Thus, an important direction of future work is to reduce these costs without compromising current advantages. Besides, it is also interesting to explore how to support batch operations, multiple clients and access control, capturing more application scenarios.

ACKNOWLEDGMENTS

This work was supported by Hong Kong RGC GRF Projects 12202922, 15238724, Shenzhen Science and Technology Program JCYJ20230807140412025, and National Natural Science Foundation of China (Grant No. 62472185).

REFERENCES

[1] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," *J. Comput. Secur.*, vol. 19, no. 5, pp. 895–934, 2011.

[2] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *NDSS*, The Internet Society, 2014.

[3] B. Minaud and M. Reichle, "Dynamic local searchable symmetric encryption," in *CRYPTO (4)*, vol. 13510 of *Lecture Notes in Computer Science*, pp. 91–120, Springer, 2022.

[4] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *CCS*, pp. 1465–1482, ACM, 2017.

[5] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *NDSS*, The Internet Society, 2014.

[6] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *USENIX Security Symposium*, pp. 707–720, USENIX Association, 2016.

[7] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *CCS*, pp. 1143–1154, ACM, 2016.

[8] J. Li, Y. Huang, Y. Wei, S. Lv, Z. Liu, C. Dong, and W. Lou, "Searchable symmetric encryption with forward search privacy," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 1, pp. 460–474, 2021.

[9] S. Sun, R. Steinfeld, S. Lai, X. Yuan, A. Sakzad, J. K. Liu, S. Nepal, and D. Gu, "Practical non-interactive searchable encryption with forward and backward privacy," in *NDSS*, The Internet Society, 2021.

[10] V. Vo, X. Yuan, S. Sun, J. K. Liu, S. Nepal, and C. Wang, "Shielddb: An encrypted document database with padding countermeasures," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 4, pp. 4236–4252, 2023.

[11] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption," in *CRYPTO (3)*, vol. 9816 of *Lecture Notes in Computer Science*, pp. 563–592, Springer, 2016.

[12] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *NSDI*, pp. 283–298, USENIX Association, 2017.

[13] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *IEEE Symposium on Security and Privacy*, pp. 279–296, IEEE Computer Society, 2018.

[14] Z. Wu and R. Li, "OBI: a multi-path oblivious RAM for forward-and-backward-secure searchable encryption," in *NDSS*, The Internet Society, 2023.

[15] G. Amjad, S. Kamara, and T. Moataz, "Forward and backward private searchable encryption with SGX," in *EuroSec@EuroSys*, pp. 4:1–4:6, ACM, 2019.

[16] V. Vo, S. Lai, X. Yuan, S. Sun, S. Nepal, and J. K. Liu, "Accelerating forward and backward private searchable encryption using trusted execution," in *ACNS (2)*, vol. 12147 of *Lecture Notes in Computer Science*, pp. 83–103, Springer, 2020.

[17] V. Vo, S. Lai, X. Yuan, S. Nepal, and J. K. Liu, "Towards efficient and strong backward private searchable encryption with secure enclaves," in *ACNS (1)*, vol. 12726 of *Lecture Notes in Computer Science*, pp. 50–75, Springer, 2021.

[18] H. Wang, C. Xu, C. Zhang, J. Xu, Z. Peng, and J. Pei, "vchain+: Optimizing verifiable blockchain boolean range queries," in *ICDE*, pp. 1927–1940, IEEE, 2022.

[19] M. Chen, R. Zhou, H. Chen, J. Xiao, H. Jin, and B. Li, "Horae: A graph stream summarization structure for efficient temporal range query," in *ICDE*, pp. 2792–2804, IEEE, 2022.

[20] J. Chang, B. Li, J. Xiao, L. Lin, and H. Jin, "Anole: A lightweight and verifiable learned-based index for time range query on blockchain systems," in *DASFAA (1)*, vol. 13943 of *Lecture Notes in Computer Science*, pp. 519–534, Springer, 2023.

[21] S. Patranabis and D. Mukhopadhyay, "Forward and backward private conjunctive searchable symmetric encryption," in *NDSS*, The Internet Society, 2021.

[22] X. Yuan, X. Yuan, Y. Zhang, B. Li, and C. Wang, "Enabling encrypted boolean queries in geographically distributed databases," *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 3, pp. 634–646, 2020.

[23] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. N. Garofalakis, "Practical private range search revisited," in *SIGMOD Conference*, pp. 185–198, ACM, 2016.

[24] Y. Guo, X. Yuan, X. Wang, C. Wang, B. Li, and X. Jia, "Enabling encrypted rich queries in distributed key-value stores," *IEEE Trans. Parallel Distributed Syst.*, vol. 30, no. 6, pp. 1283–1297, 2019.

[25] K. Ren, Y. Guo, J. Li, X. Jia, C. Wang, Y. Zhou, S. Wang, N. Cao, and F. Li, "HybrIDX: New hybrid index for volume-hiding range queries in data outsourcing services," in *ICDCS*, pp. 23–33, IEEE, 2020.

[26] X. Liu, G. Yang, Y. Mu, and R. H. Deng, "Multi-user verifiable searchable symmetric encryption for cloud storage," *IEEE Trans. Dependable Secur. Comput.*, vol. 17, no. 6, pp. 1322–1332, 2020.

[27] X. Ge, J. Yu, H. Zhang, C. Hu, Z. Li, Z. Qin, and R. Hao, "Towards achieving keyword search over dynamic encrypted cloud data with symmetric-key based verification," *IEEE Trans. Dependable Secur. Comput.*, vol. 18, no. 1, pp. 490–504, 2021.

[28] H. Li, H. Zhou, H. Huang, and X. Jia, "Verifiable encrypted search with forward secure updates for blockchain-based system," in *WASA (1)*, vol. 12384 of *Lecture Notes in Computer Science*, pp. 206–217, Springer, 2020.

[29] Y. Guo, C. Zhang, and X. Jia, "Verifiable and forward-secure encrypted search using blockchain techniques," in *ICC*, pp. 1–7, IEEE, 2020.

[30] Y. Guo, C. Zhang, C. Wang, and X. Jia, "Towards public verifiable and forward-privacy encrypted search by using blockchain," *IEEE Trans. Dependable Secur. Comput.*, vol. 20, no. 3, pp. 2111–2126, 2023.

[31] Z. Peng, H. Wu, B. Xiao, and S. Guo, "VQL: providing query efficiency and data authenticity in blockchain systems," in *ICDE Workshops*, pp. 1–6, IEEE, 2019.

[32] Z. Peng, J. Xu, H. Hu, and L. Chen, "Blockshare: A blockchain empowered system for privacy-preserving verifiable data sharing," *IEEE Data Eng. Bull.*, vol. 45, no. 2, pp. 14–24, 2022.

[33] H. Wu, Z. Peng, S. Guo, Y. Yang, and B. Xiao, "VQL: efficient and verifiable cloud query services for blockchain systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 6, pp. 1393–1406, 2022.

[34] P. Xu, W. Susilo, W. Wang, T. Chen, Q. Wu, K. Liang, and H. Jin, "ROSE: robust searchable encryption with forward and backward security," *IEEE Trans. Inf. Forensics Secur.*, vol. 17, pp. 1115–1130, 2022.

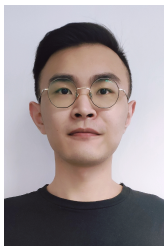
[35] Y. Zheng, R. Lu, J. Shao, F. Yin, and H. Zhu, "Achieving practical symmetric searchable encryption with search pattern privacy over cloud," *IEEE Trans. Serv. Comput.*, vol. 15, no. 3, pp. 1358–1370, 2022.

[36] B. Ferreira, B. Portela, T. Oliveira, G. Borges, H. J. L. Domingos, and J. Leitão, "Boolean searchable symmetric encryption with filters on trusted hardware," *IEEE Trans. Dependable Secur. Comput.*, vol. 19, no. 2, pp. 1307–1319, 2022.

[37] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *CCS*, pp. 1038–1055, ACM, 2018.

[38] T. Chen, P. Xu, S. Picek, B. Luo, W. Susilo, H. Jin, and K. Liang, "The power of bamboo: On the post-compromise security for searchable symmetric encryption," in *NDSS*, The Internet Society, 2023.

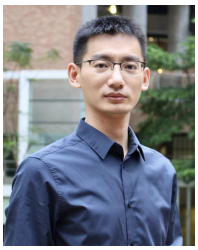
[39] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer, "SCONE: secure linux containers with intel SGX," in *OSDI*, pp. 689–703, USENIX Association, 2016.
[40] C. Zuo, S. Sun, J. K. Liu, J. Shao, J. Pieprzyk, and L. Xu, "Forward and backward private DSSE for range queries," *IEEE Trans. Dependable Secur. Comput.*, vol. 19, no. 1, pp. 328–338, 2022.
[41] H. Wu, R. Song, K. Lei, and B. Xiao, "Slicer: Verifiable, secure and fair search over encrypted numerical data using blockchain," in *ICDCS*, pp. 1201–1211, IEEE, 2022.
[42] Q. Song, Z. Liu, J. Cao, K. Sun, Q. Li, and C. Wang, "SAP-SSE: protecting search patterns and access patterns in searchable symmetric encryption," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 1795–1809, 2021.
[43] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *CHES*, vol. 10529 of *Lecture Notes in Computer Science*, pp. 69–90, Springer, 2017.
[44] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *USENIX Security Symposium*, pp. 557–574, USENIX Association, 2017.
[45] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace : Oblivious memory primitives from intel SGX," in *NDSS*, The Internet Society, 2018.
[46] S. Sasy, A. Johnson, and I. Goldberg, "Fast fully oblivious compaction and shuffling," in *CCS*, pp. 2565–2579, ACM, 2022.
[47] Z. Gui, K. G. Paterson, and S. Patranabis, "Rethinking searchable symmetric encryption," in *SP*, pp. 1401–1418, IEEE, 2023.
[48] L. Xu, L. Zheng, C. Xu, X. Yuan, and C. Wang, "Leakage-abuse attacks against forward and backward private searchable symmetric encryption," in *CCS*, pp. 3003–3017, ACM, 2023.
[49] A. Hoover, R. Ng, D. Khu, Y. Li, J. Lim, D. Ng, J. Lim, and Y. Song, "Leakage-abuse attacks against structured encryption for SQL," *IACR Cryptol. ePrint Arch.*, p. 554, 2024.



Haotian Wu is a postdoctoral fellow in the Department of Industrial and Systems Engineering, The Hong Kong Polytechnic University. He received his B.Sc. and M.Sc. degrees from Southeast University, and his Ph.D. degree from The Hong Kong Polytechnic University. His research interests include data security, applied cryptography and blockchain systems.

Zhe Peng is currently a research assistant professor in the Department of Industrial and Systems Engineering, The Hong Kong Polytechnic University. He received the B.S. degree from Northwestern Polytechnical University, the M.S. degree from University of Science and Technology of China, and the Ph.D. degree from The Hong Kong Polytechnic University. He was a visiting scholar in the Department of Electrical and Computer Engineering, Stony Brook University. His research interests include blockchain, web3, artificial intelligence of things, data security and privacy.

Jiang Xiao (Member, IEEE) is currently a Professor in School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan, China. Jiang received the B.Sc. degree from HUST in 2009 and the Ph.D. degree from Hong Kong University of Science and Technology (HKUST) in 2014. Her research interests include blockchain, and distributed computing. Her awards include CCF-Intel Young Faculty Research Program 2017, Hubei Downlight Program 2018, ACM Wuhan Rising Star Award 2019, Knowledge Innovation Program of Wuhan-Shuguang 2022, and Best Paper Awards from IEEE ICPADS/GLOBECOM/GPC/BLOCKCHAIN.



Lei Xue is an associate professor in School of Cyber Science and Technology at Sun Yat-Sen University. He received the Ph.D. degree in Computer Science from The Hong Kong Polytechnic University (PolyU). He is widely interested in designing and implementing efficient and practical security systems, with a particular focus on applying hybrid program/application analysis techniques, network traffic analysis methodology, and machine learning based algorithms to addressing challenging security and privacy issues in mobile, automotive, and network systems. Currently, his research topics mainly focus on mobile and IoT system security, program analysis, and automotive security.

Chenhao Lin received the B.E. degree in automation from Xi'an Jiaotong University in 2011, the M.Sc. degree in electrical engineering from Columbia University, in 2013 and the Ph.D. degree from The Hong Kong Polytechnic University, in 2018. He is currently a Research Fellow at the Xi'an Jiaotong University of China. His research interests are in artificial intelligence security, adversarial attack and robustness, identity authentication, and pattern recognition.

Sai-Ho Chung (Nick) is currently an associate professor and the associate head in the Department of Industrial and Systems Engineering, The Hong Kong Polytechnic University. He received his B.Eng. (Hon), M.Phil. and Ph.D. degrees in Industrial, Manufacturing and Systems Engineering from The University of Hong Kong in 2001, 2004, and 2007, respectively. His research interests include logistics and supply chain management, supply chain collaboration, supply chain finance, production scheduling, distribution network, vehicle routing, container terminal operations, airline crew scheduling, aircraft maintenance routing, flight fuel consumption estimation, etc.