Result Pattern Hiding Boolean Searchable Encryption: Achieving Negligible False Positive Rates in Low Storage Overhead

Dandan Yuan[1*], Shujie Cui[2], and Giovanni Russello[3]

[1] Centrum Wiskunde & Informatica
dandan.yuan@cwi.nl
[2] Monash University
Shujie.Cui@monash.edu
[3] University of Auckland
g.russello@auckland.ac.nz

Abstract. Boolean Searchable Symmetric Encryption (SSE) enables secure outsourcing of databases to an untrusted server in encrypted form and allows the client to execute secure Boolean queries involving multiple keywords. The leakage of keyword pair result pattern (KPRP) in a Boolean search poses a significant threat, which reveals the intersection of documents containing any two keywords involved in a search and can be exploited by attackers to recover plaintext information about searched keywords (USENIX Security'16). However, existing KPRP-hiding schemes either rely on Bloom filters (S&P'14, CCS'18), leading to high false positive search results (where non-matching documents could be erroneously identified as matches) that hinder the extension to multi-client settings (CCS'13), or require excessive server storage (PETS'23), making them impractical for large-scale sparse databases.
In this paper, we introduce Hidden Boolean Search (HBS), the first KPRP-hiding Boolean SSE scheme with both negligible false positives (essential for satisfying the standard correctness definition of SSE) and low server storage requirements. HBS leverages a novel cryptographic tool called Result-hiding Filter (RH-filter). It distinguishes itself as the first tool that supports computationally correct membership queries with hiding results at nearly constant overhead. With the help of RH-filter, compared to the most efficient KPRP-hiding scheme (CCS'18) in terms of overall storage and search efficiency, HBS surpasses it across all performance metrics, mitigates false positives, and achieves significantly stronger query expressiveness. We further extend HBS to the dynamic setting, resulting in a scheme named DHBS, which maintains KPRP-hiding while ensuring forward and backward privacy—two critical security guarantees in the dynamic setting.

Keywords: Boolean Searchable Symmetric Encryption, Keyword Pair Result Pattern, False Positive Rate, Sparse Database, Low Server Storage

1 Introduction

Searchable symmetric encryption (SSE) enables the client to outsource a database to an untrusted server and search the database securely and efficiently. The most common type of search is to look for documents that contain a keyword, referred to as a single-keyword search, and has been extensively studied in the existing literature [SWP, CGKO, KPR, CJJ$^+$a, SPS, Bos, BMO, SDY$^+$20, SYL$^+$, CPPJ, ZSL$^+$b, SSL$^+$, YCR]. However, to meet the demands of a broader range of application scenarios, more complex queries are required to be supported. These include but are not limited to Boolean search [CJJ$^+$b, LPS$^+$, KMa, PM, PPSY, YZCR], range search [ZSL$^+$a, WCb], and ranked search [MZK, KLSN]. In particular, a Boolean search allows the client to build a Boolean formula by combining multiple keywords with the Boolean operators AND($\wedge$), OR($\vee$), and NOT($\neg$) and search for the documents satisfying the Boolean formula. Hereafter, we use $\psi(w_1, \cdots, w_n)$ to denote an arbitrary Boolean formula involving n keywords $w_1, \cdots, w_n$.

KPRP Leakage. Implementing secure and efficient Boolean SSE solutions is non-trivial since there is a trade-off between security, performance, and query expressiveness. To ensure practicality, the

[*] Part of the work was done while the author was at University of Auckland.

existing solutions usually choose to leak a small amount of information (referred to as *leakage*) to the server. For instance, most of the Boolean SSE solutions in the literature reveal the *keyword pair result pattern* (KPRP) within a search. KPRP refers to the intersection of the documents matched by any two keywords involved in a search. Let $\mathrm{DB}(w)$ represent the document identifiers matched by a keyword w. KPRP related to a Boolean search $\psi(w_1, \cdots, w_n)$ could be denoted as $\mathrm{DB}(w_i) \cap \mathrm{DB}(w_j)$ for any $1 \le i < j \le n$. However, leakage-abuse attacks [CGPR, BKMb, PW] show that some leakage profiles are unacceptable as they can be utilized to recover the database and queries. In particular, Zhang *et al.*' [ZKP] file-injection attack can exploit KPRP to recover plaintext information about w_i for $1 \le i \le n$. Achieving *KPRP-hiding* is essential to mitigate this attack. According to [LPS+,YZCR], KPRP-hiding guarantees that a Boolean search does not disclose which document belongs to $\mathrm{DB}(w_i) \cap \mathrm{DB}(w_j)$ for any $1 \le i < j \le n$, except for the information that can be inferred from the final search result, namely the document identifiers matched by the search.

Naive Solution. Boolean SSE can be naively achieved through any single-keyword SSE scheme. Specifically, within a Boolean search $\psi(w_1, \cdots, w_n)$, for $1 \le i \le n$, the client can issue a single-keyword search on w_i and receive $\mathrm{DB}(w_i)$. Then, it computes the final search result $\mathrm{DB}(\psi(w_1, \cdots, w_n))$ by combining the received single-keyword results. If the single-keyword search is response-hiding, which means that it does not reveal the search result to the server, the whole search process will not reveal KPRP. However, the optimal computational and communication overhead achievable is $O(\sum_{i=1}^{n} |\mathrm{DB}(w_i)|)$, which makes the solution highly inefficient when one or more involved keywords match a large number of documents. Furthermore, the naive solution may lead to undesirable leakage. For example, if the utilized single-keyword SSE reveals the response length (*i.e.,* $|\mathrm{DB}(w_i)|$), which cannot currently be hidden without incurring significant overhead (scaling at least linearly with the maximum possible response length for a search) [KMb,PPYY,WCa], it exposes the instantiated naive solution to a wider range of leakage-abuse attacks [CGPR,BKMb,OKa]. Hence, there is a crucial need to devise KPRP-hiding schemes that not only enhance search efficiency but also mitigate unintended leakage.

Limitations of Existing KPRP-hiding Solutions. In the literature, four Boolean SSE schemes have been proposed to achieve KPRP-hiding with sub-linear search efficiency: Blind Seer [PKV+], HXT [LPS+], Rphx [JcCQ+22], and HDXT [YZCR]. We notice that all of these schemes employ search methodologies that involve querying the membership of specific keyword-document pairs within the database (a keyword-document pair is said to exist in the database if the document contains the keyword). Throughout this process, the membership query result for any given keyword-document pair must remain concealed from the server. Failure to do so may leak KPRP. For example, in a Boolean search $\psi(w_1, \cdots, w_n)$, if the server learns that two keyword-document pairs, (w_1, id) and (w_2, id), exist in the database, it can deduce that id belongs to $\mathrm{DB}(w_1) \cap \mathrm{DB}(w_2)$, thereby compromising KPRP-hiding.

Three out of these four schemes, namely Blind Seer [PKV+], Rphx [JcCQ+22], and HXT [LPS+], employ the Bloom filter [Blo70] to implement membership queries on keyword-document pairs. A Bloom filter is a vector that represents a set and facilitates rapid membership queries. These three schemes leverage a specific property of the Bloom filter to hide membership query results effectively. This property allows a Bloom filter to be trivially encrypted so that the access pattern (*i.e.,* the sequence of accessed filter entries) resulting from a membership query does not reveal the query's outcome, while maintaining high query efficiency. However, a Bloom filter may erroneously indicate that a keyword-document pair belongs to the database when it does not, resulting in what is known as a false positive. False positives could lead to search results containing documents that do not meet the corresponding search criteria. For example, in HXT, the false positive rate for a search can be as high as $\frac{|\mathrm{DB}(w_1)|}{10^6}$. This implies that HXT is almost certain to return unmatched documents for some searches, especially for large-scale databases. In this paper, we use the term *noticeable false positive rate* to refer to a false positive rate that fails to reach a negligible level. Achieving a negligible false positive rate is crucial for meeting the standard correctness definition of SSE (provided in Section 2.4). More importantly, as highlighted by Jarecki *et al.* [JJK+], noticeable false positives hinder the extension of SSE schemes to multi-client settings, as clients may access documents that they are not authorized to view.

HDXT [YZCR] is the only one that does not use the Bloom filter. Instead, it employs a mapping table that associates each possible keyword-document pair with a bit, indicating whether the pair exists in the database. While this encrypted mapping table can help achieve negligible false positive rates, it does require substantial server-side storage linear with $|\mathrm{W}| \cdot |\mathrm{D}|$, where $|\mathrm{W}|$ and $|\mathrm{D}|$ denote the total number of keywords and documents, respectively. In the case of sparse databases, the server must allocate excessive space to store nonexistent data, resulting in poor practicality and scalability.

Given the current state of research, a question naturally emerges:

Is it inevitable that achieving KPRP-hiding would entail noticeable false positive rates or excessive server storage?

Our Contributions and Techniques. In this paper, we introduce the first KPRP-hiding Boolean SSE scheme with both negligible false positive rates and low server storage requirements (lower than all the above KPRP-hiding solutions). In alignment with most KPRP-hiding solutions [PKV$^+$,LPS$^+$, JcCQ$^+$22], we primarily focus on the static setting and name our static scheme: Hidden Boolean Search (HBS). We also consider the extension to the dynamic setting and name the dynamic solution Dynamic Hidden Boolean Search (DHBS).

HBS adopts the OXT-based search framework, which was developed by Cash *et al.* [CJJ$^+$b] in their Boolean SSE scheme OXT and is still being followed by state-of-art works [LPS$^+$, PM, YZCR]. Taking a conjunctive search $w_1 \wedge \cdots \wedge w_n$ as an example, the OXT-based framework enables a search first to identify the documents matching the least frequently occurring keyword (selected as w_1, called *s-term*) and subsequently obtain the search result by checking if each identified document contains the other $(n-1)$ keywords (each is called *x-term*). OXT fails to achieve KPRP-hiding. The cornerstone of implementing KPRP-hiding within the OXT-based search framework is to conceal whether a document matched by the *s-term* contains a *x-term*, except that it is revealed by the final search result. To achieve this security property efficiently without introducing drawbacks in the other two KPRP-hiding OXT-based schemes, namely noticeable false positive rates in HXT [LPS$^+$] and extensive server storage in HDXT [YZCR], we introduce a novel cryptographic tool named Result-hiding Filter (RH-filter). As far as we know, RH-filter is the first tool supporting encrypted membership queries that achieve both result concealment and computational correctness without relying on intricate cryptographic schemes such as ORAM [SvDS$^+$,WNL$^+$]. Surprisingly, with the assistance of RH-Filter, HBS achieves desirable correctness and low server storage without sacrificing search efficiency and shows impressive query expressiveness. In Table 1, we present a comparison of HBS and DHBS with existing KPRP-hiding schemes. The main contributions can be summarized as below:

1. **Introduce a Novel Cryptographic Tool RH-filter.** As far as we know, ORAM-based Oblivious Set [WNL$^+$] is the only existing cryptographic tool that supports *result-hiding* membership queries with negligible false positive rates, where result-hiding refers to the concealment of membership query results. In this paper, we introduce a cryptographic tool RH-filter, which achieves significant performance improvements, *e.g.,* constant-level computational overhead and sub-logarithmic communication overhead, by sacrificing a slight degree of security, *i.e.,* revealing which queries are for the same element. For clarity and broader applicability, we formally define this tool and provide rigorous proofs of its correctness and security.
2. **Propose a KPRP-Hiding Scheme HBS that Exhibits Notable Advantages in Multiple Aspects.**
   – **Correctness.** HBS is the first KPRP-hiding solution to achieve negligible false positive rates with low storage overhead, which depends solely on the number of items in the database. Also, we extend HBS to multi-client settings, as detailed in Section 6, to emphasize the critical importance of achieving negligible false positive rates.
   – **Query Expressiveness.** HBS is the first constant-round solution to achieve KPRP-hiding for all Boolean queries, notably demonstrating sub-linear efficiency for Boolean queries in the search normal form (SNF) [CJJ$^+$b]. SNF refers to search queries that can be expressed as $w_1 \wedge \psi(w_2, \cdots, w_n)$ where w_i indicates a keyword and ψ signifies any arbitrary Boolean formula. Only Blind Seer [PKV$^+$] in the literature can achieve KPRP-hiding for all Boolean queries without relying on trusted hardware. However, this comes at the cost of requiring logarithmic round complexity. In Appendix A, we explain the limitations of HXT and HDXT regarding query expressiveness.
   – **Performance.** Compared to HXT, the current leading KPRP-hiding solution in terms of overall storage and search efficiency [1], HBS surpasses it across all performance metrics.

[1] While HDXT demonstrates superior search efficiency compared to HXT, this enhancement is offset by a substantial increase in storage requirements, tens to hundreds of times greater.

3. **Extension to the Dynamic Setting.** SSE that supports secure updates is referred to as dynamic SSE (DSSE). Our DSSE scheme DHBS achieves the correctness and query expressiveness of HBS while demonstrating desirable security and performance.
   - **Security.** DHBS attains the same level of security as the sole existing KPRP-hiding DSSE scheme, HDXT, by providing KPRP-hiding along with forward and backward privacy. Note that forward and backward privacy are two crucial security properties in the dynamic setting and have been extensively discussed in previous works [BMO,CPPJ,ZSL+b,SSL+,PM,YZCR].
   - **Performance.** Compared to HDXT, DHBS offers a significant advantage in curtailing server storage overhead by tens to hundreds of times. While its search complexity depends on the total number of updates (HDXT does not), our experimental results showcase that DHBS sustains high efficiency even after substantial updates. Consequently, we contend that DHBS is better suited than HDXT for specific datasets, such as sparse databases and those with infrequent updates.

Table 1. Comparison with Existing KPRP-hiding Boolean SSE Schemes

Schemes	Search				Server Storage	SNF	Negligible False Positives
	Computation	Communication	Rounds				
			Id	Doc			
Static Setting							
Blind Seer [PKV+]	$O(\gamma nm \log \lvert \mathrm{D} \rvert)$	$O((\gamma nm \log \lvert \mathrm{D} \rvert)$	$O(\log \lvert \mathrm{D} \rvert)$	$O(\log \lvert \mathrm{D} \rvert)$	$O(\lvert \mathrm{W} \rvert \lvert \mathrm{D} \rvert)$	✓	✗
HXT [LPS+]	$O(\gamma n m_1)$	$O(\gamma n m_1)$	2	3	$O(\xi N)$	✗	✗
HBS	$\boldsymbol{O(nm_1)}$	$\boldsymbol{O(\pi n m_1)}$	**2**	**2**	$\boldsymbol{O(\varphi \pi N)}$	✓	✓
Dynamic Setting							
HDXT [YZCR]	$O(u_1 + nm_1)$	$O(u_1 + nm_1)$	2	3	$O(\lvert \mathrm{W} \rvert \lvert \mathrm{D} \rvert)$	✗	✓
DHBS	$\boldsymbol{O(u_1 + \tau n m_1)}$	$\boldsymbol{O(u_1 + \pi \tau n m_1)}$	**2**	**3**	$\boldsymbol{O(\varphi \pi N^+)}$	✓	✓

The table presents KPRP-hiding solutions that achieve sub-linear search efficiency without relying on trusted hardware. The columns labeled 'Id' and 'Doc' represent the number of interaction rounds required to search for the intended document identifiers and document contents, respectively. The column 'SNF' indicates whether the corresponding scheme achieves KPRP-hiding for all Boolean queries in SNF. $|\mathrm{W}|$, $|\mathrm{D}|$, and N denote the number of keywords, documents, and keyword-document pairs that exist in the database, respectively. n represents the number of keywords involved in the search. m_1 and m denote the number of documents matched by w_1 and by the query, respectively. The parameters γ and ξ for the Bloom filter are set to 20 and 29, respectively, in [PKV+, LPS+]. The parameters φ and π for the RH-filter are set to 2 and 8 in the paper. u_1 represents the number of documents matched by w_1 in the initial database, plus the number of updates related to w_1 after the setup. τ is the total number of updates divided by a large constant. N^+ is the number of the keyword-document pairs exiting in the initial database, plus the number of updates after the setup.

2 Preliminaries

In this section, we begin by introducing the notations employed throughout the subsequent sections. Following this, we proceed to present formal definitions for three cryptographic primitives: T-set, RH-filter, and Boolean SSE.

2.1 Notations

We use $\{0,1\}^l$ to denote the set of all binary strings of length l, $\{0,1\}^*$ to stand for the set formed of arbitrary-length strings, and $[1,\zeta]$ to denote the set of elements from 1 to an integer ζ. With $a_1 \leftarrow a_2$ we denote the value of a_2 is assigned to a_1, and with $a_1 \overset{\$}{\leftarrow} \mathrm{S}$ we denote that a_1 is sampled uniformly at random from the set S. We use $|\mathrm{X}|$ to represent the cardinality of a container X, *e.g.,* a set/list/map.

2.2 T-set

T-set [CJJ+b] enables a data owner that possesses a number of documents, each containing specific keywords, to associate a list of fixed-sized tuples with each keyword. Later, the data owner can produce tokens related to keywords to retrieve the associated lists. The set of all keywords present in the database is denoted as W. T-set comprises the following three algorithms:

- **TSetSetup**$(1^\lambda, \mathrm{T}) \rightarrow (\mathrm{TSet}, K_T)$: Taking the security parameter λ and a table T that associates each keyword in W to a list of tuples as inputs, it produces two outputs: TSet and K_T.

- **TSetGetTag**$(K_T, w) \rightarrow stag$: On input the key K_T and a keyword $w \in \mathrm{W}$, it outputs the corresponding query token $stag$.
- **TSetRetrieve**$(stag, \mathrm{TSet}) \rightarrow \mathrm{T}[w]$: Taking as input a query token $stag$ related to a keyword w and TSet, it outputs $\mathrm{T}[w]$.

Intuitively, an implementation of T-set is considered correct if, for any given security parameter λ, set of keywords W, and table T, and for each keyword $w \in \mathrm{W}$, **TSetRetrieve**$(stag, \mathrm{TSet})$ always outputs the list $\mathrm{T}[w]$ when $(\mathrm{TSet}, K_T) \leftarrow$ **TSetSetup**$(1^\lambda, \mathrm{T})$ and $stag \leftarrow$ **TSetGetTag**(K_T, w). In terms of security, the main objective of T-set is to protect the confidentiality of information about the tuples in T and the keywords they are associated with, except for the retrieval results produced by TSetRetrieve. As in [CJJ+b], we allow T-set to reveal a small amount of information, which we refer to as the leakage $\mathcal{L}_T = (\mathcal{L}_T^{tup}(\mathrm{T}), \mathcal{L}_T^{tag}(\mathrm{T}, w))$. The formal definitions for correctness and security are provided in Appendix B.

2.3 Result-hiding Filter

This section introduces a cryptographic primitive named as the Result-hiding Filter, abbreviated as RH-filter. A RH-filter serves as an encrypted representation of a set and facilitates membership queries. The security requirement for an RH-filter encompasses not only safeguarding plaintext information related to the contents of the set and the queries, but also the crucial task of concealing membership query results. The latter security property is referred to as result-hiding. RH-filter will play a central role in our SSE protocols. Presenting its abstract definition not only aids in comprehending our protocol but also opens the possibility of its broader application in other suitable scenarios. Formally, RH-filter consists of the following five algorithms:

- **RHFSetup**$(1^\lambda) \rightarrow K_F$: Taking as input the security parameter λ, the algorithm outputs a secret key K_F.
- **RHFEncrypt**$(K_F, \Delta) \rightarrow \mathrm{ES}$: Taking as input the secret key K_F and a set Δ, the algorithm outputs the encrypted structure ES.
- **RHFGetTok**$(K_F, \delta) \rightarrow etok$: Taking as input the secret key K_F and an element δ, the algorithm outputs the encrypted token $etok$.
- **RHFRespond**$(etok, \mathrm{ES}) \rightarrow res$: Taking as input the token $etok$ and the encrypted structure ES, the algorithm outputs the response res.
- **RHFTest**$(K_F, \delta, res) \rightarrow b \in \{0,1\}$: On input the secret key K_F, the element δ, and the response res, the algorithm outputs a bit b.

In Definition 1, we define the correctness of the RH-filter within the context of computational correctness. Here, we introduce the function $In(\delta, \Delta)$, which yields 1 when $\delta \in \Delta$, and outputs 0 if $\delta \notin \Delta$.

1) **RHFSetup**(1^λ) is run to generate K_F.
2) $\mathcal{A}$ chooses a set Δ and is provided the encrypted structure $\mathrm{ES} \leftarrow$ **RHFEncrypt**(K_F, Δ).
3) $\mathcal{A}$ adaptively selects elements and performs queries. For each selected element δ, $\mathcal{A}$ makes a query on δ as follows: ① $\mathcal{A}$ is given the token $etok \leftarrow$ **RHFGetTok**(K_F, δ); ② $\mathcal{A}$ executes **RHFRespond**$(etok, \mathrm{ES})$ to obtain the response res; ③ $\mathcal{A}$ receives the query result $b^\delta \leftarrow$ **RHFTest**(K_F, δ, res).
4) The game outputs 1 if, in phase 3), $\mathcal{A}$ chooses an element δ on which the query result b^δ is not the output of $In(\delta, \Delta)$.

Fig. 1. $\mathrm{RHFCorr}_{\mathcal{A}}^{\Pi}(\lambda)$

Definition 1 (Correctness of RH-filter). *Let Π be a RH-filter implementation. We say Π satisfies correctness if for any security parameter λ and any PPT adversary $\mathcal{A}$, there exists a negligible function negl such that:*

$$\Pr[\mathrm{RHFCorr}^{\Pi}_{\mathcal{A}}(\lambda) = 1] \leqslant negl(\lambda)$$

where the game $\mathrm{RHFCorr}^{\Pi}_{\mathcal{A}}(\lambda)$ *is defined in Figure 1.*

In regard to security, we stipulate that the encrypted structure ES only reveals $|\Delta|$, and the encrypted token *etok* only leaks the repetition of the queried elements. We use a function *tp* to capture the repetition of queries. Formally, let $\mathcal{M}$ be the sequence of the queried elements, where $\mathcal{M}[i]$ represents the elements involved in the i-th query. For an element δ, $tp(\delta) = \{i|\mathcal{M}[i] = \delta\}$. In Definition 2, we give the simulation-based security definition for RH-filter.

$\mathrm{RHFReal}^{\Pi}_{\mathcal{A}}(\lambda)$:

1) **RHFSetup**(1^{λ}) is run to generate K_F.
2) $\mathcal{A}$ adaptively selects elements. For each element δ, $\mathcal{A}$ is given the token $etok \leftarrow$ **RHFGetTok**(K_F, δ).
3) $\mathcal{A}$ chooses a set Δ and is given the encrypted structure ES $\leftarrow$ **RHFEncrypt**(K_F, Δ).
4) $\mathcal{A}$ repeats step 2).
5) $\mathcal{A}$ outputs a bit b.

$\mathrm{RHFIdeal}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda)$:

1) $\mathcal{S}(\perp)$ outputs nothing.
2) $\mathcal{A}$ chooses elements in an adaptive way. For an element δ, $\mathcal{A}$ is given the token $etok \leftarrow \mathcal{S}(tp(\delta))$.
3) $\mathcal{A}$ selects a set Δ and receives ES $\leftarrow \mathcal{S}(|\Delta|)$.
4) $\mathcal{A}$ repeats step 2).
5) $\mathcal{A}$ outputs a bit b.

Fig. 2. Real and Ideal Games for RH-filter

Definition 2 (Security of RH-filter). *We say a RH-filter implementation Π is semantically secure if for any security parameter λ and any PPT adversary $\mathcal{A}$, there exist a a simulator $\mathcal{S}$ and a negligible function negl such that:*

$$|\Pr[\mathrm{RHFReal}^{\Pi}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathrm{RHFIdeal}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leqslant negl(\lambda)$$

where the games $\mathrm{RHFReal}^{\Pi}_{\mathcal{A}}(\lambda)$ *and* $\mathrm{RHFIdeal}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda)$ *are defined in Figure 2.*

2.4 Boolean Searchable Symmetric Encryption

The database DB is represented as $\{(id_i, \mathrm{W}_i)\}_{i=1}^{|\mathrm{D}|}$, where a document with the identifier $id_i \in \{0,1\}^{\lambda}$ contains a set of keywords $\mathrm{W}_i \subseteq \{0,1\}^*$. $\mathrm{D} = \{id_i\}_{i=1}^{|\mathrm{D}|}$ denotes the set of all the document identifiers. $\mathrm{W} = \cup_{i=1}^{|\mathrm{D}|}\mathrm{W}_i$ is the set of all the keywords. Let $\psi(\overline{\mathbf{w}})$ be a Boolean formula over a collection of keywords $\overline{\mathbf{w}} \subseteq \mathrm{W}$ with Boolean operators $\wedge$, $\vee$, and $\neg$. $\mathrm{DB}(\psi(\overline{\mathbf{w}}))$ represents the identifiers of the documents that satisfy $\psi(\overline{\mathbf{w}})$. An identifier id_i is said to satisfy $\psi(\overline{\mathbf{w}})$ iff $\psi(\overline{\mathbf{w}})$ is evaluated to be true after replacing every keyword in $\psi(\overline{\mathbf{w}})$ with true or false according to whether the keyword belongs to W_i or not. Boolean DSSE consists of the following three protocols (for static SSE, the state s and the update protocol are excluded from the subsequent protocols and definitions).

- **Setup**$(\lambda, \mathrm{DB}; \perp) \rightarrow (K, s; \mathrm{EDB})$: The client takes as input a security parameter λ and a database DB, and outputs a secret key K and an initial state s. The server outputs the encrypted database EDB.
- **Search**$(K, s, \psi(\overline{\mathbf{w}}); \mathrm{EDB}) \rightarrow (s', \mathrm{DB}(\ \psi(\overline{\mathbf{w}})); \mathrm{EDB}')$: On input the secret key K, the current state s, and a Boolean formula $\psi(\overline{\mathbf{w}})$, the client outputs a possibly updated state s' and the search result $\mathrm{DB}(\ \psi(\overline{\mathbf{w}}))$. The server inputs the encrypted database EDB and outputs a possibly updated encrypted database EDB$'$.
- **Update**$(\mathrm{K}, s, op, w, id; \mathrm{EDB}) \rightarrow (s'; \mathrm{EDB}')$: On input the secret key K, the latest state s, an update operator $op \in \{add, del\}$, a keyword w, and a document identifier id, the client outputs an updated state s'. With the encrypted database EDB as the input, the server outputs the possibly updated encrypted database EDB$'$.

The correctness requirement concerning SSE is that for any database DB, any encrypted database EDB generated from SSE.Setup or SSE.Update, and any Boolean formula $\psi(\overline{\mathbf{w}})$, the search query on $\psi(\overline{\mathbf{w}})$ returns $\mathrm{DB}(\psi(\overline{\mathbf{w}}))$ to the client except for negligible probability. Following [CGKO, KPR, CJJ+a], we have Definition 3.

Definition 3 (Correctness of SSE). *Let Π = {**Setup**, **Search**, **Update**} denote a DSSE scheme. We say Π is computationally correct if for any security parameter λ and any PPT adversary $\mathcal{A}$, there exist a negligible function negl such that:*

$$\Pr(\mathrm{SSE}\textsc{Correct}^{\Pi}_{\mathcal{A}}(\lambda) = 1) \leqslant negl(\lambda)$$

where $\mathrm{SSECorrect}^{\Pi}_{\mathcal{A}}(\lambda)$ *is defined as:*
$\mathrm{SSE}\textsc{Correct}^{\Pi}_{\mathcal{A}}(\lambda)$*: $\mathcal{A}$ chooses a database* DB*, and obtains* EDB *by calling Setup(λ,* DB*). Then it performs search queries Search($\psi(\overline{\mathbf{w}})$) and update queries Update(op, w, id) in an adaptive way. The game outputs* 1 *if the result of a search query on a Boolean formula $\psi(\overline{\mathbf{w}})$ is not the set* $\mathrm{DB}(\psi(\overline{\mathbf{w}}))$*.*

We use the function $\mathcal{L} = (\mathcal{L}^{Stp}(\mathrm{DB}), \mathcal{L}^{Srch}(\mathrm{DB}, \psi(\overline{\mathbf{w}}), \mathcal{L}^{Updt}(\mathrm{DB}, op, w, id))$ to denote the leakage profile for the setup, search, and update protocols. Definition 4 presents the security definition for SSE, which is borrowed from [KPR, CJJ+a].

Definition 4 (Adaptive Security of SSE). *Let Π = {**Setup**, **Search**, **Update**} denote a SSE scheme. We say Π 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:*

$$|\Pr[\mathrm{SSE}\textsc{Real}^{\Pi}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathrm{SSE}\textsc{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda) = 1]| \leqslant negl(\lambda)$$

where $\mathrm{SSE}\textsc{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ *and* $\mathrm{SSE}\textsc{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$ *are defined as:*

– $\mathrm{SSE}\textsc{Real}^{\Pi}_{\mathcal{A}}(\lambda)$*: At first, $\mathcal{A}$ chooses a database* DB*, and obtains* EDB *by invoking the protocol Setup(λ,* DB*). Then it repeatedly performs search queries Search($\psi(\overline{\mathbf{w}})$) and update queries Update(op, w, id) in an adaptive way. $\mathcal{A}$ outputs a bit b.*
– $\mathrm{SSE}\textsc{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$*: $\mathcal{A}$ chooses a database* DB*, and calls $\mathcal{S}(\mathcal{L}^{Stp}(\mathrm{DB}))$ to get the encrypted database* EDB*. After that, it adaptively performs search and update queries by calling $\mathcal{S}(\mathcal{L}^{Srch}(\mathrm{DB}, \psi(\overline{\mathbf{w}})))$ and $\mathcal{S}(\mathcal{L}^{Updt}(\mathrm{DB}, op, w, id))$, respectively. $\mathcal{A}$ outputs a bit b.*

The concept of keyword pair result pattern (KPRP) was first introduced by Zhang *et al.* [ZKP] to describe the intersection of documents that match any two keywords involved in a Boolean search query. Hiding KPRP from the server is crucial for mitigating the injection attack proposed in [ZKP]. We borrow the definition for KPRP-hiding from [YZCR], as presented in Definition 5.

Definition 5 (KPRP-hiding). *We say a Boolean SSE scheme satisfies KPRP-hiding if $\mathcal{L}^{Search}(\mathrm{DB}, \psi(\overline{\mathbf{w}}))$ does not reveal which identifiers belong to $\mathrm{DB}(w_i) \cap \mathrm{DB}(w_j)$ except for the information revealed by $\mathrm{DB}(\psi(\overline{\mathbf{w}}))$, for arbitrary two keywords w_i and w_j that belong to $\overline{\mathbf{w}}$.*

3 A RH-filter Construction

In this section, we present a construction for RH-filter. Before delving into the core idea and construction details, we begin by illustrating the challenges associated with achieving the desired security guarantee while maintaining high efficiency.
Design Challenges. The primary challenge in implementing a RH-filter lies in efficiently achieving result-hiding. This challenge arises because adversaries may attempt to compromise result-hiding by analyzing variations in the access patterns resulting from queries. For instance, an adversary could potentially deduce the presence of a queried element by scrutinizing the number of accesses made to ES by each query. To address this issue, a strategy involves the utilization of Oblivious RAM (ORAM) techniques [Gol, SvDS+], as done in the Oblivious Set proposed by Wang *et al.* [WNL+], to render access patterns resulting from queries indistinguishable. This effectively ensures that a query no longer leak any information. However, the adoption of ORAM introduces a significant overhead due to the necessity of continuously shuffling data.

```
RHFSetup($1^\lambda$):
 1: $k_{f1} \xleftarrow{\$} \{0,1\}^\lambda,\ k_{f2} \xleftarrow{\$} \{0,1\}^\lambda$
 2: return $K_F = (k_{f1}, k_{f2})$

RHFEncrypt($K_F, \Delta$):
 1: $(k_{f1}, k_{f2}) \leftarrow K_F$
 2: Choose a constant $\varphi$ and set $\zeta \leftarrow \varphi|\Delta|$
 3: Choose a hash $H : \{0,1\}^* \to [1, \zeta]$
 4: ES $\leftarrow$ empty array
 5: for $1 \le i \le \zeta$ do
 6:     ES[$i$] $\leftarrow$ empty set
 7: end for
 8: for each $\delta \in \Delta$ do
 9:     $tag_1 \leftarrow F(k_{f1}, \delta),\ pos \leftarrow H(tag_1)$
10:     $tag_2 \leftarrow F(k_{f2}, \delta)$
11:     ES[$pos$] $\leftarrow$ ES[$pos$] $\cup \{tag_2\}$
12: end for
13: Find the position $j$ with |ES[$j$]| being the
    largest among all the sets in ES
14: $\pi \leftarrow$ |ES[$j$]|.
15: for $1 \le i \le \zeta$ do
16:     while |ES[$i$]| $< \pi$ do
17:         $tag_2 \xleftarrow{\$} \{0,1\}^\lambda$
18:         ES[$i$] $\leftarrow$ ES[$i$] $\cup \{tag_2\}$
19:     end while
20: end for
21: return ES

RHFGetTok($K_F, \delta$):
 1: $(k_{f1}, -) \leftarrow K_F,\ tag_1 = F(k_{f1}, \delta)$
 2: return $etok = tag_1$

RHFRespond($etok$, ES):
 1: $pos \leftarrow H(etok)$
 2: return $res$ = ES[$pos$]

RHFTest($K_F, \delta, res$):
 1: $(-, k_{f2}) \leftarrow K_F$
 2: $tag_2 = F(k_{f2}, \delta)$
 3: if $tag_2 \in res$ then
 4:     return 1
 5: else
 6:     return 0
 7: end if
```

Fig. 3. Our RH-filter Construction

Core Idea. To achieve result-hiding while ensuring high efficiency, our approach prevents access patterns from disclosing membership query results, rather than rendering these patterns entirely indistinguishable. To realize this, we design the encrypted structure ES as an array of sets possessing three properties: ① every element δ, regardless of its presence within the set, can be mapped to a position (pos) within ES in a pseudorandom manner. ② given an element δ and the corresponding position pos, ES[pos] securely records δ only when δ belongs to the set Δ. ③ within ES, each position holds a set, and all these sets have the same size, composed of seemingly random strings. During a query for a specific element δ, we assure that the adversary only observes that the set ES[pos] is accessed, where pos indicates the position corresponding to δ in ES. ES[pos] is passed on to the querier that then obtains the result by locally checking whether ES[pos] records δ or not.

Construction Details. We provide the pseudocodes for our RH-filter construction in Figure 3. The construction relies on a pseudorandom function (PRF) ($F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$) and a hash function ($H : \{0,1\}^* \to [1, \zeta]$, with $\zeta = \varphi|\Delta|$, where φ is a constant).

3.1 Correctness and Security of RH-filter Construction

Our RH-filter construction guarantees computational correctness. Intuitively, during a membership query on an element δ, if $\delta \in \Delta$, the query result must be 1 because both F and H are deterministic. When $\delta \notin \Delta$, as long as F is a secure PRF, the probability distribution of the query returning 1 should be indistinguishable from the distribution where an element is uniformly and randomly selected from $\{0,1\}^\lambda$ and happens to equal to one of the elements stored in ES[$H(etok)$], where $etok$ is the query token corresponding to δ. The security of F also ensures that each element in ES[$H(etok)$] is indistinguishable from one randomly and uniformly selected from $\{0,1\}^\lambda$. Consequently, the probability of false positives is guaranteed to be negligible. Let $\pounds$ denote the RH-filter construction described above. Formally, we have Theorem 6.

Theorem 6. *If $F(k_{f2}, \cdot)$ is a secure PRF, then our RH-filter construction $\pounds$ satisfies the computational correctness defined in Definition 1.*

Proof: The proof is presented in Appendix C.1.

For the security of our RH-filter construction, we have Theorem 7.

Theorem 7. *If F is a secure PRF and the hash function H is modeled as a random oracle, our RH-filter construction $\pounds$ satisfies the semantic security defined in 2.*

Proof: The proof is presented in Appendix C.2.

3.2 Performance of RH-filter Construction

Let π represent the maximum number of elements in Δ that can be mapped by H to the same hash value. **RHFSetup**$(1^\lambda, \Delta)$ incurs a computational time cost of $O(\varphi\pi|\Delta|)$ for generating the array ES, which has a size of $O(\varphi\pi|\Delta|)$. The generation of *etok* in **RHFGetTok**(K_F, δ) involves a computational overhead of $O(1)$, and *etok* itself has a size of $O(1)$. **RHFRespond**$(etok, \text{ES})$ entails an $O(1)$ computational overhead for retrieving *res*, which has a size of $O(\pi)$. **RHFTest**(K_F, δ, res) checks the existence of a tag in *res*, resulting in an $O(1)$ computational overhead.

When H follows a uniformly random distribution, Gonnet [Gon81] and Larson [rL82] have demonstrated that the expected value of π is quite reasonable and grows very slowly ($O(\log\zeta/\log\log\zeta)$ when φ is fixed). In our experiments, assuming $|\Delta| = 5.9 \times 10^7$, we observe that π equals 11 and 8 for φ values of 1 and 2, respectively. Furthermore, in Appendix C.3, we provide evidence that selecting a very small φ value, *e.g.,* 1 or 2, can achieve an exceptional trade-off between the storage requirement and query efficiency, *i.e.,* the trade-off between $\varphi\pi$ and π.

```
    Client
1:  Select key $k_s$ for PRF $F$
2:  Select keys $k_x$ and $k_i$ for PRF $F_p$
3:  T ← empty array indexed by keywords from W
4:  XSet ← empty set
5:  for each $w \in$ W do
6:      $strap \leftarrow F(k_s, w)$
7:      $(k_z, k_e) \leftarrow (F(strap, 1), F(strap, 2))$
8:      t ← empty list, $c \leftarrow 0$
9:      Randomly permute the entries of DB($w$)
10:     for each $id$ in DB($w$) do
11:         $c \leftarrow c + 1$
12:         $xind \leftarrow F_p(k_i, id)$, $z \leftarrow F_p(k_z, c)$
13:         $y \leftarrow xind \cdot z^{-1}$
14:         $k_d \leftarrow F(k_e, c)$, $e \leftarrow k_d \oplus id$
15:         t ← t ∪ $\{(e, y)\}$
16:         $xtag \leftarrow g^{F_p(k_x, w) \cdot xind}$
17:         XSet ← XSet ∪ $\{xtag\}$
18:     end for
19:     T[$w$] ← t
20: end for
21: (TSet, $K_T$) ← TSetSetup(T)
22: $K_F \leftarrow$ RHFSetup($\lambda$)
23: ES ← RHFEncrypt($K_F$, XSet)
24: Send TSet and ES to the server
25: return $K = (k_s, k_x, k_i, K_T, K_F)$

    Server
26: return EDB = (TSet, ES)
```

Fig. 4. HBS.Setup$(1^\lambda, \text{DB}; \perp)$

4 HBS Protocol

In this section, we introduce a Boolean SSE protocol HBS. For simplicity, our focus here is on explaining how HBS manages conjunctions in the form of $w_1 \wedge con(w_2, \cdots, w_n)$, where *con* represents arbitrary conjunction over $w_2, \cdots, w_n$. Here w_i $(2 \le i \le n)$ could indicate a negated term, *e.g.,* $\neg w_2 \wedge w_3 \wedge \neg w_4$, where a negated term returns the documents that do not contain the given keyword.

```
    Client:
1:  $(k_s, k_x, k_i, K_T, K_F) \leftarrow K$
2:  $stag \leftarrow$TSetGetTag$(K_T, w_1)$
3:  $(k_{f1}, k_{f2}) \leftarrow K_F$
4:  Send $stag$ and $k_{f1}$ to the server
5:  $strap \leftarrow F(k_s, w_1)$
6:  $k_z \leftarrow F(strap, 1)$
7:  for $c = 1, 2 \cdots$ and until server sends $stop$ do
8:      xtoken$_c \leftarrow$ empty list
9:      for $i = 2$ to $n$ do
10:         $xtok_{c,i} \leftarrow g^{F_p(k_z,c) \cdot F_p(k_x,w_i)}$
11:         xtoken$_c \leftarrow$ xtoken$_c \cup \{xtok_{c,i}\}$
12:     end for
13:     Send xtoken$_c$ to the server
14: end for

    Server:
15: Responds $\leftarrow$ empty list
16: t $\leftarrow$TSetRetrieve$(stag, \text{TSet})$
17: for $c = 1, \cdots, |\text{t}|$ do
18:     $(-, y) \leftarrow \text{t}[c]$
19:     respond$_c \leftarrow$ empty list
20:     for $i = 2$ to $|\text{xtoken}_c| + 1$ do
21:         $xtok_{c,i} \leftarrow \text{xtoken}_c[i-1]$
22:         $xtag \leftarrow (xtok_{c,i})^y$
23:         $etok \leftarrow F(k_{f1}, xtag)$
                              ▷ Equivalent to executing
            RHFGetTok$(K_F, xtag)$ provided in Figure 3
24:         $res \leftarrow$ RHFRespond$(etok, \text{ES})$
25:         respond$_c \leftarrow$ respond$_c \cup \{(xtag, res)\}$
26:     end for
27:     Responds $\leftarrow$ Responds $\cup \{respond_c\}$
28: end for
29: When last tuple in t is reached, send $stop$ to
    the client.
30: Send Responds to the client

    Client:
31: ck $\leftarrow$ empty set, $k_e \leftarrow F(strap, 2)$
32: for $c = 1$ to |Responds| do
33:     $match_c \leftarrow true$
34:     respond$_c \leftarrow$ Responds$[c]$
35:     for $i = 2$ to $n$ do
36:         $(xtag, res) \leftarrow \text{respond}_c[i-1]$
37:         $b \leftarrow$ RHFTest$(K_F, xtag, res)$
38:         if ($b = 1$ and $w_i$ is a negated term) or
            ($r = 0$ and $w_i$ is a non-negated term) then
39:             $match_c \leftarrow false$
40:             Break the loop for $i$
41:         end if
42:     end for
43:     if $match_c = true$ then
44:         $k_d \leftarrow F(k_e, c)$, ck $\leftarrow$ ck $\cup \{(c, k_d)\}$
45:     end if
46: end for
47: Send ck to the server

    Server:
48: R $\leftarrow$ empty set
49: for each $(c, k_d) \in$ ck do
50:     $(e, -) \leftarrow \text{t}[c]$
51:     $id \leftarrow k_d \oplus e$
52:     R $\leftarrow$ R $\cup \{id\}$
53: end for
54: Send R to the client
```

Fig. 5. HBS.Search$(K, w_1 \wedge con(w_2, \cdots, w_n); \text{EDB})$

HBS is built based on an adapted version of OXT, introduced by Jarecki *et al.* [JJK$^+$] to facilitate the expansion of OXT into multi-client settings. We refer directly to this adapted version as OXT in the section. As depicted in Figures 4 and 5, we employ black text to denote components aligned with OXT and blue text to highlight our innovative parts. The protocol relies on two PRFs: $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ and $F_p : \{0,1\}^\lambda \times \{0,1\}^* \to \mathbb{Z}_p^*$, with p denoting the prime order of a cyclic group $\mathbb{G}_p$. It also incorporates the T-set and our RH-filter, which are utilized in a fully black-box and semi-black-box manner, respectively. Furthermore, we assume that the length of each document identifier is λ.

Setup. In the setup protocol, OXT generates TSet and XSet. TSet is created using the T-set to encrypt the array T indexed by keywords from W. For each keyword $w \in \text{W}$, $\text{T}[w]$ stores a tuple list t of size $|\text{DB}(w)|$. In t, the c-th tuple contains (e, y), where e represents the ciphertext of the c-th document identifier id in $\text{DB}(w)$ (after a random permutation of the document identifiers in $\text{DB}(w)$), and $y \in \mathbb{Z}_p^*$, computed from w, id, and c, will serve as auxiliary information for related search queries. XSet consists of a set of tags, with each tag corresponding to every keyword-document pair in the database. These tags are computed as $xtag = g^{F_p(k_x,w) \cdot F_p(k_i,id)}$ for each pair (w, id). HBS differs from OXT in two respects. First, while OXT employs k_e, derived pseudorandomly from each keyword w, to encrypt each document identifier within $\text{DB}(w)$, HBS takes an additional step by deriving another key, k_d, from k_e and the counter c, and uses k_d to encrypt the c-th document identifier id in $\text{DB}(w)$ via $e = k_d \oplus id$. The benefit of this modification is that it makes retrieving documents using the searched document identifiers more efficient by saving one round of interaction. We will detail this point later. Second, to achieve KPRP-hiding, HBS encrypts XSet using our RH-filter to obtain ES. HBS combines TSet and ES to form the final encrypted database.

We adopt the general assumptions of SSE, whereby each document is encrypted using a symmetric encryption algorithm and tagged with its identifier before being transmitted to the server.

Search. Within a search query $w_1 \wedge con(w_2, \cdots, w_n)$, following OXT, the client generates the $stag$ and xtoken_c for $c = 1, 2 \cdots$ until server sends a stop signal. The server uses the $stag$ to retrieve the tuple list t associated with w_1 from TSet. With $\mathrm{xtoken}_c = \{xtok_{c,i}\}_{i=2}^{n}$ and the element y present in the c-th tuple within t, the server calculates $(xtok_{c,i})^y$. This calculation results in the $xtag$ for the keyword-document pair (w_i, id), where id represents the document identifier concealed in the c-th tuple of t. In the following discussion, we denote this id as $\mathrm{DB}(w_1)[c]$. This particular step is a noteworthy feature of OXT, as it empowers the server to compute the $xtag$ for $(w_i, \mathrm{DB}(w_1)[c])$, all while safeguarding $\mathrm{DB}(w_1)[c]$ from disclosure to either the client or the server. A vital implication of this feature is that it renders OXT well-suited for multi-client settings by preventing the client from learning which document identifiers (other than those included in the final search result) belong to $\mathrm{DB}(w_1)$, which might not be authorized for access. In HBS, we have retained this mechanism from OXT.

In the subsequent step of OXT, the server verifies the existence of $(w_i, \mathrm{DB}(w_1)[c])$ within the database by examining whether the corresponding $xtag$ is present in XSet or not. This process aids in filtering out the documents that satisfy the search query. However, this step in OXT reveals KPRP, as it discloses whether $\mathrm{DB}(w_1)[c]$ belongs to $\mathrm{DB}(w_1) \cap \mathrm{DB}(w_i)$.

To achieve KPRP-hiding, HBS employs the RH-filter and introduces an additional round. In particular, during the initial round of HBS, the data transmitted to the server encompass not only the $stag$ and xtokens but also the first key, k_{f1}, from our RH-filter. Following OXT, for $1 \leq c \leq |\mathrm{t}|$ and $2 \leq i \leq |\mathrm{xtoken}_c| + 1$, the server in HBS obtains the $xtag$ for $(w_i, \mathrm{DB}(w_1)[c])$. Subsequently, utilizing the client's k_{f1}, the server computes the RH-filter token $etok$ for each $xtag$ by executing $F(k_{f1}, xtag)$. It then proceeds with the $\mathrm{RHFRespond}(etok, \mathrm{ES})$ to derive the RH-filter response, res, for each $xtag$. The pairs $(xtag, res)$ generated during iterations for the same c are inserted into a list, $\mathrm{respond}_c$. The produced $\mathrm{respond}_c$ is integrated into a higher-dimensional list, denoted as Responds. The server sends Responds to the client.

Upon receiving Responds, the client initiates the verification process to determine whether $\mathrm{DB}(w_1)[c]$ matches $con(w_2, \cdots, w_n)$ for $1 \leq c \leq |\mathrm{Responds}|$. To verify if $\mathrm{DB}(w_1)[c]$ matches the i-th term for $2 \leq i \leq n$, the client first retrieves $\mathrm{respond}_c$ from Responds$[c]$ and then accesses the $(i-1)$-th entry $(xtag, res)$ within $\mathrm{respond}_c$, The determination of whether the document with the identifier $\mathrm{DB}(w_1)[c]$ contains the keyword w_i precisely corresponds to the output of $\mathrm{RHFTest}(K_F, xtag, res)$. Subsequently, considering whether w_i is a non-negated term or not, the client gains insight into whether $\mathrm{DB}(w_1)[c]$ matches the i-th term. Upon confirming that $\mathrm{DB}(w_1)[c]$ matches $con(w_2, \cdots, w_n)$ (indicated by a flag $match_c$ in the pseudocodes), the client computes the key k_d from k_e and c, where k_e is derived from w_1, and adds (c, k_d) to the set ck. By repeating the above steps for each list within Responds, ck accumulates all the counters and secret keys associated with identifiers from $\mathrm{DB}(w_1)$ that meet the search query criteria.

The set ck is transmitted to the server, where each counter c stored in ck is employed to pinpoint the ciphertext e of $\mathrm{DB}(w_1)[c]$. The server then utilizes the associated secret key in ck to decrypt e, revealing the plaintext identifier. The collection of all decrypted identifiers constitutes the search result R.

Importantly, once R is obtained, the server can proceed to locate the encrypted documents corresponding to the document identifiers in R. These encrypted documents are subsequently returned to the client, along with R. It is worth noting that, while HBS requires an additional round to obtain matching document identifiers compared to OXT, the number of rounds needed for both HBS and OXT remains the same when considering the retrieval of encrypted documents, which are two rounds.

4.1 Correctness of HBS

The correctness of HBS hinges upon the correctness of its T-set, the correctness of our RH-filter, and the distinctiveness of the $xtag$ associated with each keyword-document pair. Specifically, we can establish the distinctiveness of each $xtag$ by relying on the security of PRFs and the hardness of the Decision Diffie-Hellman (DDH) assumption [Bon]. Hereafter, we provide Theorem 8.

Theorem 8. *If the T-set adheres to the correctness presented in Definition 10, the RH-filter is instantiated with the construction in Section 3, F and F_p are secure PRFs, and the DDH assumption holds within the group $\mathbb{G}_p$, then, for any security parameter λ and any PPT adversary $\mathcal{A}$, there exists a negligible function $negl(\lambda)$ such that:*

$$\Pr[\text{SSE}\textsc{Correct}^{\mathsf{HBS}}_{\mathcal{A}}(\lambda) = 1] \leq negl(\lambda)$$

Proof: The proof is presented in Appendix D.1.

4.2 Security of HBS

In this section, we write a conjunction as $q = q[1] \wedge con(q[2], \cdots, q[n])$. We use $\mathcal{Q}$ to record the list of issued search queries. Each search query is expressed as (t, q), where t represents the timestamp of the query q. To facilitate secure analysis, we employ the T-set implementation [CJJ+b] to instantiate the T-set utilized within HBS. The T-set has the leakage profile $\mathcal{L}_T = (\mathcal{L}_T^{tup}(\mathrm{T}) = \sum_{w \in \mathrm{W}} |\mathrm{T}[w]|, \mathcal{L}_T^{tag}(\mathrm{T}, w) = \perp)$, where $\sum_{w \in \mathrm{W}} |\mathrm{T}[w]| = N$ in HBS. Moreover, a T-set query over a keyword w always exposes $\mathrm{T}[w]$. Following [YZCR], we introduce several leakage functions below.

$\mathrm{EP}(q[1])$ is the equality pattern for the *s-term* $q[1]$, which outputs which past search queries use $q[1]$ as the *s-term*. Formally, $\mathrm{EP}(q[1]) = \{t | (t, q') \in \mathcal{Q} \text{ and } q[1] = q'[1]\}$.

For every search query q' that happened before q, if there exists two indices $i \geq 2$ and $j \geq 2$ such that $q[i] = q'[j]$ and $\mathrm{DB}(q[1]) \cap \mathrm{DB}(q'[1])$ is not an empty set, the conditional intersection pattern $\mathrm{IP}(q[1], q[i])$ outputs the timestamp of q', the index j, and the set $\mathrm{DB}(q[1]) \cap \mathrm{DB}(q'[1])$. Formally, $\mathrm{IP}(q) = (\mathrm{IP}(q[1], q[i]))_{i=2}^{n}$, where $\mathrm{IP}(q[1], q[i]) = \{(t, j, \mathrm{DB}(q[1]) \cap \mathrm{DB}(q'[1])| \ (t, q') \in \mathcal{Q}$ and $\exists j \geq 2 : q[i] = q'[j]$ and $\mathrm{DB}(q[1]) \cap \mathrm{DB}(q'[1]) \neq \emptyset\}$.

Intuitively, the setup phase of HBS leaks the size of EDB, which reveals N. Within a conjunction, the T-set query for $q[1]$ only exposes $\mathrm{T}[q[1]]$, revealing $|\mathrm{DB}(q[1])|$ and $\mathrm{EP}(q[1])$ to the server. Each *xtag* is determined uniquely by its corresponding keyword-document pair, which means that *xtag* values expose the repetitiveness of related keyword-document pairs. Specifically, this allows the server to link the current conjunction to the previous ones where the *s-term* matches at least one identifier from $\mathrm{DB}(q[1])$, and one of the *x-terms* corresponds to $q[i]$ for an index i ranging from 2 to n. The information leaked does not exceed what is captured by $\mathrm{IP}(q)$. Formally, we have Theorem 9.

Theorem 9. *If F and F_p are secure PRFs, the T-set implementation has the leakage profile $\mathcal{L}_T = (\mathcal{L}_T^{tup}(\mathrm{T}) = \sum_{w \in \mathrm{W}} |\mathrm{T}[w]|, \mathcal{L}_T^{tag}(\mathrm{T}, w) = \perp)$, the RH-filter is instantiated with the construction provided in Section 3, and the DDH assumption holds in a cyclic group $\mathbb{G}_p$, HBS is $\mathcal{L}_{hbs}$-adaptively secure where*

(A) $\mathcal{L}_{hbs}^{Setup}(\mathrm{DB}) = N$

(B) $\mathcal{L}_{hbs}^{Search}(\mathrm{DB}, q) = (\mathrm{DB}(q), |\mathrm{DB}(q[1])|, \mathrm{EP}(q[1]), \mathrm{IP}(q))$

Proof: The proof is presented in Appendix D.2.

4.3 Performance of HBS

In HBS, the client calculates xtoken_c for c incrementing from 1 until it receives a stop signal from the server. However, if this stop signal is not received promptly, the client may compute an excess of xtoken lists, surpassing $|\mathrm{DB}(w_1)|$. We contend that this is an issue shared with OXT. To address this problem, the client can either store the document count corresponding to keywords locally or request $|\mathrm{t}|$ from the server at the beginning of a search. For the latter method, the number of rounds required for both HBS and HXT in Table 1 should each be increased by 1. In our analysis, aligning with OXT, we assume that the client only needs to compute $|\mathrm{DB}(w_1)|$ xtokens within a search.

Setup. The setup phase creates TSet and ES. With the T-set implementation from [CJJ+b], constructing TSet incurs a computational complexity of $O(N)$, and the size of TSet is also $O(N)$. ES is generated by encrypting XSet, which has a size of N, using our RH-filter. According to Section 3.2, this process carries a computational complexity of $O(\varphi\pi N)$ and yields ES of size $O(\varphi\pi N)$, where the value of $\varphi\pi$ is reasonably small, such as 16 in our experiments.

Search. Within a search query $q = w_1 \wedge con(w_2, \cdots, w_n)$, the T-set query on w_1 incurs $O(1)$ computational complexity for the client and $O(|\mathrm{DB}(w_1)|)$ computational complexity for the server.

Simultaneously, the client computes $(n-1)|\mathrm{DB}(w_1)|$ tokens and transmits them to the server, incurring $O(n|\mathrm{DB}(w_1)|)$ computational and communication overhead. The server then calculates $(n-1)|\mathrm{DB}(w_1)|$ *xtags*, executes RHFGetTok and RHFRespond $(n-1)|\mathrm{DB}(w_1)|$ times, and sends the obtained *xtags* and the data outputted by RHFRespond to the client. It causes $O(n|\mathrm{DB}(w_1)|)$ computational complexity and incurs a communication overhead of $O(\pi n|\mathrm{DB}(w_1)|)$. After that, the client invokes RHFTest $(n-1)|\mathrm{DB}(w_1)|$ times, producing $O(n|\mathrm{DB}(w_1)|)$ computational complexity. Afterward, the client and the server incur a computational and communication complexity of $O(|\mathrm{DB}(q)|)$ to retrieve the documents in $\mathrm{DB}(w_1)$ that match q. In summary, the entire search introduces $O(n|\mathrm{DB}(w_1)|)$ computational overhead and $O(\pi n|\mathrm{DB}(w_1)|)$ communication overhead.

Comparison with Previous Works. In OXT, the required server storage is $O(N)$, and the search overhead is $O(n|\mathrm{DB}(w_1)|)$ in both computation and communication. Compared to OXT, HBS achieves KPRP-hiding at the cost of an increased performance overhead. However, we assert that this level of performance penalty remains quite reasonable. As shown in Table 1, in contrast to HXT, which currently stands as the most efficient solution for achieving KPRP-hiding in terms of overall search and storage efficiency, HBS outperforms HXT across all performance metrics. This achievement is particularly noteworthy considering that HBS ensures negligible false positive rates and supports all Boolean queries (as demonstrated in Section 5), two features not offered by HXT.

Table 1 shows that the performance of HXT involves two parameters for the Bloom filter: ξ and γ, where ξ represents the size of the Bloom filter, and γ is the number of hash functions used by the Bloom filter. In HXT, ξ and γ are set to 29 and 20, which are larger than the values for $\varphi\pi$ and π, respectively. This setting for ξ and γ results in a false positive rate of $\frac{1}{10^6}$ for a single Bloom filter query, which translates to a maximum false positive probability of $\frac{|\mathrm{DB}(w_1)|}{10^6}$ for a conjunctive query in HXT. To reduce this probability, increasing the values of ξ and γ is necessary, but the reduction in false positives would be very limited. In comparison, in HBS, the maximum false positive rate for a search query is the sum of $\frac{\pi|\mathrm{DB}(w_1)|}{2^\lambda}$ and the probability of breaking the security of the pseudorandom function F, which is negligible with λ.

We note that our comparison excludes two other KPRP-hiding schemes, namely Blind Seer [PKV$^+$] and HDXT [YZCR], because that Blind Seer requires a non-constant number of rounds of interactions for a search query, and HDXT incurs excessive storage overhead.

5 Processing All Boolean Queries with HBS

HBS can accommodate arbitrary Boolean queries while maintaining sub-linear search efficiency for Boolean queries in SNF.

Respond to Boolean Queries in SNF. A Boolean expression in SNF can be represented as a disjunction of several conjunctions, each with the *s-term*. Formally, $q = w_1 \wedge \psi(w_2, \cdots, w_n)$ can be expressed as $q_1 \vee \cdots \vee q_\eta$, where q_ε $(1 \le \varepsilon \le \eta)$ is a conjunction with w_1. For instance, $w_1 \wedge (w_2 \vee \neg w_3 \wedge w_4)$ could be rewritten as $(w_1 \wedge w_2) \vee (w_1 \wedge \neg w_3 \wedge w_4)$.

To respond to $w_1 \wedge (w_2 \vee \neg w_3 \wedge w_4)$, following the search process of a conjunction, *e.g.,* $w_1 \wedge w_2 \wedge w_3 \wedge w_4$, as described in Section 4, the client of HBS can obtain the *xtags* for keyword-document pairs (w_i, id) for $2 \le i \le 4$ and each $id \in \mathrm{DB}(w_1)$, along with the corresponding RH-filter responses. The client can then use RHFTest to determine if each *xtag* is a member of XSet, indicating the existence of the corresponding keyword-document pair in the database. With this information, the client can identify which documents in $\mathrm{DB}(w_1)$ satisfy either w_2 or $\neg w_3 \wedge w_4$, determining the positions of these documents within $\mathrm{DB}(w_1)$. Subsequently, the client can calculate the corresponding key, k_d, for each matching position c and insert (c, k_d) into set ck. As illustrated in Figure 5, ck is then transmitted to the server, which computes the search results R.

For a Boolean query in SNF, the search steps closely resemble those for conjunctions involving the same keywords, with only a slight alteration in the criteria used to ascertain whether a document in $\mathrm{DB}(w_1)$ meets the entire search query. Consequently, the asymptotic complexities introduced by the SNF query are the same as those for a conjunction query. In terms of security, a SNF query reveals no more information than a conjunction, as there is no change in the server's perspective.

Respond to All Boolean Queries. HBS inherits its capability to handle arbitrary Boolean queries directly from OXT [CJJ$^+$b]. This is accomplished by representing any Boolean query $\psi(w_1, \cdots, w_n)$ as "True $\wedge\, \psi(w_1, \cdots, w_n)$" (where "True" means matching all documents). After this transformation, the Boolean query is converted into a SNF query and can be processed as described earlier.

In Appendix A, we show that the above strategy can be applied to HXT and HDXT to enhance their query expressiveness. However, due to their inherent design limitations, neither of the two schemes can achieve the same level of query expressiveness as HBS. Specifically, HXT cannot efficiently support Boolean queries involving negated terms. Both HXT and HDXT fail to ensure KPRP-hiding for SNF queries of the form $w_1 \wedge \psi(w_2, \cdots, w_n)$ if ψ involves a disjunction with a single keyword.

6 Multi-client Setting

An important contributions of HBS is the reduction of false positive rates in search results to be negligible. As emphasized by Jarecki *et al.* [JJK+], this contribution holds particular significance in multi-client settings, where noticeable false positive search results can provide clients with access to unauthorized documents. In this section, we aim to underscore the importance of this contribution by providing a straightforward example of extending HBS to function in a multi-client setting. Below, we will solely focus on the SNF queries that HBS can effectively support. We note that part of the idea is borrowed from [JJK+].

We consider a straightforward threat model where the data owner $\mathcal{D}$, who is fully trusted, outsources its database to an honest-but-curious server and permits multiple honest-but-curious clients to submit specific search queries without the possibility of collusion with the server. To operate within this model, during the setup phase, $\mathcal{D}$ invokes HBS.Setup$(1^\lambda, \mathrm{DB}; \perp)$ to obtain the secret key $K = (k_s, k_x, k_i, k_T, k_F)$ and uploads the encrypted database EDB to the server. To authorize a client $\mathcal{U}$ to perform a search query $w_1 \wedge \psi(w_2, \cdots, w_n)$, $\mathcal{D}$ computes $stag \leftarrow \mathrm{TSetGetTag}(K_T, w_1)$, $strap \leftarrow F(k_s, w_1)$, the list $X = \{g^{w_i}\}_{i=2}^{n}$, and sends $\mathcal{M}\mathrm{tok}_u = (stag, strap, X, K_F)$ to $\mathcal{U}$. To perform the search using token_u, $\mathcal{U}$ follows the HBS search protocol provided in Figure 5. However, there is no need for $\mathcal{U}$ to calculate $stag$ (skip Line 2) and $strap$ (skip Line 5), and $\mathcal{U}$ calculates $xtok_{c,i}$ as $X[i-1]^{F_p(k_z,c)}$.

In this setting, we can observe that the client $\mathcal{U}$ will not obtain the document identifiers in $\mathrm{DB}(w_1)$ that do not match the search query. Furthermore, the final search result will not include any documents that do not match the search query except for a negligible probability. In Appendix E, we consider a more complex scenario in which clients may exhibit malicious behavior, a situation also discussed in previous works [JJK+, SLS+].

```
1: Run $\Sigma$.Setup$(1^\lambda; \mathrm{DB})$, after which the client
   gets the secret key $K_\Sigma$ and the state $s_\Sigma$, and the
   server receives the encrypted database $\mathrm{EDB}_\Sigma$.

   Client
2: $K_F \leftarrow \mathrm{RHFSetup}(\lambda)$, $\Delta \leftarrow$ empty set
3: for each $(w, id)$ existing in DB do
4:     $\Delta \leftarrow \Delta \cup \{(w, id)\}$
5: end for
6: $\mathrm{ES}_0 \leftarrow \mathrm{RHFEncrypt}(\Delta)$
7: Send $\mathrm{ES}_0$ to the server
8: $\mathrm{Ca} \leftarrow$ empty table
9: Choose $\rho$ as the maximum capacity of Ca
10: $\tau \leftarrow 0$
11: return $K = (K_\Sigma, K_F)$ and $s = (s_\Sigma, \mathrm{Ca}, \rho, \tau)$

   Server
12: $\mathrm{ESs} \leftarrow$ empty list, $\mathrm{ESs} \leftarrow \mathrm{ESs} \cup \{\mathrm{ES}_0\}$
13: return $\mathrm{EDB} = (\mathrm{EDB}_\Sigma, \mathrm{ESs})$
```

Fig. 6. DHBS.Setup$(1^\lambda, \mathrm{DB}; \perp)$

7 Supporting Dynamic Databases with DHBS

In this section, we introduce DHBS, a KPRP-hiding solution designed for the dynamic setting, allowing updates to keyword-document pairs. However, it is important to clarify that while DHBS shares a similar search logic with HBS, it does not use the *xtag*-based method in HBS. This is because multiple updates to the same keyword-document pair could cause *xtag* duplication, posing a threat to security.

```
1: Run $\Sigma$.Update($K_\Sigma, s_\Sigma, op, w, id$; EDB$_\Sigma$)
                              to update $s_\Sigma$ and EDB$_\Sigma$

   Client
2: Ca[$w, id$] $\leftarrow$ $op$
3: if |Ca| = $\rho$ then
4:     $\tau \leftarrow \tau + 1$, $(k_{f1}, k_{f2}) \leftarrow K_F$
5:     $k_{f1_\tau} \leftarrow F(k_{f1}, \tau)$, $k_{f2_\tau} \leftarrow F(k_{f2}, \tau)$
6:     $K_{F_\tau} \leftarrow (k_{f1_\tau}, k_{f2_\tau})$
7:     $\Delta_\tau \leftarrow$ empty set
8:     for each key $(w', id')$ in Ca do
9:         $op' \leftarrow$ Ca[$w', id'$]
10:        $\Delta_\tau \leftarrow \Delta_\tau \cup \{(op', w', id')\}$
11:    end for
12:    ES$_\tau$ $\leftarrow$ RHFEncrypt($K_{F_\tau}, \Delta_\tau$)
13:    Clear Ca, Send ES$_\tau$ to the server
14: end if
15: return $s = (s_\Sigma, \text{Ca}, \rho, \tau)$

    Server
16: if receives ES$_\tau$ then
17:     ESs $\leftarrow$ ESs $\cup$ {ES$_\tau$}
18: end if
19: return EDB = (EDB$_\Sigma$, ESs)
```

Fig. 7. DHBS.Update(K, s, op, w, id; EDB)

DHBS combines a response-hiding single-keyword DSSE scheme, Σ, with RH-filters, all employed in a fully black-box manner. In Figures 6 - 8, we provide the pseudocodes for DHBS. Note that the pseudocodes for the search protocol use conjunctive search without any negated term as a simplified example, but it can be extended to support all Boolean queries in the same manner as HBS.

During the setup, given the initial database DB, DHBS initiates Σ's setup protocol (*i.e.,* executing Σ.Setup(1^λ, DB)) to produce the secret key K_Σ, the state s_Σ, and the encrypted database EDB$_\Sigma$ supporting single-keyword searches. Simultaneously, it creates RH-filter ES$_0$ with the key K_F (obtained through RHFSetup(1^λ)) for all the keyword-document pairs existing in DB. K_Σ, s_Σ, and K_F are kept by the client, while EDB$_\Sigma$ and ES$_0$ are uploaded to the server.

During an update (op, w, id) where op is *add* or *del*, DHBS simply invokes Σ's update protocol to update s_Σ and EDB$_\Sigma$. The challenge then lies in reflecting this update in the subsequent filtering processes during searches. To address this, a client-side table, Ca, is introduced to cache updates (Ca[w, id] $\leftarrow op$). Ca has two parameters: ρ for maximum capacity and τ for the number of times Ca reaches capacity. When Ca is full, all cached updates must be evicted to the server-side and then cleared from Ca. The eviction of cached updates involves creating a new RH-filter, ES$_\tau$, for all cached update operations, and uploading it to the server. This causes the server to maintain a list of RH-filters ESs = {ES$_0$, ..., ES$_\tau$}. Moreover, note that when creating ES$_\tau$, the secret key used, denoted as K_{F_τ}, is derived from K_F and τ. The step is to conserve client storage space by eliminating the need for storing a unique key pair for each RH-filter.

Within a search (*e.g.,* $w_1 \wedge \cdots \wedge w_n$), the search protocol of Σ is first executed, which enables the client to receive DB(w_1). Next, the client determines which document identifiers in DB(w_1) match the other searched keywords. When checking if the c-th document identifier id in DB(w_1) matches keyword w_i, where $2 \le i \le n$, the client first looks into the local cache Ca: if Ca[w_i, id] exists, then its corresponding op being *add* or *del* determines whether (w_i, id) exists in the database or not. Otherwise, if Ca[w_i, id] is absent, the client queries remote RH-filters, sending RH-filter tokens to the server to retrieve responses regarding (add, w_i, id) or (del, w_i, id) in ES$_j$ from $j = \tau$ to 1 and (wi, id) in ES$_0$. Analyzing RH-filter responses from $j = \tau$ to 1, if the response from ES$_j$ indicates (add, w_i, id) (or (del, w_i, id)), the client concludes the existence (or absence) of (w_i, id) without checking responses from ES$_{j-1}, \cdots,$ ES$_0$. If no relevant updates related to (w_i, id) are found from ES$_\tau$ to ES$_1$, the client proceeds to test the response from ES$_0$, determining the final check result.

7.1 Security of DHBS

In the literature, forward and backward privacy [BMO] are two crucial security notions for DSSE. Forward privacy prevents the server from linking an update to previous searches, while backward privacy ensures that a search does not reveal the entries that were deleted from the database. Note that there are three types of backward privacy: from Type-I that has the least leakage to Type-III which reveals the most information [BMO, YZCR]. The definitions for forward and backward private Boolean DSSE can be found in [YZCR].

```
 1: Run $\Sigma$.Search($K_\Sigma, s_\Sigma, w_1; \mathrm{EDB}_\Sigma$), where the client receives DB($w_1$)

    Client
 2: $(k_{f1}, k_{f2}) \leftarrow K_F$
 3: for $j = \tau$ to 1 do
 4:     $k_{f1_j} \leftarrow F(k_{f1}, j)$, $k_{f2_j} \leftarrow F(k_{f2}, j)$
 5:     $K_{F_j} \leftarrow (k_{f1_j}, k_{f2_j})$
 6: end for
 7: for $c = 1$ to $|\mathrm{DB}(w_1)|$ do
 8:     $id \leftarrow$ the $c$-th identifier in DB($w_1$)
 9:     $\mathrm{DTok}_c \leftarrow$ empty table
10:     for $i = 2$ to $n$ do
11:         for $j = \tau$ to 1 do
                ▷ $\mathrm{DTok}_c[i, j, op]$ is the token for querying the response of whether $\mathrm{ES}_j$ includes $(op, w_i, id)$.
12:             $\mathrm{DTok}_c[i, j, add] \leftarrow$ RHFGetTok($K_{F_j}, (add, w_i, id)$)
13:             $\mathrm{DTok}_c[i, j, del] \leftarrow$ RHFGetTok($K_{F_j}, (del, w_i, id)$)
14:         end for
            ▷ $\mathrm{DTok}_c[i, 0, \perp]$ is the token for querying the response of whether $\mathrm{ES}_0$ includes $(w_i, id)$.
15:         $\mathrm{DTok}_c[i, 0, \perp] \leftarrow$ RHFGetTok($K_F, (w_i, id)$)
16:     end for
17:     Send $\mathrm{DTok}_c$ to the server
18: end for

    Server
19: while receives $\mathrm{DTok}_c$ do
20:     $\mathrm{DR}_c \leftarrow$ empty table
21:     for $i = 2$ to $n$ do
22:         for $j = \tau$ to 1 do
23:             $\mathrm{ES}_j \leftarrow \mathrm{ESs}[j + 1]$
24:             $\mathrm{DR}_c[i, j, add] \leftarrow$ RHFRespond($\mathrm{DTok}_c[i, j, add], \mathrm{ES}_j$)
25:             $\mathrm{DR}_c[i, j, del] \leftarrow$ RHFRespond($\mathrm{DTok}_c[i, j, del], \mathrm{ES}_j$)
26:         end for
27:         $\mathrm{ES}_0 \leftarrow \mathrm{ESs}[1]$
28:         $\mathrm{DR}_c[i, 0, \perp] \leftarrow$ RHFRespond($\mathrm{DTok}_c[i, 0, \perp], \mathrm{ES}_0$)
29:     end for
30:     Send $\mathrm{DR}_c$ to the client
31: end while

    Client
32: R $\leftarrow$ empty set
33: while receives $\mathrm{DR}_c$ do
34:     $match_c \leftarrow true$
35:     $id \leftarrow$ the $c$-th identifier in DB($w_1$)
36:     for $i = 2$ to $n$ do
37:         if $\mathrm{Ca}[w_i, id] = add$ then
38:             Continue with the next iteration for $i$ (Line 36)
                                                  ▷ To test $(w_{i+1}, id)$
39:         else if $\mathrm{Ca}[w_i, id] = del$ then
40:             $match_c \leftarrow false$
41:             Terminate the loop for $i$ and continue with the next iteration for $\mathrm{DR}_c$ (Line 33)
                                                  ▷ To test next identifier
42:         else if $\mathrm{Ca}[w_i, id]$ does not exist then
43:             for $j = \tau$ to 1 do
44:                 $r_a \leftarrow$ RHFTest($K_{F_j}, (add, w_i, id), \mathrm{DR}_c[i, j, add]$)
45:                 $r_d \leftarrow$ RHFTest($K_{F_j}, (del, w_i, id), \mathrm{DR}_c[i, j, del]$)
46:                 if $r_a = 1$ then
47:                     Terminate the loop for $j$ and continue with the next iteration for $i$ (Line 36)
48:                 else if $r_d = 1$ then
49:                     $match_c \leftarrow false$
50:                     Terminate the loop for $j$ and $i$, and continue with the next iteration for $\mathrm{DR}_c$ (Line 33)
51:                 end if
52:             end for
53:             $r \leftarrow$ RHFTest($K_{DF}, (w_i, id), \mathrm{DR}_c[i, 0, \perp]$)
54:             if $r = 1$ then
55:                 Continue with the next iteration for $i$
56:             else if $r = 0$ then
57:                 $match_c \leftarrow false$
58:                 Terminate the loop for $i$ and continue with the next iteration for $\mathrm{DR}_c$ (Line 33)
59:             end if
60:         end if
61:     end for
62:     if $match_c = true$ then
63:         R $\leftarrow$ R $\cup \{id\}$
64:     end if
65: end while
```

Fig. 8. DHBS.Search($K, s, w_1 \wedge \cdots \wedge w_n; \mathrm{EDB}$)

For an update query, DHBS invokes the single-keyword DSSE Σ to update EDB_{Σ}. Meanwhile, the update is either cached locally or included in an RH-filter, which is then sent to the server. From this process, the server only learns the update leakage of Σ and the size of the RH-filter (that reveals the value of ρ). During a search, to test whether a keyword-document pair (w, id) is in the database, the client issues RH-filter queries. Recall that a RH-filter query only reveals when the same query occurs. Same as HBS, from the repetition of RH-filter queries, the server only can link this search to previous ones whose *s-term* matches id and that uses w as one of *x-terms*, which is captured by the conditional intersection pattern IP. As in Section 4.2, let q represent a search. The search leakage profiles of DHBS are composed of $|\mathrm{DB}(q[1])|$, $\mathrm{IP}(q)$, and the search leakage of Σ over $q[1]$.

If Σ satisfies both forward and backward privacy, DHBS reveals neither any association of an update with previous searches nor any information about a deleted entry. Therefore, DHBS inherits forward and backward security properties from Σ. This also indicates that the type of backward privacy that DHBS can achieve is equivalent to that of Σ.

Client-side Security. In the earlier discussion, we highlight that when developing DHBS, we opt not to utilize the *xtag*-based method employed in HBS. However, we acknowledge that this decision involves a trade-off, which allows the client within a search to obtain all document identifiers matched by the keyword w_1. Nevertheless, DHBS maintains negligible false positive rates in the final search results, ensuring that the client will not receive the contents of documents that do not match the search query, except for a negligible probability. In this context, DHBS still offers superior client-side security compared to the Bloom-filter-based KPRP-hiding solutions [PKV+, JcCQ+22, LPS+] while achieving the same level of client-side security as HDXT.

7.2 Performance of DHBS

For an update query, adding it to the cache Ca incurs a computational overhead of $O(1)$ for the client. When the cache size exceeds ρ, a new RH-filter for ρ update operations is created and transmitted to the server, incurring $O(\varphi\pi\rho)$ computational and communication overhead. On average, the overhead per update amortizes to $O(\varphi\pi)$. To respond to a search query, following the single-keyword search on w_1, the client needs to verify the existence of $(n-1)|\mathrm{DB}(w_1)|$ keyword-document pairs in the database. For each pair checked, the client sends $2\tau+1$ RH-filter tokens to the server that returns $2\tau+1$ RH-filter responses. Subsequently, in the worst-case scenario, the client must test all the received RH-filter responses. Assuming the search protocol of Σ costs P_{Σ} for computational overhead and M_{Σ} for communication overhead per execution, the overall computational and communication overhead incurred by DHBS becomes $O(P_{\Sigma} + n\tau|\mathrm{DB}(w_1)|)$ and $O(M_{\Sigma} + \pi\tau n|\mathrm{DB}(w_1)|)$, respectively. In table 1, we assume that Σ in DHBS is instantiated using MITRA [CPPJ], a forward and Type-II backward private DSSE single-keyword DSSE scheme, consistent with HDXT.

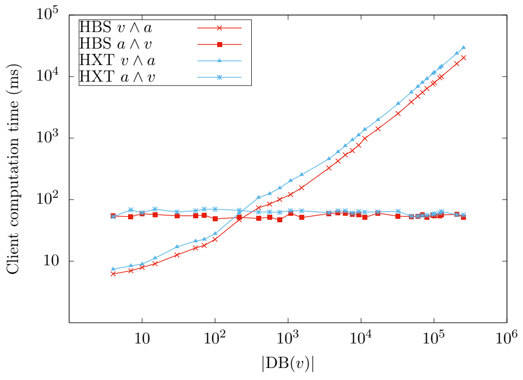
The value τ is determined by the number of updates occurring after the setup divided by ρ, indicating that as the number of updates increases, the rate of decline in DHBS's search performance could periodically accelerate. In Section 8.2, the experimental results show that even after 10^6 updates, DHBS remains highly efficient. However, to ensure that DHBS consistently maintains high efficiency regardless of N^u, a viable approach is to integrate updates from multiple RH-filters into a single RH-filter. To achieve this, during the eviction process of updates, apart from creating a RH -filter for the cached updates, all cached updates should be encrypted and uploaded to the server. Then, when the client is idle and has sufficient storage available, assuming it intends to merge from ES_{x_1} to ES_{x_2}, it can download the encrypted updates corresponding to $\mathrm{ES}_{x_1}, \cdots, \mathrm{ES}_{x_2}$, decrypt them, create a new RH-filter ES_{x_3} for them, upload ES_{x_3} to the server, and remove $\mathrm{ES}_{x_1}, \cdots \mathrm{ES}_{x_2}$. In subsequent search processes, the steps for querying from ES_{x_1} to ES_{x_2} are replaced with querying ES_{x_3}, reducing the search burden by a factor of $(x_2 - x_1)$.

8 Experimental Evaluation

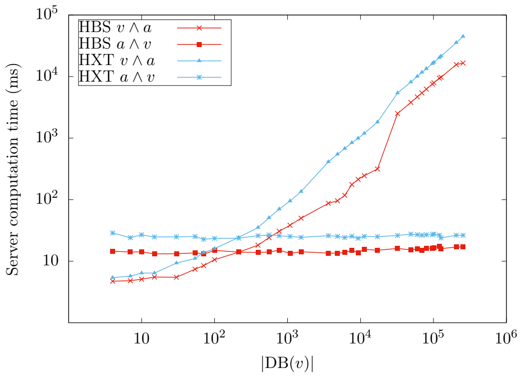
In this section, we present the experimental results to demonstrate the practicality of HBS and DHBS. To evaluate the performance of HBS, we select HXT [LPS+] as the baseline, which is recognized in the literature as the most efficient KPRP-hiding scheme in terms of overall storage and search efficiency. However, it is important to note that, compared to HBS, HXT has noticeable false positive rates and considerably weaker query expressiveness. In addition, we emphasize that we refrain from comparing the search efficiency of DHBS with that of the other KPRP-hiding dynamic solution, HDXT [YZCR]. For the dataset used in this experiment, HDXT requires storage capacity exceeding the server's storage limit of 4TB, which is several hundred times greater than that required by DHBS[2].

Implementation. We utilize the C++ implementation of HXT from [YZCR] and implement HBS and DHBS in the same manner. Specifically, we utilize the Crypto++ library [Dai] to implement the cryptographic operations: AES-ECB-128 + SHA-256 for PRFs, SHA-256 for hash functions, and the elliptic curve secp256r1 for group operations. RocksDB [Fac] is employed for storage, and gRPC [Goo] facilitates communication between the client and the server. Following [LPS+], we set γ and ξ for the Bloom filter to 20 and 29, respectively. For RH-filters used by HBS, we set parameter φ to 2, which results in π being 8 for the dataset used. These prototypes are deployed on two machines running Ubuntu 18.04 LTS. The server machine featured 16× Intel Core Processor (Broadwell, IBRS 2.15GHz), 64GB RAM, and 4TB of hard drive space. The client machine has 16 cores (Intel Core i9-9900 CPU 3.10 GHz), 32GB memory, and 400GB disk space.

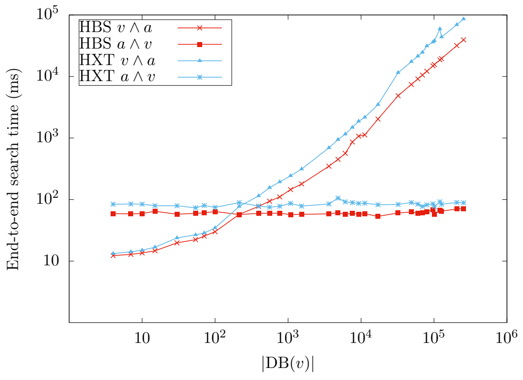
Dataset. Our experiments utilize a real-world dataset from Wikimedia [wik], comprising 124989 keywords, 453337 documents, and 59434360 keyword-document pairs.



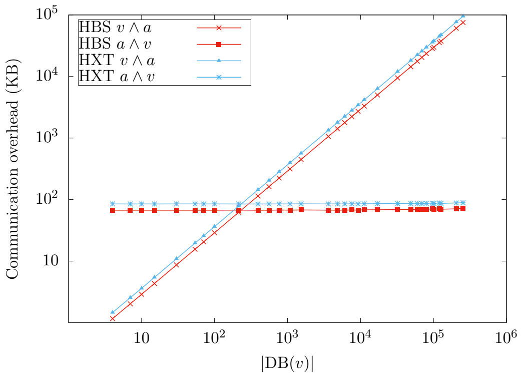
(a) Client



(b) Server



(c) End-to-end



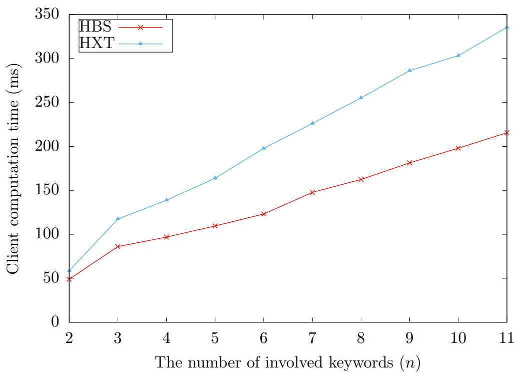
(d) Communication Overhead

Fig. 9. Search Performance of 2-conjunctions

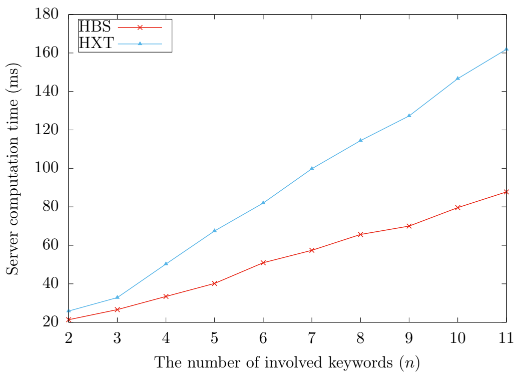
8.1 Search Performance of Static Solutions

In this subsection, we conduct a comprehensive evaluation of search performance for conjunctions of two keywords and n ($n \geq 2$) keywords, respectively, comparing HBS and HXT. The evaluation

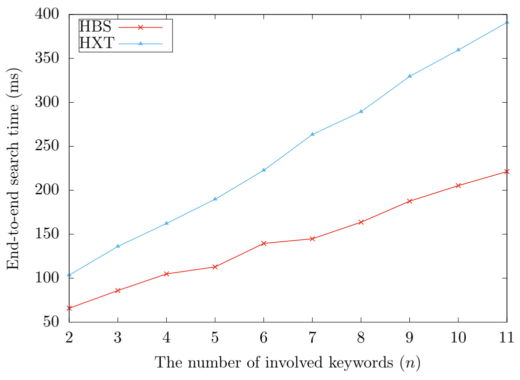
[2] For very dense datasets, this gap can be reduced, but it remains substantial, *e.g.,* several dozen times larger than that of DHBS.



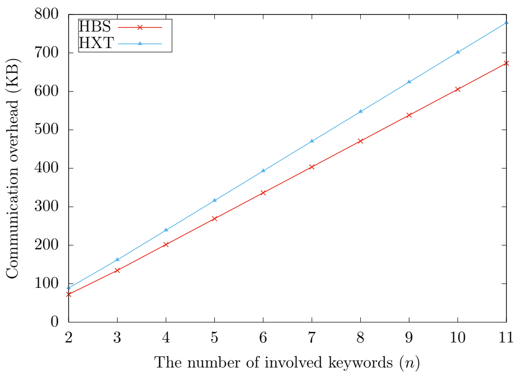
(a) Client



(b) Server



(c) End-to-end



(d) Communication Overhead

Fig. 10. Search Performance of n-conjunctions

metrics include the computational time spent by the client and the server and the end-to-end search latency (total time for search completion). Each search is repeated 100 times, and the average time is reported. Moreover, we evaluate the communication overhead for each unique search query by quantifying the volume of data exchanged between the client and the server.

2-conjunctions Following the methodology in [LPS+], we select two terms: v and a. The term v is a variable with $|\mathrm{DB}(v)|$ increasing from 4 to 254651, while $|\mathrm{DB}(a)|$ is fixed at 232. We evaluate the search performance for conjunctions $v \wedge a$ and $a \wedge v$. The experimental results are presented in Figure 9, with logarithmic scaling in both the X-Axis and Y-Axis.

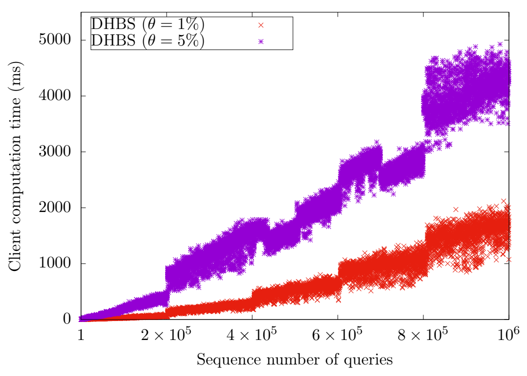
The results align with the asymptotic complexities given in Section 4.3. As $|\mathrm{DB}(v)|$ increases, both the computational time and the communication overhead for $v \wedge a$ rise. In contrast, the overhead for $a \wedge v$ remains almost constant. This indicates that the search overhead for both HBS and HXT is directly proportional to the number of documents matched by the *s-term*, while it is independent of the number of documents matched by the *x-term*.

Comparing HBS and HXT, the figures demonstrate that HBS outperforms HXT in both the computational and communication overhead for each 2-conjunction search. Specifically, for $v \wedge a$, the end-to-end search efficiency of HBS is, on average, 136% better than that of HXT.

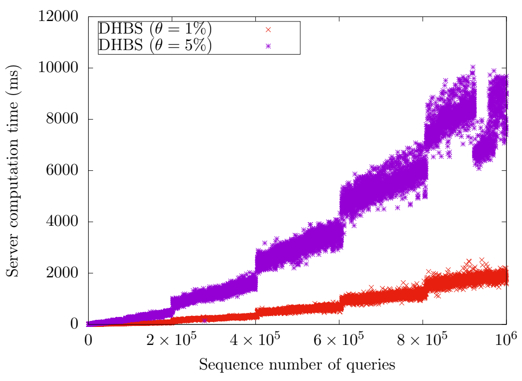
Notably, in Figure 9(b), there is a significant increase in the server's computational time for both HBS and HXT as $|\mathrm{DB}(v)|$ grows from nearly 2×10^4 to over 3×10^4. The increase is more pronounced for HBS. Here, we explain that, in our experimental environment, data transmission speed is considerably slower than computation speed. When $|\mathrm{DB}(v)|$ reaches around 2×10^4, to prevent the buffer, used for storing the data to be transmitted, from overflowing, the server automatically suspends some computation threads and waits, resulting in a sudden increase in its computational time. Due to the higher computational efficiency of HBS compared to HXT, the threads on the server side of HBS wait longer, leading to a more significant increase in the time taken. This computational time can be improved in a network environment with better bandwidth.

***n*-conjunctions** For n-conjunctions, we select $n-1$ terms: $v_1, \cdots, v_{n-1}$. Subsequently, we conduct the conjunction $a \wedge v_1 \wedge \cdots \wedge v_{n-1}$, with term a consistent with the experiments for 2-conjunctions. The experimental results are depicted in Figures 10.

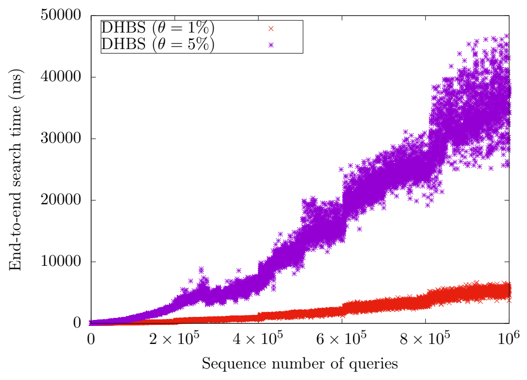
The results convey that computational and communication overhead for both HBS and HXT increases proportionally with n, aligning with asymptotic complexities. Furthermore, these figures demonstrate that for n-conjunctions, HBS outperforms HXT, with the performance advantage of HBS rising from 57% at $n = 2$ to 77% (at $n = 11$).



(a) Client



(b) Server



(c) End-to-end

Fig. 11. Search Time of DHBS

8.2 Search Performance of DHBS

In this sub-section, we evaluate the search performance of our dynamic solution DHBS. For this experiment, we instantiate the single-keyword DSSE scheme Σ with MITRA [CPPJ]. Also, recall that DHBS requires a local cache. The cache capacity (ρ) is set to 2×10^5, utilizing approximately 1.2MB of client-side storage space.

Similar to the methodology used in [YZCR], we generate two traces of 10^6 queries. Each query is either a conjunctive query $w_1 \wedge \cdots \wedge w_5$ with 1% probability, where w_i for $1 \leq i \leq 5$ represents a distinct keyword, or an update query with a probability of 99%. We record the time taken for each conjunctive query as our experimental results. The distinguishing factor between the two traces lies in the ratio of $|\mathrm{DB}(w_1)|$ to the total number of updates. This ratio (denoted as θ) amounts to approximately 1% in the first trace and 5% in the second. Our decision to generate these traces is rooted in the performance analysis of DHBS detailed in Section 7.2, revealing that the time spent processing $w_1 \wedge \cdots \wedge w_5$ is predominately influenced by $|\mathrm{DB}(w_1)|$ and τ. Adjusting θ enables us to illustrate the impact of $|\mathrm{DB}(w_1)|$ on search performance.

The experimental results are depicted in Figure 11. We observe a surge in search time approximately every ρ (*i.e.,* 2×10^5) updates for each trace, indicating the influence of τ on search performance. The search time exhibits some instability due to network bandwidth fluctuations and dynamic thread pool management. Furthermore, comparing the results corresponding to the two traces reveals the impact of $|\mathrm{DB}(w_1)|$ on DHBS's search performance. When θ increases from 1% to 5%, the average end-to-end search time grows by 4.2×, with client and server computation time increasing by approximately 3.5×.

Importantly, when contrasting DHBS's search time with that of HBS and HXT, as shown in Figure 9, we observe that after experiencing 10^6 updates, the search efficiency of DHBS remains quite acceptable. In the case of $\theta = 5\%$, the maximum time spent on a single search is less than $50s$. It is important to note that we set θ relatively high in this experiment. Recall that w_1 should be the least frequently occurring keyword among the involved keywords, so in practice, $|\mathrm{DB}(w_1)|$ can be much lower in many cases, indicating that DHBS can exhibit superior search performance.

Table 2. Comparison of Server Storage

	HBS	DHBS	HXT [LPS+]
Setup	21.9 GB	17.5 GB	31.9 GB
After 10^6 updates	Static	17.6 GB	Static

8.3 Storage Overhead

In this sub-section, we evaluate the storage overhead of HBS and DHBS, comparing them with that of HXT.

We evaluate the client storage required by DHBS, which amounts to 3.2 MB. There is no need to assess the client storage required by HBS and HXT, as they only require storing several secret keys on the client side.

We measure the server storage utilized by the three schemes mentioned above. Additionally, as given in Table 1, the server storage used by DHBS grows with the number of updates. Consequently, we also provide DHBS's server storage usage after experiencing 10^6 updates. The results for server storage requirements are displayed in Table 2. They indicate that HXT's storage overhead is worse than HBS and DHBS by 46% and 82%, respectively. DHBS exhibits the best storage efficiency among the three schemes after the setup, because, as discussed in Section 7, DHBS does not utilize the *xtag*-based method used by HBS and HXT. Moreover, the server storage required by DHBS only increases by 0.1GB after 10^6 updates. This can demonstrate that DHBS demonstrates good scalability in terms of server-side storage.

9 Related Work

SSE has been under investigation for over two decades, since it was first introduced by Song *et al.* [SWP] in 2000. While conventional cryptographic techniques such as fully homomorphic encryption [Gen, vDGHV] and ORAM [Gol, SvDS+] could theoretically achieve encrypted searches, SSE offers a distinctive advantage: it aims for a much more practical performance by permitting a slight leakage to the server. A significant challenge confronting the SSE community is the imperative to bolster security in SSE schemes, driven by leakage cryptanalysis [IKK, CGPR, BKMb, OKa, OKb, GPP, ZKP, PWLP]. Developing approaches to enhance security while preserving practical performance is of paramount importance but often poses considerable challenges.

The majority of prior research in SSE has predominantly revolved around single-keyword search scenarios [SWP, CGKO, KPR, CJJ+a, SPS, Bos, BMO, SDY+20, SYL+, CPPJ, ZSL+b, SSL+, YCR].

Conversely, the investigation into more expressive search queries has been somewhat overlooked. This paper specifically addresses the support for Boolean searches. In this field, early solutions [GSW, BKMa, BLL] are limited to support conjunctive queries and require overhead linear with the total number of documents. Since 2013, research on Boolean SSE has allowed for a slight information leakage to achieve sub-linear search efficiency. Existing Boolean SSE schemes are built mainly based on several frameworks: OXT-based solutions [CJJ+b, LPS+, PM, YZCR, BTR+], Tree-based indexing solutions [PKV+, LL, WL18, JcCQ+22], and IEX-based solutions [KMa, PPSY]. Note that HBS and DHBS are constructed upon the OXT-based framework.

OXT-based Solutions. In 2013, Cash *et al.* [CJJ+b] introduced OXT, supporting all Boolean queries in SNF with sub-linear search efficiency, by trading off a small degree of security. However, as analyzed by Zhang *et al.* [ZKP], the leakage from OXT includes KPRP, which could be exploited by attackers to recover plaintext information about searched keywords. In 2018, Lai *et al.* [LPS+] proposed HXT, which hides KPRP by combining the Bloom filter and symmetric hidden vector encryption to optimize the filtering process in OXT. In 2021, Patranabis and Mukhopadhyay [PM] extended OXT to support updates, resulting in a scheme named ODXT. ODXT supports both forward and backward privacy (Type-II) but fails to achieve KPRP-hiding. In 2023, Yuan *et al.* [YZCR] designed HDXT, the first conjunctive DSSE scheme with the guarantee of KPRP-hiding. HDXT also ensures forward and backward privacy (Type-I and Type-II). However, it requires extensive server storage, which grows linearly with the number of all possible keyword-document pairs that may exist in the database, thus affecting its scalability to large-scale sparse databases. The same year, Bag *et al.* [BTR+] raised an approach named TWINSSE to convert schemes that support conjunctive queries into schemes that support arbitrary Boolean queries by pre-selecting meta-keywords. They also instantiated their approach with OXT. However, there is no evidence that their method can inherit the KPRP-hiding property from KPRP-hiding conjunctive schemes like HXT. Furthermore, its applicability to dynamic databases is limited due to the need for pre-selected meta-keywords.

Tree-based Indexing Solutions. In 2014, Pappas *et al.* [PKV+] introduced tree-based indexing to support all Boolean queries and developed the scheme Blind Seer. For conducting a Boolean search, they combined Bloom filters and garbled-circuit-based secure computation [LP09] to securely traverse the index tree, thereby disclosing only some search pattern. However, their approach makes the entire search cost $O(\log|\mathrm{D}|)$ round complexity. The subsequent tree-based solutions, including IBTree [LL], VBTree [WL18], and Rphx [JcCQ+22], have moved away from employing secure two-party computation, achieving significantly enhanced performance. However, both IBTree and VBTree introduce more severe search leakage than KPRP, revealing the documents matched by each searched keyword. In contrast, Rphx ensures KPRP-hiding by delegating the task of traversing the index tree to Intel SGX enclaves [CD16]. Note that Intel SGX has some security vulnerabilities, and no perfect countermeasure is available [FYDX21]. Furthermore, only VBTree supports dynamic databases while ensuring both forward and backward privacy (Type-II).

IEX-based Solutions. In 2017, Kamara and Moataz [KMa] proposed IEX-series schemes, which are grounded in the inclusion-exclusion principle of set theory. IEX distinguishes itself by achieving sub-linear complexity to support all types of Boolean queries while necessitating only a single round of interaction. Nevertheless, IEX-series schemes do not guarantee KPRP-hidng. Subsequently, in 2021, Patel *et al.* [PPSY] put forward CNFFilter, aiming to enhance the security of IEX; however, it still falls short of fully concealing KPRP. For this series of schemes, Kamara and Moataz [KMa] extended their IEX approach to the dynamic setting, achieving solely forward privacy without considering backward privacy.

10 Conclusion

In this paper, we propose HBS, a Boolean SSE scheme that represents the first KPRP-hiding solution that ensures negligible false positive rates in low server storage overhead. To develop HBS, we introduce a novel cryptographic tool, RH-filter, which distinguishes itself as the inaugural solution supporting computationally correct membership queries with nearly constant overhead while ensuring the confidentiality of query results. Through extensive analysis and experiments, we showcase that HBS also outperforms previous KPRP-hiding solutions in terms of query expressiveness and performance. Furthermore, we extend HBS to operate in the dynamic setting, where the resulting scheme maintains KPRP-hiding while ensuring both forward and backward privacy.

References

BKMa. L. Ballard, S. Kamara, and F. Monrose. Achieving efficient conjunctive keyword searches over encrypted data. In *7th International Conference on Information and Communications Security, ICICS 2005, Beijing, China, December 10-13, 2005*, pages 414–426.

BKMb. L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, CA, USA, February 23-26, 2020.*

BLL. J. W. Byun, D. H. Lee, and J. Lim. Efficient conjunctive keyword search on encrypted data storage system. In *3rd European PKI Workshop: Theory and Practice, EuroPKI 2006, Turin, Italy, June 19-20, 2006*, pages 184–196.

Blo70. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

BMO. R. Bost, B. Minaud, and O. Ohrimenko. 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 03, 2017*, pages 1465–1482.

Bon. D. Boneh. The decision diffie-hellman problem. In *3rd International Algorithmic Number Theory Symposium, ANTS 1998, Portland, Oregon, USA, June 21-25,, 1998*, pages 48–63.

Bos. R. Bost. $\Sigma o \varphi o \varsigma$–forward secure searchable encryption. In *23rd ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, Vienna, Austria, October 24-28, 2016*, pages 1143–1154.

BTR$^+$. A. Bag, D. Talapatra, A. Rastogi, S. Patranabis, and D. Mukhopadhyay. Two-in-one-sse: Fast, scalable and storage-efficient searchable symmetric encryption for conjunctive and disjunctive boolean queries. In *23rd Privacy Enhancing Technologies Symposium, PETS 2023, Lausanne, Switzerland, July 10–15, 2023*, page 115–139.

CD16. V. Costan and S. Devadas. Intel sgx explained. *http://eprint.iacr.org/2016/086*, 2016.

CGKO. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. 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, 2006*, pages 79–88.

CGPR. D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, Denver, CO, USA, October 12-16, 2015*, pages 668–679.

CJJ$^+$a. D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, and M. Steiner. 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, 2014*, pages 23–26.

CJJ$^+$b. D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *33rd Annual Cryptology Conference, CRYPTO 2013, Santa Barbara, CA, USA, August 18-22, 2013*, pages 353–373.

CPPJ. J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili. 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, 2018*, pages 1038–1055.

Dai. W. Dai. Crypto++ library 8.2. `https://www.cryptopp.com`.

Fac. Facebook. Rocksdb 6.6.4. `https://github.com/facebook/rocksdb/tree/v6.6.4`.

FYDX21. S. Fei, Z. Yan, W. Ding, and H. Xie. Security vulnerabilities of sgx and countermeasures: A survey. *ACM Computing Surveys*, 54(6):1–36, 2021.

Gen. C. Gentry. Fully homomorphic encryption using ideal lattices. In *41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178.

Gol. O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *19th Annual ACM Conference on Theory of Computing, STOC 1987, New York, NY, USA, 182-194, 1987*, pages 182–194.

Gon81. G. H. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, 28(2):289–304, 1981.

Goo. Google. grpc. `https://github.com/grpc/grpc`.

GPP. Z. Gui, K. G. Paterson, and S. Patranabis. Rethinking searchable symmetric encryption. In *44th IEEE Symposium on Security and Privacy, S&P 2023, San Francisco, CA, USA, May 22-25, 2023*, pages 485–502.

GSW. P. Golle, J. Staddon, and B. Waters. 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, 2004*, pages 31–45.

IKK. M. S. Islam, M. Kuzu, and M. Kantarcioglu. 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, 2012.*

JcCQ$^+$22. Q. Jiang, E. chien Chang, Y. Qi, S. Qi, P. Wu, and J. Wang. Rphx: Result pattern hiding conjunctive query over private compressed index using intel sgx. *IEEE Transactions on Information Forensics and Security*, 17:1053–1068, 2022.

JJK$^+$. S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *20th ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, November 4–8, 2013*, page 875–888.

KLSN. S. K. Kermanshahi, J. K. Liu, R. Steinfeld, and S. Nepal. Generic multi-keyword ranked search on encrypted cloud data. In *24th European Symposium on Research in Computer Security, ESORICS 2019, Luxembourg, September 23–27, 2019*, pages 322–343.

KMa. S. Kamara and T. Moataz. 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, 2017*, pages 94–124.

KMb. S. Kamara and T. Moataz. 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, 2019*, pages 183–213.

KPR. S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *19th ACM conference on Computer and communications security, CCS 2012, Raleigh, NC, USA, October 16-18, 2012*, pages 965–976.

LL. R. Li and A. X. Liu. 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, 2017*, pages 697–708.

LP09. Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, 22:161–188, 2009.

LPS$^+$. S. Lai, S. Patranabis, A. Sakzad, J. K. Liu, D. Mukhopadhyay, R. Steinfeld, S.-F. Sun, D. Liu, and C. Zuo. 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, 2018*, pages 745–762.

MZK. X. Meng, H. Zhu, and G. Kollios. Top-k query processing on encrypted databases with strong security guarantees. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 353–364.

OKa. S. Oya and F. Kerschbaum. 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, 2021*, pages 127–142.

OKb. S. Oya and F. Kerschbaum. 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, 2022*, pages 2407–2424.

PKV$^+$. V. Pappa, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *35th IEEE Symposium on Security and Privacy, S&P 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 359–374.

PM. S. Patranabis and D. Mukhopadhyay. Forward and backward private conjunctive searchable symmetric encryption. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, February 21-25, 2021.*

PPSY. S. Patel, G. Persiano, J. Y. Seo, and K. Yeo. 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, 2021*, pages 577–607.

PPYY. S. Patel, G. Persiano, K. Yeo, and M. Yung. 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, 2019*, pages 79–93.

PW. D. Pouliot and C. V. Wright. 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, 2016*, pages 1341–1352.

PWLP. R. Poddar, S. Wang, J. Lu, and R. A. Popa. Practical volume-based attacks on encrypted databases. In *5th IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*, pages 354–369.

rL82. P. Åke Larson. Expected worst-case performance of hash files. *The Computer Journal*, 25(3):347–352, 1982.

SDY$^+$20. X. Song, C. Dong, D. Yuan, Q. Xu, and M. Zhao. Forward private searchable symmetric encryption with optimized i/o efficiency. *IEEE Transactions on Dependable and Secure Computing*, 17(5):912–927, 2020.

SLS$^+$. S.-F. Sun, J. K. Liu, A. Sakzad, R. Steinfeld, and T. H. Yuen. An efficient non-interactive multi-client searchable encryption with support for boolean queries. In *21st European Symposium on Research in Computer Security, ESORICS 2016, Heraklion, Greece, September 26-30, 2016*, pages 173–195.

SPS. E. Stefanov, C. Papamanthou, and E. Shi. 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, 2014*.

SSL$^+$. S.-F. 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 *28th Annual Network and Distributed System Security Symposium, NDSS 2021, February 21-25, 2021*, pages 5–20.

SvDS$^+$. E. Stefanov, M. van Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious ram protocol. In *20th ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, November 4–8, 2013*, page 299–310.

SWP. D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *21st IEEE Symposium on Security and Privacy, S&P 2000, Berkeley, CA, USA, May 14-17, 2000*, pages 44–55.

SYL$^+$. S.-F. Sun, X. Yuan, J. K. Liu, R. Steinfeld, A. Sakzad, V. Vo, and S. Nepal. 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, 2018*, pages 763–780.

vDGHV. M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2010, French Riviera, May 30–June 3, 2010*, pages 24–43.

WCa. J. Wang and S. Chow. Simple storage-saving structure for volume-hiding encrypted multi-maps. In *35th IFIP Annual Conference on Data and Applications Security and Privacy, DBSec 2021, Calgary, Canada, July 19-20, 2021*, pages 63–83.

WCb. J. Wang and S. S. M. Chow. Forward and backward-secure range-searchable symmetric encryption. In *22nd Privacy Enhancing Technologies Symposium, PETS 2022, Sydney, Australia, July 11-15, 2022*, pages 28–48.

wik. Wikimedia dump service. `https://dumps.wikimedia.org/enwiki/`.

WL18. Z. Wu and K. Li. Vbtree: forward secure conjunctive queries over encrypted data for cloud computing. *The International Journal on Very Large Data Bases*, 28(1):25–46, 2018.

WNL$^+$. X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *21st ACM SIGSAC Conference on Computer and Communications Security, CCS 2014, Scottsdale, AZ, USA, November 3-7, 2014*, pages 215–226.

YCR. D. Yuan, S. Cui, and G. Russello. 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, 2022*, pages 587–605.

YZCR. D. Yuan, C. Zuo, S. Cui, and G. Russello. Result-pattern-hiding conjunctive searchable symmetric encryption with forward and backward privacy. In *23rd Privacy Enhancing Technologies Symposium, PETS 2023, Lausanne, Switzerland, July 10–15, 2023*, pages 40–58.

ZKP. Y. Zhang, J. Katz, and C. Papamanthou. 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, 2016*, pages 707–720.

ZSL$^+$a. C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, and J. Pieprzyk. 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, 2018*, pages 228–246.

ZSL$^+$b. C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, and J. Pieprzyk. 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, 2019*, pages 283–303.

A Limitations on Query Expressiveness in HXT and HDXT

HXT and HDXT represent two solutions for KPRP-hiding within the search framework of OXT. A notable feature of OXT is its efficient support for arbitrary Boolean queries in SNF. Both HXT and HDXT focus on handling conjunctive queries. However, the authors (Lai *et al.* [LPS$^+$] and Yuan *et al.* [YZCR]) did not provide clear specifications regarding the potential extension of HXT and HDXT to achieve the same level of query expressiveness as OXT. In this section, we elucidate how HXT and HDXT do not fully meet this standard.

In a SNF query $w_1 \wedge \psi(w_2, \cdots, w_n)$, OXT initially identifies documents matching w_1 and then refines the search result by testing whether each of these documents also satisfies $\psi(w_2, \cdots, w_n)$. This refinement process involves querying the membership of keyword-document pair (w_i, id) within the database for each $id \in \mathrm{DB}(w_1)$ and $2 \leq i \leq n$. However, this process in OXT exposes the membership status of each keyword-document pair (w_i, id) to the server, consequently undermining KPRP. For HXT and HDXT, their effectiveness in achieving KPRP-hiding for a conjunctive search $w_1 \wedge \cdots \wedge w_n$ hinges on how they test whether each document $id \in \mathrm{DB}(w_1)$ matches the other $n-1$ keywords. Hereafter, we explain how their methodologies in this crucial aspect could fail to be extended to test whether an $id \in \mathrm{DB}(w_1)$ satisfies $\psi(w_2, ..., w_n)$.

A.1 HXT

HXT employs a Bloom filter and a Symmetric Hidden Vector Encryption (SHVE) scheme to execute the core functionality mentioned above. In the following, we provide a brief overview of the Bloom filter and SHVE. Subsequently, we describe how HXT utilizes the two tools to achieve its objectives. Finally, we discuss the limitations on query expressiveness in HXT.

A Bloom filter represents a set using a ζ-bit vector. Initially, an empty Bloom filter sets all bits to 0. The insertion of an element involves mapping it to γ positions within the Bloom filter using uniformly random hash functions and setting the corresponding bits to 1. To query membership for an element, the positions corresponding to the element are computed. If all bits at these positions are set to 1, the element is determined to belong to the set.

SHVE supports the encryption of a vector $\mathbf{v_a}$ into a ciphertext, after which it enables the generation of a query token linked to another vector $\mathbf{v_p}$. Utilizing this generated token and the ciphertext, anyone can ascertain whether $\mathbf{v_p}$ matches with $\mathbf{v_a}$ at specified positions, that is, whether $\mathbf{v_p}[i] = \mathbf{v_a}[i]$ for every specified position i. Importantly, SHVE ensures the positions where $\mathbf{v_p}$ and $\mathbf{v_a}$ match (or mismatch) will not be disclosed in the event of a negative result.

During the setup phase, HXT incorporates the tags of keyword-document pairs existing in the database into a Bloom filter BF and utilizes SHVE to encrypt BF to a ciphertext, which is then outsourced to the server. In a conjunctive search $w_1 \wedge \cdots w_n$, the client and the server collaboratively generate Bloom filters $\{BF_{id}\}_{id \in \mathrm{DB}(w_1)}$, where BF_{id} represents a Bloom filter containing the tags of keyword-document pairs in $\{(w_i, id)\}_{i=2}^{n}$. Subsequently, the client transmits a SHVE query token to the server to inquire whether each BF_{id} matches BF in the positions where BF_{id} stores 1. If the above query returns *true*, it signifies the existence of the keyword-document pair (w_i, id) in the database for each $2 \leq i \leq n$. Throughout this process, the server can ascertain whether the tested document identifier id simultaneously matches all the other $n-1$ keywords, but if id does not simultaneously match, the server remains unaware of which w_i $(2 \leq i \leq n)$ is contained (or not contained) in id, thereby ensuring KPRP-hiding.

Lai *et al.* [LPS+] did not discuss the extensibility of HXT to support other types of Boolean queries. However, given that HBS proposed in this paper is also based on the OXT framework, it is natural to explore whether the method discussed in Section 5 for extending HBS to support SNF queries could also apply to HXT. Recall that this extension involves transforming a SNF query $w_1 \wedge \psi(w_2, \cdots, w_n)$ to a set of conjunctions, each including w_1 and potentially involving some negated terms (*e.g.,* $w_1 \wedge w_2 \wedge \neg w_3$). A SNF query can then be processed by performing each transformed conjunction and taking the union of the search results for these conjunctions.

Failing to Support Conjunctions with Negated Terms Effectively. The first challenge in extending HXT using this strategy is its ineffectiveness in supporting conjunctions with negated terms. For example, to evaluate a conjunctive query like $w_1 \wedge w_2 \wedge \neg w_3$, the client must check whether each document identifier $id \in \mathrm{DB}(w_1)$ satisfies $w_2 \wedge \neg w_3$. This requires querying whether the Bloom filter BF stores 1 in all γ positions corresponding to the tag of (w_2, id) and 0 in at least one position for the tag of (w_3, id). Since there are $2^\gamma - 1$ combinations where BF stores 0 in at least one of the γ positions, up to $2^\gamma - 1$ SHVE queries, each related to a different combination, could be issued in order to obtain the final result. With γ typically set to 20 [LPS+, PKV+], a single negated term in a conjunctive query could result in a cost over 10^6 times higher than that of a conjunction without any negated term, and this cost increases linearly with the number of negated terms.

Failing to Support (or Achieve KPRP-hiding for) Certain Types of SNF Queries. Due to HXT's ineffectiveness in handling conjunctions with negated terms, it cannot be efficiently extended to support any SNF query involving negated terms using the aforementioned strategy. Furthermore, even when HXT is extended to support SNF queries without negated terms, it may still fail to achieve KPRP-hiding in certain cases. For instance, if a SNF query involves a disjunction with a single keyword, *e.g.,* $w_1 \wedge (w_2 \vee w_3 \wedge w_4)$, HXT could expose KPRP. This is because SHVE queries in the conjunctions reveal whether a document identifier $id \in \mathrm{DB}(w_1)$ satisfies w_2 or $w_3 \wedge w_4$ to the server, exposing whether id belongs to $\mathrm{DB}(w_1) \cap \mathrm{DB}(w_2)$.

A.2 HDXT

The main feature of HDXT is its ability to securely support update operations, but we do not need to delve into this aspect here, as query expressiveness is solely related to its search strategy. HDXT utilizes a mapping table and a Symmetric Hidden Map Encryption (SHME) scheme to determine whether a document $id \in \mathrm{DB}(w_1)$ matches other $n-1$ keywords during a conjunctive search $w_1 \wedge \cdots \wedge w_n$. The mapping table (denoted as DB') associates every possible keyword-document pair [3] with a bit that indicates whether the pair exists in the database. The SHME scheme can encrypt a map $\mathbf{m_a}$ and support queries to determine whether the pairs in another map $\mathbf{m_p}$ are all included in $\mathbf{m_a}$. If a SHME query returns *false*, it does not reveal which pair in $\mathbf{m_p}$ is included or not included in $\mathbf{m_a}$. During the setup phase of HDXT, the client employs SHME to encrypt DB' and sends the resulting ciphertext to the server. To test whether a document $id \in \mathrm{DB}(w_1)$ contains the other $n-1$ keywords, HDXT enables the client to build a map I_{id} that associates (w_i, id) (for $2 \leq i \leq n$) with bit 1 and issues a SHME query to check if all pairs in I_{id} are included in DB'.

In [YZCR], Yuan *et al.* demonstrated that HDXT can be extended to support conjunctive queries with negated terms while ensuring KPRP-hiding, by modifying I_{id} to associate (w_i, id) $(2 \leq i \leq n)$ with bit 1 or 0 based on whether w_i is a non-negated term or negated term. However, since the result for each SHME query is revealed to the server during the search, HDXT still faces the final query expressiveness issue discussed earlier regarding HXT, *i.e.,* it fails to achieve KPRP-hiding for a SNF query if the SNF query involves a disjunction with a single keyword.

B Correctness and Security Definitions for T-set

Definition 10 (Correctness of T-set). *Let Π represent a T-set implementation. We say that Π satisfies correctness if for any security parameter λ and any PPT adversary $\mathcal{A}$, there exists a negligible function negl such that:*

$$\Pr[\mathrm{TSetCorr}^{\Pi}_{\mathcal{A}}(\lambda) = 1] \leqslant negl(\lambda)$$

where the game $\mathrm{TSetCorr}^{\Pi}_{\mathcal{A}}(\lambda)$ *is defined as follows:*
$\mathrm{TSetCorr}^{\Pi}_{\mathcal{A}}(\lambda)$*: $\mathcal{A}$ chooses* W *and* T*, and receives* TSet *outputted by* ***TSetSetup***$(1^\lambda, \mathrm{T})$*. Then it selects keywords $w \in$ W in an adaptive manner. For each selected w, $\mathcal{A}$ obtains the corresponding stag produced by* ***TSetGetTag***(K_T, w) *and t_w generated by* ***TSetRetrieve***$(stag, \mathrm{TSet})$*. The game outputs 1 if $t_w \neq \mathrm{T}[w]$ for any chosen keyword w.*

Definition 11 (Security of T-set). *Let Π be a T-set implementation. We say Π is $\mathcal{L}_T$−adaptively−secure if for any security parameter λ, any PPT adversary $\mathcal{A}$, there exist a simulator $\mathcal{S}$ and a negligible function negl such that:*

$$|\Pr[\mathrm{TSReal}^{\Pi}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathrm{TSIdeal}^{\Pi}_{\mathcal{A},\mathcal{S},\mathcal{L}_T}(\lambda) = 1]| \leqslant negl(\lambda)$$

where $\mathrm{TSReal}^{\Pi}_{\mathcal{A}}(\lambda)$ *and* $\mathrm{TSIdeal}^{\Pi}_{\mathcal{A},\mathcal{S},\mathcal{L}_\mathrm{T}}(\lambda)$ *are defined as:*

- $\mathrm{TSReal}^{\Pi}_{\mathcal{A}}(\lambda)$*: At first, $\mathcal{A}$ chooses* W *and* T*, and obtains* TSet *by calling* ***TSetSetup***$(1^\lambda, \mathrm{T})$*. Then it repeatedly selects w from* W *in an adaptive way. For each selected w, $\mathcal{A}$ receives stag outputted by* ***TSetGetTag***(K_T, w)*. $\mathcal{A}$ receives all the transcripts generated during the above operations and outputs a bit b.*
- $\mathrm{TSIdeal}^{\Pi}_{\mathcal{A},\mathcal{S},\mathcal{L}_\mathrm{T}}(\lambda)$*: $\mathcal{A}$ chooses* W *and* T*, and calls $\mathcal{S}(\mathcal{L}_T^{tup}(\mathrm{T}))$ to get* TSet*. After that, it adaptively chooses keywords w from* W*. For each w, it receives the output of $\mathcal{S}(\mathcal{L}_T^{tag}(\mathrm{T}, w), \mathrm{T}[w])$. $\mathcal{A}$ observes the transcripts of all operations and outputs a bit b.*

[3] A keyword-document pair is possible as long as the keyword belongs to set W and the document identifier belongs to set D.

C Proof and Experiments for RH-filter Construction

In this section, we present the correctness proof, security proof, and parameter section experimental results for the RH-filter construction introduced in Section 3.

C.1 Proof of Theorem 6

Proof. Given $K_F \leftarrow \text{RHFSetup}(1^\lambda)$ and $\text{ES} \leftarrow \text{RHFEncrypt}(K_F, \Delta)$, during a query on an element δ, if $\delta \in \Delta$, then $\text{RHFTest}(K_F, \delta, \text{ES}[H(etok)])$, where $etok \leftarrow \text{RHFGetTag}(K_F, \delta)$, must return 1. This is due to the deterministic nature of both F and H, which ensures that $tag_2 = F(k_{f2}, \delta)$ must correspond to one of the elements in the set $\text{ES}[H(etok)]$. Consequently, the game $\text{RHFC}\textsc{orr}^{\pounds}_{\mathcal{A}}(\lambda)$ can only output 1 when an adversary $\mathcal{A}$ selects a key δ that does not exist in Δ, and the result of $\text{RHFTest}(K_F, \delta, \text{ES}[H(etok)]$ is not $\perp$. In other words, the probability of obtaining an output of 1 in $\text{RHFC}\textsc{orr}^{\pounds}_{\mathcal{A}}(\lambda)$ is indeed equal to the probability of a false positive event occurring in $\text{RHFC}\textsc{orr}^{\pounds}_{\mathcal{A}}(\lambda)$.

To illustrate the negligible probability of false positives, we introduce a game, denoted as $G^{\pounds}_c$. Additionally, we assume that an adversary never repeats a query, which does not affect the adversary's advantage since the algorithms involved in queries are all deterministic.

Game $G^{\pounds}_c$: In $G^{\pounds}_c$, the only difference from $\text{RHFC}\textsc{orr}^{\pounds}_{\mathcal{A}}(\lambda)$ lies in the creation of a table $F2$ and the replacement of every call to $F(k_{f2}, \delta)$ with the following procedure: if δ is a new input to $F(k_{f2}, \cdot)$, selecting an output tag_2 uniformly at random from $\{0,1\}^\lambda$ and assigning it to $F2[\delta]$; otherwise, outputting $F2[\delta]$. Consequently, there exists a PPT adversary $\mathcal{B}$ for F such that

$$\Pr[\text{RHFC}\textsc{orr}^{\pounds}_{\mathcal{A}}(\lambda) = 1] - \Pr[G^{\pounds}_c = 1] \leqslant Adv^{\text{PRF}}_{F,\mathcal{B}}$$

where $Adv^{\text{PRF}}_{F,\mathcal{B}}$ refers to the PRF advantage of adversary $\mathcal{B}$ on F.

Game $G^{\pounds}_c$ outputs 1 only when a false positive event occurs. In a false positive event, for an element δ that does not exist in Δ, $\text{RHFTest}(K_F, \delta, \text{ES}[H(etok)])$, where $etok \leftarrow \text{RHFGetTag}(K_F, \delta)$, selects a string tag_2 uniformly at random from $\{0,1\}^\lambda$, but tag_2 happens to be an element in $\text{ES}[H(etok)]$. As every element in $\text{ES}[H(etok)]$ was also selected from $\{0,1\}^\lambda$ uniformly at random, this event occurs with a probability of $|\text{ES}[H(etok)]|/2^\lambda$, which is upper-bounded by $|\Delta|/2^\lambda$.

Assuming that $\mathcal{A}$ can issue μ queries on keys that do not exist in Δ, we can deduce that

$$\Pr[G^{\pounds}_c = 1] = \Pr[\text{ a false positive event happens in } G^{\pounds}_c] \leq \frac{u|\Delta|}{2^\lambda}$$

Given that $\mathcal{A}$ is a PPT adversary, $\mu|\Delta|$ can be bounded by a real polynomial in λ. Consequently, the term $\frac{u|\Delta|}{2^\lambda}$ is negligible in λ.

In conclusion:

$$\Pr[\text{RHFC}\textsc{orr}^{\pounds}_{\mathcal{A}}(\lambda) = 1] \leqslant Adv^{\text{PRF}}_{F,\mathcal{B}} + \frac{u|\Delta|}{2^\lambda}$$

C.2 Proof of Theorem 7

Proof. In Figure 12, we introduce the simulation experiment $\text{RHFI}\textsc{deal}^{\pounds}_{\mathcal{A},\mathcal{S}}(\lambda)$. To establish the indistinguishability of this simulation experiment from the real one, we progressively define three games: $G^{\pounds}_0$, $G^{\pounds}_1$, and $G^{\pounds}_2$ as follows.

Game $G^{\pounds}_0$: Game $G^{\pounds}_0$ is exactly the real experiment $\text{RHFR}\textsc{eal}^{\pounds}_{\mathcal{A}}(\lambda)$.

Game $G^{\pounds}_1$: In $G^{\pounds}_1$, we create an empty table F1 and replace each call to $F(k_{f1}, \delta)$ with the following procedure: if δ is a new input to $F(k_{f1}, \cdot)$, choosing the output tag_1 uniformly at random from $\{0,1\}^\lambda$ and setting $\text{F1}[\delta]$ to tag_1, otherwise, outputting $\text{F1}[\delta]$. Additionally, since the inputs to $F(k_{f2}, \cdot)$ never repeat in $G^{\pounds}_0$, we substitute each call to $F(k_{f2}, \delta)$ with the selection of an output uniformly at random from $\{0,1\}^\lambda$. To distinguish $G^{\pounds}_1$ from $G^{\pounds}_0$, an adversary must compromise the security of $F(k_{f1}, \cdot)$ or $F(k_{f2}, \cdot)$. That is, there exists a PPT adversary $\mathcal{B}$ for F such that

$$|\Pr[G^{\pounds}_0(\lambda) = 1] - \Pr[G^{\pounds}_1(\lambda) = 1]| \leqslant 2 \cdot Adv^{\text{PRF}}_{F,\mathcal{B}}$$

Game $G^{\pounds}_2$: To create $G^{\pounds}_2$, we replace every call to H in RHFEncrypt with the selection of an output from $[1, \zeta]$ uniformly at random. Given that H is a random oracle and the set Δ does not contain duplicate elements, an adversary cannot distinguish $G^{\pounds}_2$ from $G^{\pounds}_1$.

1) $\mathcal{S}(\perp)$ creates an empty table F1.
2) $\mathcal{A}$ chooses elements in an adaptive way. For the i-th element δ, $\mathcal{S}(tp(\delta))$ runs the following three procedures: ① If $tp(\delta)$ is not empty, it takes the minimal value j in $tp(\delta)$ and sets $etok \leftarrow \mathrm{F1}[j]$; ② If $tp(\delta)$ is empty, $\mathcal{S}$ selects $etok \overset{\$}{\leftarrow} \{0,1\}^\lambda$ and sets $\mathrm{F1}[i] \leftarrow etok$; ③ $\mathcal{S}$ delivers $etok$ to $\mathcal{A}$. In the end, $\mathcal{S}$ records the total number of queries that happened in this phase, which is denoted as num.
3) $\mathcal{A}$ selects a set Δ. $\mathcal{S}(|\Delta|)$ runs the following four procedures: ① As in the real experiment, $\mathcal{S}$ chooses a constant φ, sets $\zeta \leftarrow \varphi|\Delta|$, and creates an array ES containing ζ empty sets; ② For $1 \le i \le |\Delta|$, $\mathcal{S}$ computes $tag_2 \overset{\$}{\leftarrow} \{0,1\}^\lambda$, $pos \overset{\$}{\leftarrow} [1,\zeta]$, and sets $\mathrm{ES}[pos] \leftarrow \mathrm{ES}[pos] \cup \{tag_2\}$; ③ $\mathcal{S}$ pads ES following Line 14-20 in RHFEncrypt provided in Figure 3; ④ $\mathcal{S}$ gives ES to $\mathcal{A}$.
4) $\mathcal{A}$ runs as in step 2), except that $\mathcal{S}$ of this phase sets $\mathrm{F1}[\ num + i] \leftarrow etok$ in procedure ②, instead of updating $\mathrm{F1}[i]$.
5) $\mathcal{A}$ outputs a bit b.

Fig. 12. $\mathrm{RHFI\small{DEAL}}^{\pounds}_{\mathcal{A},\mathcal{S}}(\lambda)$

$$\Pr[G_1^{\pounds}(\lambda) = 1] = \Pr[G_2^{\pounds}(\lambda) = 1]$$

$\mathrm{RHFI\small{DEAL}}^{\pounds}_{\mathcal{A},\mathcal{S}}(\lambda)$: The only distinction between $G_2^{\pounds}$ and $\mathrm{RHFI\small{DEAL}}^{\pounds}_{\mathcal{A},\mathcal{S}}(\lambda)$ is that the latter experiment substitutes each key δ in $F1$ in the former experiment with the sequence number of the earliest query regarding δ. The observations made by an adversary in both experiments are indistinguishable. Consequently, we can draw the following conclusion:

$$|\Pr[\mathrm{RHFR\small{EAL}}^{\pounds}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathrm{RHFI\small{DEAL}}^{\pounds}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leqslant 2 \cdot Adv^{\mathrm{PRF}}_{F,\mathcal{B}}$$

‘

C.3 Parameter Selection for RH-filter

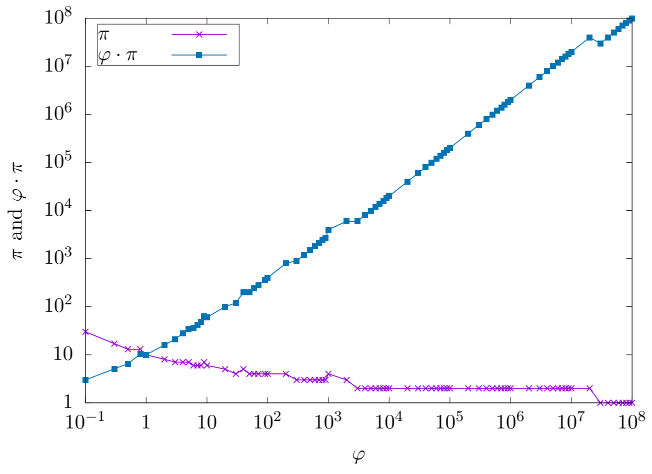


Fig. 13. Trends of π and $\varphi \cdot \pi$ as φ Increases

The performance analysis provided in Section 3.2 demonstrates that, given a set Δ, the communication efficiency for a membership query improves as π decreases, while the encryption time and the size of the encrypted structure ES depend on the product $\varphi\pi$. An increase in φ implies that fewer elements will be mapped to the same hash, leading to a decrease of π. However, the impact of increasing φ on $\varphi\pi$ is not immediately intuitive.

In this section, we evaluate the trend of π and $\varphi\pi$ by varying the parameter φ from 10^{-1} to 10^8, using all the keyword-document pairs from the dataset in Section 8 as input. The experiment results, shown in Figure 13, are plotted on a logarithmic scale for both axes. The figure highlights three key observations. First, as φ increases, π decreases and $\varphi \cdot \pi$ increases, indicating a trade-off between storage and query efficiency for the RH-filter. Second, π remains small even when φ is very small. For instance, when φ is set to 10^{-1}, π is only 30. Third, as φ increases, the rate at which π decreases slows down significantly while the growth trend of $\varphi \cdot \pi$ is nearly linear. For instance, when φ increases from 10^{-1} to 1, π decreases by 67%. However, there is no change in π when φ ranges from 10^4 to 10^7. This demonstrates that setting φ to a small value (such as 1 or 2) is likely to achieve an exceptional trade-off.

D Proofs for HBS

This section presents the correctness and security proofs for HBS.

D.1 Proof of Theorem 8

Proof. We establish the validity of Theorem 8 by constructing a hybrid of games denoted as $G_{Cor,0}$ through $G_{Cor,5}$.

Game $G_{Cor,0}$: Game $G_{Cor,0}$ is exactly the experiment $\mathrm{SSECorrect}_{\mathcal{A}}^{\mathsf{HBS}}(\lambda)$.

Game $G_{Cor,1}$: Game $G_{Cor,1}$ introduces a modification to the search protocol. As depicted in Figure 14, in $G_{Cor,1}$, we trigger an output of 1 if the resulting tuple list t, obtained through the execution of $\mathrm{TSetRetrieve}(stag, \mathrm{TSet})$, does not match $\mathrm{T}[w_1]$. This modification leads to the inequality:

$$\Pr[G_{Cor,0} = 1] \leq \Pr[G_{Cor,1} = 1] \tag{1}$$

Game $G_{Cor,2}$: As illustrated in Figure 14, $G_{Cor,2}$ introduces a set of pseudocode lines enclosed within single-layer boxes. Building upon $G_{Cor,1}$, $G_{Cor,2}$ also triggers an output of 1 of the result bit, b, from $\mathrm{RHFTest}(K_F, xtag, res)$ does not match $In(xtag, \mathrm{XSet})$. Consequently, we establish the following relationship:

$$\Pr[G_{Cor,1} = 1] \leq \Pr[G_{Cor,2} = 1] \tag{2}$$

Game $G_{Cor,3}$: In Figure 14, we introduce the pseudocodes enclosed by double-layer boxes to create $G_{Cor,3}$. In $G_{Cor,3}$, we specify that the tuple list t, acquired through the search involving w_1 within TSet, must precisely match $\mathrm{T}[w_1]$. Additionally, when verifying the membership of $xtag$, it is mandated that the output b must align with $In(xtag, \mathrm{XSet})$. Consequently, for any adversary $\mathcal{A}$, we can identify two adversaries, $\mathcal{B}_t$ and $\mathcal{B}_r$, such that:

$$\Pr[G_{Cor,2} = 1] - \Pr[G_{Cor,3} = 1] \leq Adv^{corr}_{tset,\mathcal{B}_t}(\lambda) + Adv^{corr}_{\pounds,\mathcal{B}_r}(\lambda) \tag{3}$$

where $tset$ represents a T-set implementation, $\pounds$ denotes our RH-filter construction as provided in Section 3, and $Adv^{corr}_{tset,\mathcal{B}_t}(\lambda)$ and $Adv^{corr}_{\pounds,\mathcal{B}_r}(\lambda)$ represent the advantages that adversaries $\mathcal{B}_t$ and $\mathcal{B}_r$ gain in compromising the correctness of $tset$ of $\pounds$, respectively.

$G_{Cor,3}$ produces an output of 1 only under one condition: the existence of at least one keyword-document pair (w, id) that is not present in the database, yet its $xtag$ belongs to XSet. Below, we construct two games, $G_{Cor,4}$ and $G_{Cor,5}$, to illustrate that the probability of this particular scenario occurring is negligible.

Game $G_{Cor,4}$: As shown in Figure 15 and Figure 16, to construct $G_{Cor,4}$, we first create three mapping tables, namely $\mathcal{X}$, $\mathcal{S}$, and $\mathcal{I}$, corresponding to the functions $F_p(k_x, \cdot)$, $F(k_s, \cdot)$, and $F_p(k_i, \cdot)$, respectively. The purpose is to replace every call to $F_p(k_i, \cdot)$ (*resp.* $F(k_s, \cdot)$ and $F_p(k_x, \cdot)$) with a new procedure. If the input i for $F_p(k_i, \cdot)$ (*resp.* $F(k_s, \cdot)$ and $F_p(k_x, \cdot)$) is new, $G_{Cor,4}$ randomly selects an output o from $\mathbb{Z}_p^*$ (*resp.* $\{0,1\}^\lambda$ and stores o in the respective mapping table $\mathcal{I}[i]$ (*resp.* $\mathcal{S}[i]$ and $\mathcal{X}[i]$). If i has been seen before, it simply retrieves the previously stored output.

Additionally, two mapping tables, $\mathcal{KZ}$ and $\mathcal{Z}$, are created for functions $F(strap, \cdot)$ and $F_p(k_z, \cdot)$, respectively. For each new input of the form $(strap, i)$ (*resp.* (k_z, i)), the output is randomly and uniformly selected from the set $\{0,1\}^\lambda$ (*resp.* $\mathbb{Z}_p^*$) and stored in $\mathcal{KZ}[strap, i]$ (*resp.* $\mathcal{Z}[k_z, i]$). Note that there can be $|\mathrm{W}|$ different $strap$ and k_z values used as secret keys for $F(strap, \cdot)$ and $F(k_z, \cdot)$, respectively.

Then, there exists a PPT adversary $\mathcal{B}_f$ such that:

$$\Pr[G_{Cor,3} = 1] - \Pr[G_{Cor,4} = 1] \leq (|\mathrm{W}| + 1) \cdot Adv^{\mathrm{PRF}}_{F,\mathcal{B}_f} + (|\mathrm{W}| + 2) \cdot Adv^{\mathrm{PRF}}_{F_p,\mathcal{B}_f} \tag{4}$$

```
Search$(K, w_1 \wedge con(w_2, \cdot, w_n); \mathrm{EDB})$ in $G_{Cor,1}$, $\boxed{G_{Cor,2}}$, and $\boxed{\boxed{G_{Cor,3}}}$:
    Client:
 1: $(k_s, k_x, k_i, K_T, K_F) \leftarrow K$
 2: $stag \leftarrow$TSetGetTag$(K_T, w_1)$
 3: $(k_{f1}, k_{f2}) \leftarrow K_F$
 4: Send $stag$ and $k_{f1}$ to the server
 5: $strap \leftarrow F(k_s, w_1),\ k_z \leftarrow F(strap, 1)$
 6: for $c = 1, 2 \cdots$ and until server sends $stop$ do
 7:     $\mathrm{xtoken}_c \leftarrow$ empty list
 8:     for $i = 2$ to $n$ do
 9:         $xtok_{c,i} \leftarrow g^{F_p(k_z,c)\cdot F_p(k_x,w_i)}$
10:         $\mathrm{xtoken}_c \leftarrow \mathrm{xtoken}_c \cup \{xtok_{c,i}\}$
11:     end for
12:     Send $\mathrm{xtoken}_c$ to the server
13: end for

    Server:
14: Responds $\leftarrow$ empty list
15: t $\leftarrow$TSetRetrieve$(stag, \mathrm{TSet})$
16: if $\mathrm{t} \neq \mathrm{T}[w_1]$ then
17:     Game $G_{Cor,1}$ $\boxed{G_{Cor,2}}$ outputs 1
18: end if
19: $\boxed{\boxed{\mathrm{t} \leftarrow \mathrm{T}[w_1]}}$
20: for $c = 1, \cdots, |\mathrm{t}|$ do
21:     $(-, y) \leftarrow \mathrm{t}[c]$
22:     $\mathrm{respond}_c \leftarrow$ empty list
23:     for $i = 2$ to $|\mathrm{xtoken}_c| + 1$ do
24:         $xtok_{c,i} \leftarrow \mathrm{xtoken}_c[i-1]$
25:         $xtag \leftarrow (xtok_{c,i})^y$
26:         $etok \leftarrow F(k_{f1}, xtag)$
                                        ▷ Equivalent to executing RHFGetTok$(K_F, xtag)$ provided in Figure 3
27:         $res \leftarrow$ RHFRespond$(etok, \mathrm{ES})$
28:         $\mathrm{respond}_c \leftarrow \mathrm{respond}_c \cup \{(xtag, res)\}$
29:     end for
30:     Responds $\leftarrow$ Responds $\cup \{respond_c\}$
31: end for
32: When last tuple in t is reached, send $stop$ to the client.
33: Send Responds to the client

    Client:
34: ck $\leftarrow$ empty set, $k_e \leftarrow F(strap, 2)$
35: for $c = 1$ to |Responds| do
36:     $match_c \leftarrow true$
37:     $\mathrm{respond}_c \leftarrow \mathrm{Responds}[c]$
38:     for $i = 2$ to $n$ do
39:         $(xtag, res) \leftarrow \mathrm{respond}_c[i-1]$
40:         $b \leftarrow$ RHFTest$(K_F, xtag, res)$
41:         if $\boxed{b \neq In(xtag, \mathrm{XSet})}$ then
42:             $\boxed{\text{Game } G_{Cor,2} \text{ outputs } 1}$
43:         end if
44:         $\boxed{\boxed{b \leftarrow In(xtag, \mathrm{XSet})}}$
45:         if ($b = 1$ and $w_i$ is a negated term) or ($r = 0$ and $w_i$ is a non-negated term) then
46:             $match_c \leftarrow false$
47:             Break the loop for $i$
48:         end if
49:     end for
50:     if $match_c = true$ then
51:         $k_d \leftarrow F(k_e, c)$, ck $\leftarrow$ ck $\cup \{(c, k_d)\}$
52:     end if
53: end for
54: Send ck to the server

    Server:
55: R $\leftarrow$ empty set
56: for each $(c, k_d) \in$ ck do
57:     $(e, -) \leftarrow \mathrm{t}[c]$
58:     $id \leftarrow k_d \oplus e$
59:     R $\leftarrow$ R $\cup \{id\}$
60: end for
61: Send R to the client
```

Fig. 14. Search Protocol in Games $G_{Cor,1}$, $\boxed{G_{Cor,2}}$, and $\boxed{\boxed{G_{Cor,3}}}$

```
**Setup**$(1^\lambda, \mathrm{DB}; \perp)$ in $G_{Cor,4}$ and $\boxed{G_{Cor,5}}$:
    *Client*
 1: Select key $k_s$ for PRF $F$
 2: Select keys $k_x$ and $k_i$ for PRF $F_p$
 3: $\mathrm{T} \leftarrow$ empty array indexed by keywords from $\mathrm{W}$
 4: $\mathrm{XSet} \leftarrow$ empty set
 5: **for** each $w \in \mathrm{W}$ **do**
 6:     $x \xleftarrow{\$} \mathbb{Z}_p^*,\ \mathcal{X}[w] \leftarrow x$
 7:     $strap \xleftarrow{\$} \{0,1\}^\lambda,\ \mathcal{S}[w] \leftarrow strap$
 8:     $k_z \xleftarrow{\$} \{0,1\}^\lambda,\ \mathcal{KZ}[strap, 1] \leftarrow k_z$
 9:     $k_e \xleftarrow{\$} \{0,1\}^\lambda,\ \mathcal{KZ}[strap, 2] \leftarrow k_e$
10:     $\mathrm{t} \leftarrow$ empty list, $c \leftarrow 0$
11:     Randomly permute the entries of $\mathrm{DB}(w)$
12:     **for** each $id$ in $\mathrm{DB}(w)$ **do**
13:         $c \leftarrow c + 1$
14:         **if** $\mathcal{I}[id]$ exists **then**
15:             $xind \leftarrow \mathcal{I}[id]$
16:         **else**
17:             $xind \xleftarrow{\$} \mathbb{Z}_p^*,\ \mathcal{I}[id] \leftarrow xind$
18:         **end if**
19:         $z \xleftarrow{\$} \mathbb{Z}_p^*,\ \mathcal{Z}[k_z, c] \leftarrow z$
20:         $y \leftarrow xind \cdot z^{-1}$
21:         $k_d \leftarrow F(k_e, c),\ e \leftarrow k_d \oplus id$
22:         $\mathrm{t} \leftarrow \mathrm{t} \cup \{(e, y)\}$
23:             $xtag \leftarrow g^{x \cdot xind}$, $\boxed{xtag \xleftarrow{\$} \mathbb{G}_p}$
24:             $\boxed{\mathcal{XG}[w, id] \leftarrow xtag,\ \mathcal{CD}[w, c] \leftarrow id}$
25:             $\mathrm{XSet} \leftarrow \mathrm{XSet} \cup \{xtag\}$
26:     **end for**
27:     $\mathrm{T}[w] \leftarrow \mathrm{t}$
28:     **for** $\boxed{\text{each } id \in \mathrm{D} \setminus \mathrm{DB}(w)}$ **do**
29:         $\boxed{\mathcal{XG}[w, id] \xleftarrow{\$} \mathbb{G}_p}$
30:     **end for**
31:     **for** $\boxed{\text{each } w' \in \mathrm{W} \setminus \{w\}}$ **do**
32:         **for** $\boxed{c = (|\mathrm{DB}(w)| + 1) \text{ to } |\mathrm{D}|}$ **do**
33:             $\boxed{\mathcal{XT}[w, w', c] \xleftarrow{\$} \mathbb{G}_p}$
34:         **end for**
35:     **end for**
36: **end for**
37: $(\mathrm{TSet}, K_T) \leftarrow \mathrm{TSetSetup}(\mathrm{T})$
38: $K_F \leftarrow \mathrm{RHFSetup}(\lambda)$
39: $\mathrm{ES} \leftarrow \mathrm{RHFEncrypt}(K_F, \mathrm{XSet})$
40: Send TSet and ES to the server
41: **return** $K = (k_s, k_x, k_i, k_z, K_T, K_F)$

    *Server*
42: **return** $\mathrm{EDB} = (\mathrm{TSet}, \mathrm{ES})$
```

Fig. 15. Setup Protocol in Games $G_{Cor,4}$ and $\boxed{G_{Cor,5}}$

Game $G_{Cor,5}$: To construct the game $G_{Cor,5}$, we introduce the pseudocodes enclosed by single-layer boxes, as shown in Figure 15 and Figure 16. Specifically, during the setup protocol, $G_{Cor,5}$ introduces three tables: $\mathcal{XG}$, $\mathcal{CD}$, and $\mathcal{XT}$. $\mathcal{XG}$ stores the mapping between every keyword-document pair (w, id) and the $xtag$ of (w, id), regardless of whether (w, id) is in the database or not. Notably, instead of calculating the $xtag$ of (w, id) using an exponential operation, $G_{Cor,5}$ directly selects $xtag$ uniformly at random from $\mathbb{G}_p$. $\mathcal{CD}$ is utilized to record the c-th document identifier id in $\mathrm{DB}(w)$ by storing the correspondence between (w, c) and id. For a keyword w, any keyword $w' \neq w$, and $|\mathrm{DB}(w)| + 1 \leq c \leq |\mathrm{D}|$, $\mathcal{XT}[w, w', c]$ stores an element selected from $\mathbb{G}_p$ uniformly at random.

$G_{Cor,5}$ requires modifications in the search protocol to ensure consistency. Specifically, within a search query $w_1 \wedge con(w_2, \cdots, w_n)$, after obtaining $\mathrm{t} = \mathrm{T}[w_1]$, it must ensure that $xtok_{c,i}^y$ remains equal to $xtag$ of (w_i, id) for $1 \leq c \leq |\mathrm{DB}(w_1)|$ and $2 \leq i \leq n$, where y is taken from $\mathrm{t}[c]$ and id is $\mathcal{CD}[w_1, c]$. Also, if the client does not receive the stop signal promptly, for $c > |\mathrm{DB}(w_1)|$, $G_{Cor,5}$ must ensure that the value of $xtok_{c,i}$ is deterministic with respect to the *s-term* w_1, the *x-term* w_i, and the counter c. To achieve th consistency, when generating $xtok_{c,i}$, if $1 \leq c \leq |\mathrm{DB}(w_1)|$, we take id from $\mathcal{CD}[w_1, c]$, computes $y = \mathcal{I}[id] \cdot \mathcal{Z}[k_z, c]^{-1}$, and sets $xtok_{c,i} = \mathcal{XG}[w_i, id]^{y^{-1}}$. If $c > |\mathrm{DB}(w_1)|$, we set $xtok_{c,i}$ to be $\mathcal{XT}[w_1, w_i, c]$. Note that in this proof, we assume that the client will compute and transmit a maximum of $|\mathrm{D}|$ xtoken lists within a search query.

The distinction between $G_{Cor,4}$ and $G_{Cor,5}$ lies in how xtags and xtokens are obtained. In $G_{Cor,4}$, each $xtag$ and $xtok$ are computed through exponentiation ($xtag = g^{\mathcal{X}[w] \cdot \mathcal{I}[id]}$ and $xtok_{c,i} = g^{\mathcal{Z}[k_z, c] \cdot \mathcal{X}[w_i]}$), while in $G_{Cor,5}$, $xtag$ and $xtok_{c,i}$ are uniformly distributed in $\mathbb{G}_p$. As in [CJJ$^+$b], we reduce the advantage that an adversary can identify this difference to that of breaking the hardness of DDH assumption. This reduction relies on a standard lemma derived from the DDH assumption [CJJ$^+$b]. The lemma considers vector $\mathbf{a} \in (\mathbb{Z}_p^*)^\alpha$ (*i.e.,* $\mathbf{a}$ consists of α elements, where every element is a member of $\mathbb{Z}_p^*$), vector $\mathbf{b} \in (\mathbb{Z}_p^*)^\alpha$, vector $g^{\mathbf{a}} = (g^{\mathbf{a}[1]}, \cdots, g^{\mathbf{a}[\alpha]}) \in (\mathbb{G}_p)^\alpha$,

```
**Search**$(K, w_1 \wedge con(w_2, \cdot, w_n); \mathrm{EDB})$ in $G_{Cor,4}$ and $\boxed{G_{Cor,5}}$:
    Client:
 1: $(k_s, k_x, k_i, K_T, K_F) \leftarrow K$
 2: $stag \leftarrow \mathrm{TSetGetTag}(K_T, w_1)$
 3: $(k_{f1}, k_{f2}) \leftarrow K_F$
 4: Send $stag$ and $k_{f1}$ to the server
 5: $strap \leftarrow \mathcal{S}[w_1]$,  $k_z \leftarrow \mathcal{KZ}[strap, 1]$
 6: for $c = 1, 2 \cdots$ and until server sends $stop$ do
 7:     $\mathrm{xtoken}_c \leftarrow$ empty list
 8:     for $i = 2$ to $n$ do
 9:         $xtok_{c,i} \leftarrow g^{\mathcal{Z}[k_z,c] \cdot \mathcal{X}[w_i]}$
10:         if $\boxed{c \leq |\mathrm{DB}(w_1)|}$ then
11:             $\boxed{id \leftarrow \mathcal{CD}[w_1, c]}$
12:             $\boxed{y \leftarrow \mathcal{I}[id] \cdot \mathcal{Z}[k_z, c]^{-1}}$
13:             $\boxed{xtok_{c,i} \leftarrow \mathcal{XG}[w_i, id]^{y^{-1}}}$
14:         else
15:             $\boxed{xtok_{c,i} \leftarrow \mathcal{XT}[w_1, w_i, c]}$
16:         end if
17:         $\mathrm{xtoken}_c \leftarrow \mathrm{xtoken}_c \cup \{xtok_{c,i}\}$
18:     end for
19:     Send $\mathrm{xtoken}_c$ to the server
20: end for

    Server:
21: $\mathrm{t} \leftarrow \mathrm{T}[w_1]$
22: for $c = 1, \cdots, |\mathrm{t}|$ do
23:     $(-, y) \leftarrow \mathrm{t}[c]$
24:     $\mathrm{respond}_c \leftarrow$ empty list
25:     for $i = 2$ to $|\mathrm{xtoken}_c| + 1$ do
26:         $xtok_{c,i} \leftarrow \mathrm{xtoken}_c[i-1]$
27:         $xtag \leftarrow (xtok_{c,i})^y$
28:         $etok \leftarrow F(k_{f1}, xtag)$
                  ▷ Equivalent to executing RHFGetTok$(K_F, xtag)$ provided in Figure 3
29:         $res \leftarrow \mathrm{RHFRespond}(etok, \mathrm{ES})$
30:         $\mathrm{respond}_c \leftarrow \mathrm{respond}_c \cup \{(xtag, res)\}$
31:     end for
32:     $\mathrm{Responds} \leftarrow \mathrm{Responds} \cup \{respond_c\}$
33: end for
34: When last tuple in t is reached, send $stop$ to the client.
35: Send Responds to the client

    Client:
36: ck $\leftarrow$ empty set, $k_e \leftarrow \mathcal{KZ}[strap, 2]$
37: for $c = 1$ to $|\mathrm{Responds}|$ do
38:     $match_c \leftarrow true$
39:     $\mathrm{respond}_c \leftarrow \mathrm{Responds}[c]$
40:     for $i = 2$ to $n$ do
41:         $(xtag, res) \leftarrow \mathrm{respond}_c[i-1]$
42:         $b \leftarrow In(xtag, \mathrm{XSet})$
43:         if ($b = 1$ and $w_i$ is a negated term) or ($r = 0$ and $w_i$ is a non-negated term) then
44:             $match_c \leftarrow false$
45:             Break the loop for $i$
46:         end if
47:     end for
48:     if $match_c = true$ then
49:         $k_d \leftarrow F(k_e, c)$, ck $\leftarrow$ ck $\cup \{(c, k_d)\}$
50:     end if
51: end for
52: Send ck to the server

    Server:
53: R $\leftarrow$ empty set
54: for each $(c, k_d) \in$ ck do
55:     $(e, -) \leftarrow \mathrm{t}[c]$
56:     $id \leftarrow k_d \oplus e$
57:     R $\leftarrow$ R $\cup \{id\}$
58: end for
59: Send R to the client
```

Fig. 16. Search Protocol in Games $G_{Cor,4}$ and $\boxed{G_{Cor,5}}$

vector $g^{\mathbf{b}} = (g^{\mathbf{b}[1]}, \cdots, g^{\mathbf{b}[\beta]}) \in (\mathbb{G}_p)^{\beta}$, and matrix $g^{\mathbf{a}\cdot\mathbf{b}^{\mathrm{T}}} \in (\mathbb{G}_p)^{\alpha\times\beta}$ where the (i,j)-th entry is $g^{\mathbf{a}[i]\cdot\mathbf{b}[j]}$. The lemma can demonstrates that, for a vector $\mathbf{a} = (g^{\mathcal{X}[w]})_{w\in \mathrm{W}} \in (\mathbb{G}_p)^{|\mathrm{W}|}$ and a vector $\mathbf{b}$ that concatenates $(g^{\mathcal{I}[id]})_{id\in \mathrm{D}} \in (\mathbb{G}_p)^{|\mathrm{D}|}$ and $(g^{\mathcal{Z}[k_z,c]})_{w\in \mathrm{W}, c\in[1,\mathrm{D}]} \in (\mathbb{G}_p)^{|\mathrm{W}|\cdot|\mathrm{D}|}$, the matric $g^{a\cdot b^{\mathrm{T}}} \in (\mathbb{G}_p)^{|\mathrm{W}|\times(|\mathrm{D}|+|\mathrm{W}|\cdot|\mathrm{D}|)}$ is computationally indistinguishable from a matrix M uniform over $(\mathbb{G}_p)^{|\mathrm{W}|\times(|\mathrm{D}|+|\mathrm{W}|\cdot|\mathrm{D}|)}$, given that the DDH assumption holds.

Consequently, there exists a PPT adversary $\mathcal{B}_d$ such that:

$$\Pr[G_{Cor,4} = 1] - \Pr[G_{Cor,5} = 1] \leq Adv^{\mathrm{DDH}}_{\mathbb{G}_p,\mathcal{B}_d}(\lambda) \tag{5}$$

A database can involve a total of $|\mathrm{W}|\cdot|\mathrm{D}|$ keyword-document pairs, with N of these pairs existing within the database. In $G_{Cor,5}$, a *xtag* is assigned to each keyword-document pair. As each *xtag* is chosen uniformly and randomly from the set $\mathbb{G}_p$, the probability of a non-existing pair share the same *xtag* with an existing pair is less than $N\cdot(|\mathrm{W}|\cdot|\mathrm{D}| - N)/p$. Consequently, we can establish that:

$$\Pr[G_{Cor,5} = 1] \leq \frac{N\cdot(|\mathrm{W}|\cdot|\mathrm{D}| - N)}{p} \tag{6}$$

By following the inequalities from (1) to (6), we arrive at the conclusion:

$$\begin{aligned} \mathrm{SSE}\mathrm{Correct}^{\mathsf{HBS}}_{\mathcal{A}}(\lambda) \leq Adv^{corr}_{tset,\mathcal{B}_t}(\lambda) + Adv^{corr}_{\pounds,\mathcal{B}_r}(\lambda) + 2\cdot Adv^{\mathrm{PRF}}_{F,\mathcal{B}_f} + 3\cdot Adv^{\mathrm{PRF}}_{F_p,\mathcal{B}_f} + \\ Adv^{\mathrm{DDH}}_{\mathbb{G}_p,\mathcal{B}_d}(\lambda) + \frac{N\cdot(|\mathrm{W}|\cdot|\mathrm{D}| - N)}{p} \end{aligned} \tag{7}$$

The T-set can be instantiated using the construction presented in [CJJ+b], leading to $Adv^{corr}_{tset,\mathcal{B}_t}(\lambda)$ being negligible. As demonstrated in Section 3.1, $Adv^{corr}_{\pounds,\mathcal{B}_r}(\lambda)$, equivalent to $\Pr[\mathrm{RHFCorr}^{\pounds}_{\mathcal{A}}(\lambda) = 1]$, is also negligible. Therefore, we can confidently assert that the probability of compromising the correctness of HBS is negligible.

D.2 Proof of Theorem 9

Proof. We construct a hybrid of games, through which we gradually prove that HBS is adaptively secure with respect to the leakage function $\mathcal{L}_{hbs}$.

Game G_0: The sole distinction between G_0 and the real experiment $\mathrm{SSEReal}^{\mathsf{HBS}}_{\mathcal{A}}(\lambda)$ lies in the guaranteed correctness of search results in G_0. Figure 18 illustrates the alterations implemented by G_0 in the generation process of the set ck to ensure that the server always returns the search result R as $DB(q)$ for every search query q. Consequently, we can reduce the advantage of an adversary $\mathcal{A}$ in distinguishing between G_0 and $\mathrm{SSEReal}\mathcal{A}^{\mathsf{HBS}}(\lambda)$ to the advantage of an adversary in compromising the correctness of HBS. Then, there exists a PPT adversary, denoted as $\mathcal{B}_h$, targeting the correctness of HBS. such that:

$$|\Pr[\mathrm{SSEReal}^{\mathsf{HBS}}_{\mathcal{A}}(\lambda) = 1] - \Pr[G_0 = 1]| \leq Adv^{corr}_{\mathsf{HBS},\mathcal{B}_h}(\lambda) \tag{8}$$

where $Adv^{corr}_{\mathsf{HBS},\mathcal{B}_h}(\lambda)$ represents the advantage that $\mathcal{B}_h$ compromises the correctness of HBS.

Game G_1: To create G_1, we incorporate the pseudocodes enclosed within single-layer boxes shown in Figure 17 and Figure 18. Building upon G_0, G_1 integrates the modifications introduced by game $G_{Cor,4}$ into game $G_{Cor,3}$. These modifications entail the introduction of tables $\mathcal{X}$, $\mathcal{S}$, $\mathcal{KZ}$, $\mathcal{I}$, and $\mathcal{Z}$ for PRFs. Note that both $G_{Cor,4}$ and $G_{Cor,3}$ were defined in Section 4.1. Furthermore, in G_1, we introduce a table $\mathcal{KD}$ for $F(k_e,\cdot)$. When presented with a new input in the format of (k_e, c), we randomly select an output value k_d from the set $\{0,1\}^{\lambda}$ and store it in $\mathcal{KD}[k_e, c]$. Subsequently, $\mathcal{KD}[k_e, c]$ remains the output for the input (k_e, c). Then, there exists a PPT adversary $\mathcal{B}_f$ such that:

$$|\Pr[G_0 = 1] - \Pr[G_1 = 1]| \leq (2|\mathrm{W}| + 1)\cdot Adv^{\mathrm{PRF}}_{F,\mathcal{B}_f} + (|\mathrm{W}| + 2)\cdot Adv^{\mathrm{PRF}}_{F_p,\mathcal{B}_f} \tag{9}$$

Game G_2: The modifications introduced by game G_2 to game G_1 are identical to those made by game $G_{Cor,5}$ to game $G_{Cor,4}$. These changes are enclosed using double-layer boxes in Figure 17 and Figure 18. Specifically, within the setup protocol of G_2, an *xtag* corresponding to a keyword-document pair is randomly and uniformly selected from the group $\mathbb{G}_p$. To facilitate the programming of xtokens during the search protocol, tables $\mathcal{XG}$, $\mathcal{CD}$, and $\mathcal{XT}$ are introduced. In G_2's search protocol,

```
Setup$(1^\lambda, \mathrm{DB}; \perp)$ in $G_0$, $\boxed{G_1}$, and $\boxed{\boxed{G_2}}$:
  Client
  Select key $k_s$ for PRF $F$
  Select keys $k_x$ and $k_i$ for PRF $F_p$
  $\mathrm{T} \leftarrow$ empty array indexed by keywords from W
  XSet $\leftarrow$ empty set
  for each $w \in \mathrm{W}$ do
      $k_e \leftarrow F(strap, 2)$
      $strap \leftarrow F(k_s, w)$
      $(k_z, k_e) \leftarrow (F(strap, 1), F(strap, 2))$
      $\boxed{x \xleftarrow{\$} \mathbb{Z}_p^*,\ \mathcal{X}[w] \leftarrow x}$
      $\boxed{strap \xleftarrow{\$} \{0,1\}^\lambda,\ \mathcal{S}[w] \leftarrow strap}$
      $\boxed{k_z \xleftarrow{\$} \{0,1\}^\lambda,\ \mathcal{KZ}[strap, 1] \leftarrow k_z}$
      $\boxed{k_e \xleftarrow{\$} \{0,1\}^\lambda,\ \mathcal{KZ}[strap, 2] \leftarrow k_e}$
      $\mathrm{t} \leftarrow$ empty list, $c \leftarrow 0$
      Randomly permute the entries of $\mathrm{DB}(w)$
      for each $id$ in $\mathrm{DB}(w)$ do
          $c \leftarrow c + 1$
          $xind \leftarrow F_p(k_i, id)$, $z \leftarrow F_p(k_z, c)$
          if $\boxed{\mathcal{I}[id] \text{ exists}}$ then
              $\boxed{xind \leftarrow \mathcal{I}[id]}$
          else
              $\boxed{xind \xleftarrow{\$} \mathbb{Z}_p^*,\ \mathcal{I}[id] \leftarrow xind}$
          end if
          $\boxed{z \xleftarrow{\$} \mathbb{Z}_p^*,\ \mathcal{Z}[k_z, c] \leftarrow z}$
          $y \leftarrow xind \cdot z^{-1}$
          $k_d \leftarrow F(k_e, c)$
          $\boxed{k_d \xleftarrow{\$} \{0,1\}^\lambda,\ \mathcal{KD}[k_e, c] \leftarrow k_d}$
          $e \leftarrow k_d \oplus id$
          $\mathrm{t} \leftarrow \mathrm{t} \cup \{(e, y)\}$
          $xtag \leftarrow g^{F_p(k_x, w) \cdot xind}$
          $\boxed{xtag \leftarrow g^{x \cdot xind}}$, $\boxed{\boxed{xtag \xleftarrow{\$} \mathbb{G}_p}}$
          $\boxed{\boxed{\mathcal{XG}[w, id] \leftarrow xtag,\ \mathcal{CD}[w, c] \leftarrow id}}$
          $\mathrm{XSet} \leftarrow \mathrm{XSet} \cup \{xtag\}$
      end for
      $\mathrm{T}[w] \leftarrow \mathrm{t}$
      for $\boxed{\boxed{\text{each } id \in \mathrm{D} \setminus \mathrm{DB}(w)}}$ do
          $\boxed{\boxed{\mathcal{XG}[w, id] \xleftarrow{\$} \mathbb{G}_p}}$
      end for
      for $\boxed{\boxed{\text{each } w' \in \mathrm{W} \setminus \{w\}}}$ do
          for $\boxed{\boxed{c = (|\mathrm{DB}(w)| + 1) \text{ to } |\mathrm{D}|}}$ do
              $\boxed{\boxed{\mathcal{XT}[w, w', c] \xleftarrow{\$} \mathbb{G}_p}}$
          end for
      end for
  end for
  $(\mathrm{TSet}, K_T) \leftarrow \mathrm{TSetSetup(T)}$
  $K_F \leftarrow \mathrm{RHFSetup}(\lambda)$
  $\mathrm{ES} \leftarrow \mathrm{RHFEncrypt}(K_F, \mathrm{XSet})$
  Send TSet and ES to the server
  return $K = (k_s, k_x, k_i, k_z, K_T, K_F)$

  Server
  return $\mathrm{EDB} = (\mathrm{TSet}, \mathrm{ES})$
```

Fig. 17. Setup Protocol in Games G_0, $\boxed{G_1}$, and $\boxed{\boxed{G_2}}$

if $c \leq |\mathrm{DB}(q[1])|$, the computation of $xtok_{c,i}$ involves the respective $xtag$ and the element y from the c-th tuple of $\mathrm{T}[q[1]]$. This ensures that the server computes the correct $xtags$. If $c > |\mathrm{DB}(q[1])|$, $xtok_{c,i}$ is set to $\mathcal{XT}[q[1], q[i], c]$. Drawing insights from the analysis of $G_{Cor,5}$ in Section 4.1, we can confidently assert the existence of a PPT adversary $\mathcal{B}_d$ such that:

$$|\Pr[G_1 = 1] - \Pr[G_2 = 1]| \leq Adv^{\mathrm{DDH}}_{\mathbb{G}_p, \mathcal{B}_d}(\lambda) \tag{10}$$

Game G_3: As depicted in Figure 19 and Figure 20, we construct G_3 by removing redundant pseudocode lines and selecting every element y from $\mathbb{Z}_p^*$ uniformly at random and computes. This change does not affect the distribution of transcripts. Consequently, we can deduce the following result:

$$\Pr[G_2 = 1] = \Pr[G_3 = 1] \tag{11}$$

Game G_4: In this game, as demonstrated in Figure 19 and Figure 20, we employ the simulator $\mathcal{S}_T$ designed for the T-set instantiation, as presented in [CJJ+b]. The simulator $\mathcal{S}_T$ is used to generate TSet and $stag$ in the setup and search protocols, respectively. Let $tset$ represent the TSet implementation. There exists a PPT adversary $\mathcal{B}_t$ for which:

```
**Search**$(K, q; \mathrm{EDB})$ in $G_0$, $\boxed{G_1}$ and $\boxed{\boxed{G_2}}$:
    Client:
1:  $(k_s, k_x, k_i, K_T, K_F) \leftarrow K$
2:  $stag \leftarrow$TSetGetTag$(K_T, q[1])$
3:  $(k_{f1}, k_{f2}) \leftarrow K_F$
4:  Send $stag$ and $k_{f1}$ to the server
5:  $strap \leftarrow F(k_s, q[1]),\ k_z \leftarrow F(strap, 1)$
6:  $\boxed{strap \leftarrow \mathcal{S}[q[1]],\ k_z \leftarrow \mathcal{KZ}[strap, 1]}$
7:  for $c = 1, 2 \cdots$ and until server sends $stop$ do
8:      xtoken$_c \leftarrow$ empty list
9:      for $i = 2$ to $n$ do
10:         $xtok_{c,i} \leftarrow g^{F_p(k_z, c) \cdot F_p(k_x, q[i])}$
11:         $\boxed{xtok_{c,i} \leftarrow g^{\mathcal{Z}[k_z, c] \cdot \mathcal{X}[q[i]]}}$
12:         if $\boxed{\boxed{c \leq |\mathrm{DB}(q[1])|}}$ then
13:             $\boxed{\boxed{id \leftarrow \mathcal{CD}[q[1], c]}}$
14:             $\boxed{\boxed{y \leftarrow \mathcal{I}[id] \cdot \mathcal{Z}[k_z, c]^{-1}}}$
15:             $\boxed{\boxed{xtok_{c,i} \leftarrow \mathcal{XG}[q[i], id]^{y^{-1}}}}$
16:         else
17:             $\boxed{\boxed{xtok_{c,i} \leftarrow \mathcal{XT}[q[1], q[i], c]}}$
18:         end if
19:         xtoken$_c \leftarrow$ xtoken$_c \cup \{xtok_{c,i}\}$
20:     end for
21:     Send xtoken$_c$ to the server
22: end for

    Server:
23: Run Line 15-29 in Figure 5
24: Send Responds to the client

    Client:
25: ck $\leftarrow$ empty set, $k_e \leftarrow \mathcal{KZ}[strap, 2]$
26: for each $id \in \mathrm{DB}(q)$ do
27:     $c \leftarrow$ the position of $id$ within $\mathrm{DB}(w_1)$
28:     $k_d \leftarrow F(k_e, c)$
29:     $\boxed{k_d \leftarrow \mathcal{KD}[k_e, c]}$
30:     ck $\leftarrow$ ck $\cup \{(c, k_d)\}$
31: end for
32: Send ck to the server

    Server:
33: Run Line 48-53 in Figure 5
34: Send R to the client
```

Fig. 18. Search Protocol in Games G_0, $\boxed{G_1}$, and $\boxed{\boxed{G_2}}$

$$|\Pr[G_3 = 1] - \Pr[G_4 = 1]| \leq Adv^{\text{T-set}}_{tset,\mathcal{B}_t}(\lambda) \tag{12}$$

where $Adv^{\text{T-set}}_{tset,\mathcal{B}_t}(\lambda)$ represents the advantage that an adversary $\mathcal{B}_t$ has in compromising the security of $tset$.

Game G_5: In this game, as indicated by the pseudocode enclosed within double-layer boxes in Figure 19, we utilize $\mathcal{S}_R(N)$ to generate ES, where $\mathcal{S}R$ serves as the simulator for the RH-filter implementation presented in Section 3. Consequently, there exists a PPT adversary $\mathcal{B}_r$ for which:

$$|\Pr[G_4 = 1] - \Pr[G_5 = 1]| \leq Adv^{\text{RHF}}_{\pounds,\mathcal{B}_r}(\lambda) \tag{13}$$

where $Adv^{\text{RHF}}_{\pounds,\mathcal{B}_r}(\lambda)$ represents the advantage that an adversary $\mathcal{B}_r$ has in compromising the security of our RH-filter construction $\pounds$.

The Simulator $\mathcal{S}$: Figure 21 and Figure **??** depict the construction for the simulator $\mathcal{S}$.

$\mathcal{S}(N)$ generates TSet and ES following the same procedure as in G_5.

In the ideal game, when processing a search query q, $\mathcal{S}$ initially selects the minimum timestamp (denoted as $\overline{w_1}$) from $\mathrm{EP}(q[1])$ to represent $q[1]$. If $\overline{w_1} = t$, it indicates that q is the first search query employing keyword $q[1]$ as the *s-term*. In such a scenario, $\mathcal{S}$ is tasked with generating the tuple list $\mathrm{T}[\overline{w_1}]$ by creating $|\mathrm{DB}(q[1])|$ tuples (e, y). Utilizing $\mathrm{T}[\overline{w_1}]$, $\mathcal{S}$ proceeds to execute $\mathcal{S}_T$ to generate the *stag*, following the procedure in G_5. In both G_5 and the ideal game, the tuples (e, y) within T are uniformly distributed in $(\{0,1\}^\lambda, \mathbb{Z}_p^*)$. Consequently, the distribution of *stags* produced by $\mathcal{S}_T(\mathrm{T}[\overline{w_1}])$ in the ideal game remains consistent with that in G_5.

In G_5 (*resp.* the ideal game), an adversary $\mathcal{A}$ also has access to $\mathrm{T}[q[1]]$ (*resp.* $\mathrm{T}[\overline{w_1}]$), as it is returned when the server executes TSetRetrieve$(stag, \mathrm{TSet})$. In G_5, each entry in $\mathrm{T}[q[1]]$ corresponds to a document identifier belonging to $\mathrm{DB}(q[1])$. As G_5 randomly permutes the entries in $\mathrm{DB}(w_1)$ before producing $\mathrm{T}[q[1]]$, the association between an entry in $\mathrm{T}[q[1]]$ and a document identifier in

```
**Setup**$(1^\lambda, \mathrm{DB}; \perp)$ in $G_3$, $\boxed{G_4}$, and $\boxed{\boxed{G_5}}$:
    *Client*
 1: T ← empty array indexed by keywords from W
 2: XSet ← empty set
 3: **for** each $w \in \mathrm{W}$ **do**
 4:     $strap \xleftarrow{\$} \{0,1\}^\lambda$, $\mathcal{S}[w] \leftarrow strap$
 5:     $k_e \xleftarrow{\$} \{0,1\}^\lambda$, $\mathcal{KZ}[strap, 2] \leftarrow k_e$
 6:     t ← empty list, $c \leftarrow 0$
 7:     Randomly permute the entries of $\mathrm{DB}(w)$
 8:     **for** each $id$ in $\mathrm{DB}(w)$ **do**
 9:         $c \leftarrow c + 1$
10:         $y \xleftarrow{\$} \mathbb{Z}_p^*$, $\mathcal{Y}[w, c] \leftarrow y$
11:         $k_d \xleftarrow{\$} \{0,1\}^\lambda$, $\mathcal{KD}[k_e, c] \leftarrow k_d$
12:         $e \leftarrow k_d \oplus id$
13:         $\mathrm{t} \leftarrow \mathrm{t} \cup \{(e, y)\}$
14:         $xtag \xleftarrow{\$} \mathbb{G}_p$
15:         $\mathcal{XG}[w, id] \leftarrow xtag$, $\mathcal{CD}[w, c] \leftarrow id$
16:         $\mathrm{XSet} \leftarrow \mathrm{XSet} \cup \{xtag\}$
17:     **end for**
18:     $\mathrm{T}[w] \leftarrow \mathrm{t}$
19:     **for** each $id \in \mathrm{D} \setminus \mathrm{DB}(w)$ **do**
20:         $\mathcal{XG}[w, id] \xleftarrow{\$} \mathbb{G}_p$
21:     **end for**
22:     **for** each $w' \in \mathrm{W} \setminus \{w\}$ **do**
23:         **for** $c = (|\mathrm{DB}(w)| + 1)$ to $|\mathrm{D}|$ **do**
24:             $\mathcal{XT}[w, w', c] \xleftarrow{\$} \mathbb{G}_p$
25:         **end for**
26:     **end for**
27: **end for**
28: $(\mathrm{TSet}, K_T) \leftarrow \mathrm{TSetSetup}(\mathrm{T})$
29: $\boxed{\mathrm{TSet} \leftarrow \mathcal{S}_T(N)}$
30: $K_F \leftarrow \mathrm{RHFSetup}(\lambda)$
31: $\mathrm{ES} \leftarrow \mathrm{RHFEncrypt}(K_F, \mathrm{XSet})$
32: $\boxed{\boxed{\mathrm{ES} \leftarrow \mathcal{S}_R(N)}}$
33: Send TSet and ES to the server
34: **return** $K = (K_T, K_F)$

    *Server*
35: **return** $\mathrm{EDB} = (\mathrm{TSet}, \mathrm{ES})$
```

Fig. 19. Setup Protocol in Games G_3, $\boxed{G_4}$, and $\boxed{\boxed{G_5}}$

```
**Search**$(K, q; \mathrm{EDB})$ in $G_3$, $\boxed{G_4}$, and $\boxed{\boxed{G_5}}$:
    *Client:*
 1: $(K_T, K_F) \leftarrow K$
 2: $stag \leftarrow \mathrm{TSetGetTag}(K_T, q[1])$
 3: $\boxed{stag \leftarrow \mathcal{S}_T(\mathrm{T}[q[1]])}$
 4: $(k_{f1}, k_{f2}) \leftarrow K_F$
 5: Send $stag$ and $k_{f1}$ to the server
 6: $strap \leftarrow \mathcal{S}[q[1]]$
 7: **for** $c = 1, 2 \cdots$ and until server sends $stop$ **do**
 8:     $\mathrm{xtoken}_c \leftarrow$ empty list
 9:     **for** $i = 2$ to $n$ **do**
10:         **if** $c \le |\mathrm{DB}(q[1])|$ **then**
11:             $id \leftarrow \mathcal{CD}[q[1], c]$
12:             $y \leftarrow \mathcal{Y}[q[1], c]$
13:             $xtok_{c,i} \leftarrow \mathcal{XG}[q[i], id]^{y^{-1}}$
14:         **else**
15:             $xtok_{c,i} \leftarrow \mathcal{XT}[q[1], q[i], c]$
16:         **end if**
17:         $\mathrm{xtoken}_c \leftarrow \mathrm{xtoken}_c \cup \{xtok_{c,i}\}$
18:     **end for**
19:     Send $\mathrm{xtoken}_c$ to the server
20: **end for**

    *Server:*
21: Run Line 15-29 in Figure 5
22: Send Responds to the client

    *Client:*
23: ck ← empty set, $k_e \leftarrow \mathcal{KZ}[strap, 2]$
24: **for** each $id \in \mathrm{DB}(q)$ **do**
25:     $c \leftarrow$ the position of $id$ within $\mathrm{DB}(w_1)$
26:     $k_d \leftarrow \mathcal{KD}[k_e, c]$, $\mathrm{ck} \leftarrow \mathrm{ck} \cup \{(c, k_d)\}$
27: **end for**
28: Send ck to the server

    *Server:*
29: Run Line 48-53 in Figure 5
30: Send R to the client
```

Fig. 20. Search Protocol in Games G_3, $\boxed{G_4}$, and $\boxed{\boxed{G_5}}$

```
$\boldsymbol{\mathcal{S}}(\boldsymbol{N})$:
    Client
 1: $t \leftarrow 0$
 2: TSet $\leftarrow \mathcal{S}_T(N)$, ES $\leftarrow \mathcal{S}_R(N)$
 3: Send TSet and ES to the server

**AssignId**$(\overline{w_1}, id, \mathcal{C}, \mathcal{CD}, \mathcal{DC})$:

 1: $c \xleftarrow{\$} \mathcal{C}[\overline{w_1}]$, $\mathcal{C}[\overline{w_1}] \leftarrow \mathcal{C}[\overline{w_1}] \setminus \{c\}$
 2: $\mathcal{CD}[\overline{w_1}, c] \leftarrow id$, $\mathcal{DC}[\overline{w_1}, id] \leftarrow c$, **return** $c$

$\boldsymbol{\mathcal{S}}.(\mathbf{DB}(\boldsymbol{q}), |\mathbf{DB}(\boldsymbol{q}[1])|, \mathbf{EP}(\boldsymbol{q}[1]), \mathbf{IP}(\boldsymbol{q}))$:

    Client:
 1: $t \leftarrow t + 1$                              ▷ $t$ is the timestamp
 2: $\overline{w_1} \leftarrow$ Min(EP($q[1]$))
 3: **if** $\overline{w_1} = t$ **then**
 4:     T$[\overline{w_1}] \leftarrow$ empty list
 5:     $strap \xleftarrow{\$} \{0,1\}^\lambda$, $\mathcal{S}[\overline{w_1}] \leftarrow strap$
 6:     $k_e \xleftarrow{\$} \{0,1\}^\lambda$, $\mathcal{KZ}[strap, 2] \leftarrow k_e$
 7:     **for** $c = 1$ to $|$DB$(q[1])|$ **do**
 8:         $e \xleftarrow{\$} \{0,1\}^\lambda$, $y \xleftarrow{\$} \mathbb{Z}_p^*$, $\mathcal{Y}[\overline{w_1}, c] \leftarrow y$
 9:         Append $(e, y)$ to T$[\overline{w_1}]$,
10:     **end for**
11:     $\mathcal{C}[\overline{w_1}] \leftarrow [1, |$DB$(q[1])|]$
12: **end if**
13: $stag \leftarrow \mathcal{S}_T($T$[\overline{w_1}])$
14: $(k_{f1}, k_{f2}) \leftarrow K_F$
15: Send $stag$ and $k_{f1}$ to the server
16: $\mathcal{EP}[t] \leftarrow \overline{w_1}$
17: $I \leftarrow$ the set of all the document identifiers existing in
    DB$(q)$ and IP$(q)$.
18: **for** each $id \in I$ **do**
19:     **if** $\mathcal{DC}[\overline{w_1}, id]$ does not exist **then**
20:         **AssignId**$(\overline{w_1}, id, \mathcal{C}, \mathcal{CD}, \mathcal{DC})$
21:     **end if**
22: **end for**
23: $(\mathrm{IP}(q[1], q[i]))_{i=2}^n \leftarrow \mathrm{IP}(q)$
24: **for** $i = 2$ to $n$ **do**
25:     $I_i \leftarrow$ the set of all the document identifiers existing
    in IP$(q[1], q[i])$
26:     **for** each $id \in I_i$ **do**
27:         $c \leftarrow \mathcal{DC}[\overline{w_1}, id]$
28:         Find the entry $(t', j, \Lambda)$ in IP$(q[1], q[i])$ such
    that $t'$ is the minimum timestamp whose corresponding
    $\Lambda$ contains $id$
      ▷ Next, proceed to determine the $xtag$ corresponding
    to $(q[i], id)$, denoted as $xtag_{c,i}$, where $c = \mathcal{DC}[\overline{w_1}, id]$
29:         **if** $\mathcal{XG}[t', j, id]$ exists **then**
30:             $xtag_{c,i} \leftarrow \mathcal{XG}[t', j, id]$
31:         **else**
32:             $\overline{w_1'} \leftarrow \mathcal{EP}[t']$
33:             $c' \leftarrow$ **AssignId**$(\overline{w_1'}, id, \mathcal{C}, \mathcal{CD}, \mathcal{DC})$
34:             $k \leftarrow 2$
35:             **while** $\mathcal{CXG}[t', k, c']$ exists **do**
36:                 $\mathcal{XG}[t', k, id] \leftarrow \mathcal{CXG}[t', k, c']$
37:                 Delete $\mathcal{CXG}[t', k, c']$
38:                 $k \leftarrow k + 1$
39:             **end while**
40:             $xtag_{c,i} \leftarrow \mathcal{XG}[t', j, id]$
41:         **end if**
42:     **end for**
43: **end for**
44: **for** each $id \in I$ **do**
45:     $c \leftarrow \mathcal{DC}[\overline{w_1}||id]$
46:     **for** $i = 2$ to $n$ **do**
47:         **if** $xtag_{c,i}$ does not exist **then**
48:             $xtag_{c,i} \xleftarrow{\$} \mathbb{G}_p$, $\mathcal{XG}[t, i, id] \leftarrow xtag_{c,i}$
49:         **end if**
50:     **end for**
51: **end for**
52: **for** each $c \in \mathcal{C}[\overline{w_1}]$ **do**
53:     **for** $i = 2$ to $n$ **do**
54:         $xtag_{c,i} \xleftarrow{\$} \mathbb{G}_p$, $\mathcal{CXG}[t, i, c] \leftarrow xtag_{c,i}$
55:     **end for**
56: **end for**
57: **for** $i = 2$ to $n$ **do**
58:     Find the entry $(t^*, j, \perp)$ in IP$(q[1], q[i])$ such that
    $t^*$ is the minimum timestamp for which $\mathcal{EP}[t^*] = \overline{w_1}$
59:     **if** $t^*$ exists **then**   ▷ Indicate that $q[1] = q^*[1]$ and
    $q[i] = q^*[j]$ for $q^*$ occurring at $t^*$ and for some $j \geq 2$
60:         **for** $c = |$DB$(q[1])| + 1$ to $|$D$|$ **do**
61:             $xtok_{c,i} \leftarrow \mathcal{XT}[t^*, j, c]$
62:         **end for**
63:     **else**
64:         **for** $c = |$DB$(q[1])| + 1$ to $|$D$|$ **do**
65:             $xtok_{c,i} \xleftarrow{\$} \mathbb{G}_p$, $\mathcal{XT}[t, i, c] \leftarrow xtok_{c,i}$
66:         **end for**
67:     **end if**
68: **end for**
69: **for** $c = 1, 2 \cdots$ and until server sends $stop$ **do**
70:     xtoken$_c \leftarrow$ empty list
71:     **for** $i = 2$ to $n$ **do**
72:         **if** $c \leq |$DB$(q[1])|$ **then**
73:             $y \leftarrow \mathcal{Y}[\overline{w_1}, c]$, $xtok_{c,i} \leftarrow xtag_{c,i}^{y^{-1}}$
74:         **end if**
75:         xtoken$_c \leftarrow$ xtoken$_c \cup \{xtok_{c,i}\}$
76:     **end for**
77:     Send xtoken$_c$ to the server
78: **end for**

    Server
79: Run Line 15-29 in Figure 5
80: Send Responds to the client

    Client:
81: ck $\leftarrow$ empty set, $k_e \leftarrow \mathcal{KZ}[strap, 2]$
82: **for** each $id \in$ DB$(q)$ **do**
83:     $c \leftarrow \mathcal{DC}[\overline{w_1}, id]$
84:     $(e, -) \leftarrow$ the $c$-th tuple in T$[\overline{w_1}]$
85:     $k_d \leftarrow e \oplus id$, ck $\leftarrow$ ck $\cup \{(c, k_d)\}$
86: **end for**
87: Send ck to the server

    Server:
88: Run Line 48-53 in Figure 5
89: Send R to the client
```

Fig. 21. The Simulator $\mathcal{S}$

$\mathrm{DB}(q[1])$ becomes randomized. To simulate this randomized correspondence, the simulator $\mathcal{S}$ utilizes the tables $\mathcal{C}$, $\mathcal{CD}$, and $\mathcal{DC}$. We note that simulating this correspondence is linked to simulating the generation of *xtags* later on.

For any search query q, where $\overline{w_1}$ represents the keyword $q[1]$, $\mathcal{C}[\overline{w_1}]$ is defined as a subset of $[1, |\mathrm{DB}(q[1])|]$, where a member c indicates that the document identifier corresponding to the c-th entry in $\mathrm{T}[\overline{w_1}]$ has not been determined. $\mathcal{C}[\overline{w_1}]$ is initialized as $[1, |\mathrm{DB}(q[1])|]$, while $\mathcal{CD}$ and $\mathcal{DC}$ are initially empty. Whenever $\mathcal{S}$ learns that a document identifier id belongs to $\mathrm{DB}(q[1])$, it invokes the function $\mathbf{AssignId}(\overline{w_1}, id, \mathcal{C}, \mathcal{CD}, \mathcal{DC})$, determining the entry in $\mathrm{T}[\overline{w_1}]$ that corresponds to id. This function uniformly and randomly selects an integer c from $\mathcal{C}[\overline{w_1}]$, removes c from $\mathcal{C}[\overline{w_1}]$, sets $\mathcal{CD}[\overline{w_1}, c] \leftarrow id$, and $\mathcal{DC}[\overline{w_1}, id] \leftarrow c$. Finally, the function returns c. Notably, during a search query q, $\mathcal{S}$ gains knowledge not only of the document identifiers belonging to $\mathrm{DB}(q[1])$ through $\mathrm{DB}(q)$ and $\mathrm{IP}(q)$ (Line 17-22) but also of document identifiers belonging to $\mathrm{DB}(q'[1])$ for any previous search query q' whose timestamp t' is included in $\mathrm{IP}(q)$. To address the latter case, $\mathcal{S}$ introduces a table $\mathcal{EP}$, which maps the timestamp t of a query q to $\overline{w_1}$ representing $q[1]$. When $\mathcal{S}$ learns that a document identifier id is a member of $\mathrm{DB}(q'[1])$ through $\mathrm{IP}(q)$ where q' occurred at t', it retrieves $\overline{w'_1}$ from $\mathcal{EP}[t']$ and invokes the function $\mathbf{AssignId}(\overline{w'_1}, id, \mathcal{C}, \mathcal{CD}, \mathcal{DC})$ to determine the entry in $\mathrm{T}[\overline{w'_1}]$ corresponding to id (Line 32-33).

Then, $\mathcal{S}$ simulates the token list $\mathrm{xtoken}_c = \{xtok_{c,i}\}_{i=2}^{n}$ for $1 \le c \le |\mathrm{DB}(q[1])|$. The key challenge of simulating $xtok_{c,i}$ lies in the simulation of $xtag_{c,i}$, denoting the *xtag* of $(q[i], id)$ where id is the document identifier corresponding to the c-th entry in $\mathrm{T}[\overline{w_1}]$. To ensure consistency with G_5's distribution of *xtags*, the *xtags* of keyword-document pairs must be computed deterministically, and the *xtag* of a fresh keyword-document pair should be selected uniformly at random from $\mathbb{G}_p$. During a search query q at timestamp t, to obtain the *xtag* of a keyword-document pair $(q[i], id)$ for an $id \in \mathrm{DB}(q[1])$, $\mathcal{S}$ must check whether the *xtag* of $(q[i], id)$ has been computed in a previous search query. This information is captured by the leakage profile $\mathrm{IP}(q[1], q[i])$, which is a part of $\mathrm{IP}(q)$. If there exists an entry $(t', j, \Lambda) \in \mathrm{IP}(q[1], q[i])$ such that id is a member of the set Λ, it indicates that the *xtag* of $(q[i], id)$ has been computed in the search query q' occurring at timestamp t', where $q[i] = q'[j]$.

$\mathcal{S}$ must provide a method that allows a search query to access the *xtag* it needs, which has been computed by previous search queries. To achieve this, $\mathcal{S}$ utilizes the table $\mathcal{XG}$ with a minor modification: for any search query q occurring at any timestamp t and $2 \le i \le n$, if q is the earliest query that requires the *xtag* of $(q[i], id)$ (denoted as $xtag_{c,i}$ where $c \leftarrow \mathcal{DC}[\overline{w_1}, id]$), during the execution of q, $\mathcal{S}$ selects the $xtag_{c,i}$ from $\mathbb{G}_p$ uniformly at random and stores it in $\mathcal{XG}[t, i, id]$ (instead of $\mathcal{XG}[q'[i], id]$ in G_5) (Line 44-51). However, the subtle point is that the value of id may not be revealed to $\mathcal{S}$ during the execution of q, as id may not be exposed by $\mathrm{IP}(q)$ and $\mathrm{DB}(q)$. To address this issue, $\mathcal{S}$ uses table $\mathcal{C}$ and additionally introduces a new table: $\mathcal{CXG}$. For each $c \in \mathcal{C}[\overline{w_1}]$ and $2 \le i \le n$, $\mathcal{S}$ selects $xtag_{c,i}$ from $\mathbb{G}_p$ uniformly at random and stores $xtag_{c,i}$ into $\mathcal{CXG}[t, i, c]$ (Line 52-56).

Next, we illustrate how $\mathcal{S}$ acquires the *xtags* that have been computed by previous search queries through $\mathrm{IP}(q)$ (Line 24-43). For $2 \le i \le n$, $\mathcal{S}$ extracts all the document identifiers present in $\mathrm{IP}(q[1], q[i])$ and stores them in the set I_i. For each $id \in I_i$, $\mathcal{S}$ finds the entry (t', j, Λ) in $\mathrm{IP}(q[1], q[i])$ such that t' is the earliest timestamp whose corresponding Λ contains id. Based on the explanation in the previous paragraph, the *xtag* of $(q[i], id)$ is either stored in $\mathcal{XG}[t', j, id]$ or included in $\mathcal{CXG}$. In the former case, $\mathcal{S}$ sets $xtag_{c,i} \leftarrow \mathcal{XG}[t', j, id]$ where $c \leftarrow \mathcal{DC}[\overline{w_1}, id]$. In the latter case, it indicates that the position of the entry in $\mathrm{T}[\overline{w'_1}]$ ($\overline{w'_1} \leftarrow \mathcal{EP}[t]$) corresponding to id is still uncertain. In this situation, $\mathcal{S}$ assigns id to the c'-th entry in $\mathrm{T}[\overline{w'_1}]$ and updates $\mathcal{XG}$ and $\mathcal{CXG}$ by setting $\mathcal{XG}[t', k, id] \leftarrow \mathcal{CXG}[t', k, c']$ and deleting $\mathcal{CXG}[t', k, c']$, for any k for which $\mathcal{CXG}[t', k, c]$ exists. After that, the *xtag* of $(q[i], id)$ is precisely $\mathcal{XG}[t', j, id]$.

After handling the case where $1 \le c \le |\mathrm{DB}(q[1])|$, $\mathcal{S}$ is also tasked with simulating the token list $\mathrm{xtoken}_c = \{xtok_{c,i}\}_{i=2}^{n}$ for $c > |\mathrm{DB}(q[1])|$. The key to successfully achieve this simulation lies in ensuring that $xtok_{c,i}$ follows a uniform distribution across $\mathbb{G}_p$, and its value is determined solely by $q[1]$, $q[i]$, and c. We achieve this by employing the table $\mathcal{XT}$. Within our approach, $\mathcal{XT}[q[1], q[i], c]$ from G_5 is rewritten as $\mathcal{XT}[t^*, j, c]$, where t^* represents the timestamp of the earliest query q^* that employs $q[1]$ as the *s-term* and $q[i]$ as one of the *x-terms* for some $2 \le i \le n$, and j denotes the corresponding index where $q^*[j] = q[i]$. As detailed in Line 58-62, $\mathcal{S}$ accesses $xtok_{c,i}$ ($c > |\mathrm{DB}(q[1])|$) that has already been computed through previous search queries, using the insights gleaned from the leakage profile $\mathrm{IP}(q)$ and table $\mathcal{XT}$. In instances where $xtok_{c,i}$ ($c > |\mathrm{DB}(q[1])|$) has not yet been calculated by previous searches, $\mathcal{S}$ uniformly selects $xtok_{c,i}$ from $\mathbb{G}_p$ at random and assigns it to $\mathcal{XT}[t,i,c]$, as illustrated in Line 63-66.

By following the above process, $\mathcal{S}$ successfully simulates the generation of $\{xtag_{c,i}\}_{i=2}^{n}$ for $1 \leq c \leq |\mathrm{DB}(q[1])|$ and $\{xtok_{c,i}\}_{i=2}^{n}$ for $|\mathrm{DB}(q[1])|+1 \leq c \leq |\mathrm{D}|$. With $\{xtag_{c,i}\}_{i=2}^{n}$ for $1 \leq c \leq |\mathrm{DB}(q[1])|$, $\mathcal{S}$ computes $xtok_{c,i} \leftarrow xtag_{c,i}^{\mathcal{Y}[\overline{w_1}||c]^{-1}}$, just as done in G_5.

In the final step, $\mathcal{S}$ is required to simulate ck. The sole distinction from G_5 in this context is that $\mathcal{S}$ calculates k_d using the formula $e \oplus \mathrm{id}$ instead of retrieving it from table KD. Importantly, this alteration does not impact the distribution of k_d since, in both games, k_d is deterministic with respect to keyword $q[1]$ and the counter c, and uniformly distributed over $0, 1^{\lambda}$.

Therefore, we can get that

$$\Pr[G_5 = 1] = \Pr[\mathrm{SSEI{\scriptsize DEAL}}_{\mathcal{A},\mathcal{S},\mathcal{L}_{hbs}}^{\mathsf{HBS}}(\lambda) = 1] \tag{14}$$

By summarizing equations (8) to (14), we can draw a conclusion.

$$|\Pr[\mathrm{SSER{\scriptsize EAL}}_{\mathcal{A}}^{\mathsf{HBS}}(\lambda) = 1] - \Pr[\mathrm{SSEI{\scriptsize DEAL}}_{\mathcal{A},\mathcal{S},\mathcal{L}_{hbs}}^{\mathsf{HBS}}(\lambda) = 1]| \leqslant Adv_{\mathsf{HBS},\mathcal{B}_h}^{corr}(\lambda) + (2|\mathrm{W}|+1) \cdot Adv_{F,\mathcal{B}_f}^{\mathrm{PRF}} + (|\mathrm{W}|+2) \cdot Adv_{F_p,\mathcal{B}_f}^{\mathrm{PRF}} + Adv_{\mathbb{G}_p,\mathcal{B}_d}^{\mathrm{DDH}}(\lambda) + Adv_{tset,\mathcal{B}_t}^{\text{T-set}}(\lambda) + Adv_{\pounds,\mathcal{B}_r}^{\mathrm{RHF}}(\lambda) \tag{15}$$

E Multi-client Setting with Malicious Clients

A malicious client could potentially attempt unauthorized access, such as utilizing $\mathcal{M}\mathrm{tok}_u$ for $w_1 \wedge w_2 \wedge w_3 \wedge w_4$ to search for documents matching $w_1 \wedge w_2 \wedge w_3$. To address this challenge, a solution can be implemented by having $\mathcal{D}$ and the server share a secret key k_u and employing a keyed hash function denoted as $H_k : \{0,1\}^{\lambda} \times \{0,1\}^{*} \rightarrow \{0,1\}^{\lambda}$. When $\mathcal{D}$ authorizes $\mathcal{U}$ to execute a search query $w_1 \wedge \psi(w_2, \cdots, w_n)$, $\mathcal{D}$ takes the initiative to generate the necessary tokens that $\mathcal{U}$ must transmit to the server. These tokens include $stag$, k_{f1}, $\{\mathrm{xtoken}_c\}_{c=1}^{|\mathrm{DB}(w_1)|}$, and their corresponding ck. Note that $\mathcal{D}$ can obtain these data by executing this particular search query. Subsequently, $\mathcal{D}$ constructs $\mathcal{M}\mathrm{tok}_u = (stag, strap, X, K_F, sig_1 = H_k(k_u, stag||k_{f1}||\{\mathrm{xtoken}_c\}_{c=1}^{|\mathrm{DB}(w_1)|}), sig_2 = H_k(k_u, sig_1||\mathrm{ck}))$ and transmits $\mathcal{M}\mathrm{tok}_u$ to client $\mathcal{U}$.

In the execution of $w_1 \wedge \cdots \psi(w_2, \cdots, w_n)$, alongside the HBS search procedure, the client $\mathcal{U}$ also sends sig_1 to the server during the first round of the search. The server, upon verifying that $H_k(k_u, stag||k_{f1}||\{\mathrm{xtoken}_c\}_{c=1}^{|\mathrm{DB}(w_1)|})$ is equivalent to sig_1, proceeds to transmit the response Responds for the first round to the client. If, on the other hand, the verification fails, the server halts the service. In the second round, after receiving Responds, the client computes the corresponding ck and sends it to the server along with sig_2. The server then verifies whether sig_2 matches $H_k(k_u, sig_1||\mathrm{ck})$. Only if this verification step succeeds will the server continue with the service. This robust mechanism ensures that the client cannot engage in unauthorized access, as it lacks the capability to forge sig_1 and sig_2 without knowledge of k_u.