Result Pattern Hiding Searchable Encryption for Conjunctive Queries

Shangqi Lai*
Monash University
Melbourne, Australia
shangqi.lai@monash.edu

Sikhar Patranabis
Indian Institute of Technology
Kharagpur, India
sikhar.patranabis@iitkgp.ac.in

Amin Sakzad
Monash University
Melbourne, Australia
amin.sakzad@monash.edu

Joseph K. Liu†
Monash University
Melbourne, Australia
joseph.liu@monash.edu

Debdeep Mukhopadhyay
Indian Institute of Technology
Kharagpur, India
debdeep@iitkgp.ac.in

Ron Steinfeld
Monash University
Melbourne, Australia
ron.steinfeld@monash.edu

Shi-Feng Sun*†
Monash University
Melbourne, Australia
shifeng.sun@monash.edu

Dongxi Liu
Data61, CSIRO
Syndey, Australia
dongxi.liu@data61.csiro.au

Cong Zuo*
Monash University
Melbourne, Australia
cong.zuo1@monash.edu

ABSTRACT

The recently proposed Oblivious Cross-Tags (OXT) protocol (CRYPTO 2013) has broken new ground in designing efficient searchable symmetric encryption (SSE) protocol with support for conjunctive keyword search in a single-writer single-reader framework. While the OXT protocol offers high performance by adopting a number of specialised data-structures, it also trades-off security by leaking 'partial' database information to the server. Recent attacks have exploited similar partial information leakage to breach database confidentiality. Consequently, it is an open problem to design SSE protocols that plug such leakages while retaining similar efficiency. In this paper, we propose a new SSE protocol, called Hidden Cross-Tags (HXT), that removes 'Keyword Pair Result Pattern' (KPRP) leakage for conjunctive keyword search. We avoid this leakage by adopting two additional cryptographic primitives - Hidden Vector Encryption (HVE) and probabilistic (Bloom filter) indexing into the HXT protocol. We propose a 'lightweight' HVE scheme that only uses efficient symmetric-key building blocks, and entirely avoids elliptic curve-based operations. At the same time, it affords selective simulation-security against an unbounded number of secret-key queries. Adopting this efficient HVE scheme, the overall practical storage and computational overheads of HXT over OXT are relatively small (no more than 10% for two keywords query, and 21% for six keywords query), while providing a higher level of security.

*Also with Data61, CSIRO, Melbourne, Australia.
†Corresponding authors.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243753

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**; **Management and querying of encrypted data**;

KEYWORDS

Searchable Encryption, Hidden Vector Encryption, Leakage Profile

ACM Reference Format:
Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K. Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shi-Feng Sun, Dongxi Liu, and Cong Zuo. 2018. Result Pattern Hiding Searchable Encryption for Conjunctive Queries. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243753

1 INTRODUCTION

Privacy of stored data is important in many applications. Yet it is becoming increasingly common for data to be hosted off-site, especially with the rise of cloud computing. However, cloud storage providers often cannot be trusted to respect the privacy of the data they host [13], especially in the face of malicious insiders. A simple solution is to encrypt the data before uploading it to the cloud. However, this would prevent the data from being searched [26, 27]. For example, users may wish to use their mobile phone to search their email. The cloud server will not be able to identify which documents match the search query if the user's email data is encrypted using standard encryption techniques.

Searchable symmetric encryption (SSE) [11, 12, 15, 18, 21, 31, 33, 34] offers a potential solution to this problem by providing a way for encrypted data to be searched securely. However, all SSE protocols must have trade-off between security, performance, and functionality. The performance of SSE protocols is crucial in practice and needs to be comparable to performance of search on unencrypted databases in order to remain feasible for most practical applications. Consequently, recent research has focused on high performance SSE protocols that scale efficiently to large databases, by adapting data structures used in efficient unencrypted database search to the

encrypted database setting. In particular, an efficient class of SSE protocols [11, 12, 15, 18, 34] preprocessing the database using an *inverted-index* to allow keyword searches to be performed in sub-linear time, with a careful choice of encryption techniques applied to the index to hide its contents and the queries from the server, while allowing a flexible range of search queries. The benchmark in this class of SSE protocols (supporting conjunctive queries) is 'Oblivious Cross-Tags' (OXT) [12]. Nevertheless, to achieve their high performance, these protocols do trade-off security. In particular, they leak some 'partial' information to the server, both on the queries themselves, as well as on the database contents. Recent attacks [10, 24, 28, 37] have shown that such 'partial' leakages can sometimes be exploited in practical applications, especially when the attacker has available some auxiliary side information (which may be obtained publicly in many cases). This state of affairs motivates a re-examination of the security-efficiency trade-offs for SSE protocols and raises the question:

Is it possible to reduce the leakage in existing state-of-the-art efficient SSE protocols while preserving a practical level of performance?

In this paper, we make progress on this question. We focus, like OXT, only on the setting of conjunctive keyword queries, since such queries tend to be the most common in many practical applications. In this setting, queries consist of any number of keywords, and the result consists of all documents containing all those keywords.

Overview of OXT**.** To explain the main technical ideas behind our construction, it is instructive to first briefly review the 'TSet' and 'XSet' data structures of OXT and how they are used to efficiently process conjunction queries of the form $w_1 \wedge \cdots \wedge w_n$. In OXT, TSet is cryptographic data structure that associates a list of fixed-length data tuples to each keyword in a database. It is an encrypted inverted index that enables the owner to issue corresponding tokens to retrieve these lists related to the queried keywords. In particular, the client sends the server a 'search token' (called an stag) related to the keyword w_1 (called the 's-term' and denoted by sterm), which allows the server to retrieve from the TSet, the set $\mathsf{DB}(w_1)$ of all database documents containing w_1. In addition, the client sends 'intersection tokens' (called 'xtraps') related to the $n-1$ keyword *pairs* (w_1, w_i) consisting of the 's-term' paired with each of the remaining query keywords w_i, $2 \le i \le n$ (called 'x-terms'). The intersection tokens allow the server to *filter* the set $\mathsf{DB}(w_1)$ to determine the $n-1$ subsets of documents $\mathsf{DB}(w_1) \cap \mathsf{DB}(w_i)$ that contains the pairs (w_1, w_i), returning only those documents that contain all $\{w_i\}_{1 \le i \le n}$. The intersection subsets $\mathsf{DB}(w_1) \cap \mathsf{DB}(w_i)$ are efficiently computed by the server using the 'XSet' data structure; the 'XSet' is in essence a list of hashed pairs $h(\mathsf{id}, w)$, over all database document identities id and keywords w contained in id, where h is a certain (public) cryptographic hash function. To filter $\mathsf{DB}(w_1)$ to compute $\mathsf{DB}(w_1) \cap \mathsf{DB}(w_i)$, the server runs through each document id in $\mathsf{DB}(w_1)$ and checks, using the xtrap token for (w_1, w_i), whether $h(\mathsf{id}, w_i)$ is in the XSet. Therefore, the server computation time is dominated by $|\mathsf{DB}(w_1)|(n-1)$ evaluations of h, which is proportional to just the number of database documents containing the *least frequent* 's-term' w_1, even if other 'xterm' keywords are much more common. However, this method also reveals to the server the Keyword Pair Result Pattern (KPRP, called RP in [12]), i.e. the partial result sets $\mathsf{DB}(w_1) \cap \mathsf{DB}(w_i)$ for each pair (w_1, w_i).

Result Pattern (RP**) Leakage in SSE protocols.** Information leakage in SSE protocols can be classified into three types: storage leakage, query pattern leakage, and result pattern leakage. In this paper, we focus on the third type of leakage, namely result pattern leakage (RP), i.e. information leaked to the server on the query results. *Ideally, in the conjunctive query SSE context, result pattern leakage would reveal only the Whole Result Pattern (*WRP*), i.e. the number (and possibly also, identities) of documents matching all query keywords.* Leaking the WRP is in most SSE settings unavoidable, as it would be leaked to the server via the document identities retrieved by the client after the query (unless an ORAM or PIR protocol is used, which currently seems impractical). But in OXT, RP leakage is larger than the WRP ideally desired, and comes in three forms: single keyword result pattern (SP) leakage, Keyword-Pair Result Pattern (KPRP) leakage and multiple keyword cross-query intersection result pattern (IP) leakage.

The KPRP leakage is a 'non-ideal' leakage related to multiple keywords in each query. This reveals *partial query results* to the database server; for an n keyword conjunction query $w_1 \wedge \cdots \wedge w_n$, with w_1 designated as the 's-term' keyword, the KPRP reveals to the server the set $\mathsf{DB}(w_1) \cap \mathsf{DB}(w_i)$ of documents containing every *pair* of query keywords of the form (w_1, w_i), $2 \le i \le n$. This may leak significantly more information on the inverted-index and/or the query than what is leaked by WRP, which is the set $\cap_{j=1}^{n} \mathsf{DB}(w_j)$ of documents matching all n query keywords.

The recent proposed file-injection attacks [37] have an adapted version which exploits KPRP leakage in OXT to reveal all keywords of a conjunctive queries with 100% accuracy. Thus, finding an SSE protocol that eliminates KPRP leakage may be a feasible way to mitigate such attack towards the conjunctive queries on the inverted-index.

Our Contributions. In this paper, we present the first efficient SSE protocol, called 'Hidden Cross-Tags' (HXT) that eliminates KPRP leakage presented in the state-of-the-art OXT protocol [12]. Our protocol removes the KPRP 'partial query' leakage component, leaving in its place only the minimal (in our SSE context) and significantly smaller WRP. Furthermore, it leaves other query and result pattern leakage components in existing SSE protocols (i.e. SP and IP) unchanged. Thus, in terms of security, our protocol offers strictly better guarantees than OXT protocol, and is likely to significantly reduce the rate of query pattern leakage to the database server, as discussed above.

The improved security of our protocol as compared to OXT may be attributed to the adoption of two additional cryptographic primitives - namely, Hidden Vector Encryption (HVE) and probabilistic (Bloom filter) indexing [4]. HVE is a functional encryption primitive typically used in the public-key setting. All existing HVE schemes [6, 23, 25, 30] in the cryptographic literature use either prime order or composite order bilinear pairings over elliptic curve groups. In our HVE-based scheme, a dataset owner can encrypt a set $S \subseteq T = \{1, \ldots, n\}$, for some positive integer n, into a ciphertext c_S, which specifies a 'policy'. Using a master secret key msk, the owner can generate a search token $tk_{S'}$ for any subset $S' = \{s'_1, \ldots, s'_\ell\}$

of T. Using the token $tk_{S'}$ for S' and the ciphertext c_S for S, anyone can efficiently check whether $S' \subseteq S$ or not, without leaking any partial information if $S' \not\subseteq S$, e.g. whether any particular element s'_i of S' is in S or not (note that in the scheme of [6], the set S is used to generate the token, while the set S' is encrypted in the ciphertext). Unfortunately, adopting such schemes into our protocol leads to a huge compromise in overall performance and efficiency. We address this issue by proposing a 'lightweight' HVE scheme that only uses efficient symmetric-key building blocks, and entirely avoids elliptic curve-based operations. At the same time, it affords selective simulation-security against an unbounded number of secret-key queries. Given this subset membership searchable encryption protocol, a natural idea to apply it to eliminate KPRP leakage in OXT would be to use it to *encrypt the* 'XSet' during set up: we let $S \subseteq T$ denote the XSet list of hashed pairs $h(\mathsf{id}, w)$, over all documents id and keywords w contained in id, and we encrypt S into a ciphertext c_S stored on the server using our HVE-based subset searchable encryption scheme. In the search phase with query $w_1 \wedge \cdots \wedge w_n$, the client issues the server a HVE search token $tk_{S'}$ for $S' = \{h(\mathsf{id}, w_i)\}_{i=2}^{n}$, $\mathsf{id} \in \mathsf{DB}(w_1)$. This allows the client to check whether $S' \subseteq S$, i.e., whether id contains all n keywords $\{w_i\}_{1 \le i \le n}$, without revealing the KPRP information on whether id contains any particular pair (w_1, w_i).

To demonstrate the high performance of our protocols and assess the efficiency overhead of our security improvements, we report our experimental results on the implementation of our protocol and its performance evaluation, compared with the original OXT protocol.

Security of HXT**.** We prove the client privacy of HXT against the "honest-but-curious" server in a natural extension of the security model used to prove the security of OXT [12], assuming the security of the underlying HVE scheme, the DDH assumption, and the security of the employed symmetric key primitives. The main difference from [12] is that the leakage function in our security model is weaker, as it eliminates the KPRP leakage component and replaces it with the smaller WRP.

Practicality of HXT**.** We believe HXT is a practical, more secure alternative to OXT for searching large databases. To support this statement, we discuss the practical storage, computation, and communication costs of HXT versus OXT (see Sec. 6 and Sec. 7 for more details and experimental results). In terms of storage, we note that, like HXT, practical implementations of OXT [12] also use a Bloom filter for the XSet, and thus have a similar false positive probability P_e and overhead factor in number of array storage locations. One additional overhead in our HXT protocol is that each binary entry of our Bloom filter array is encrypted into 1 PRF generated value, i.e. typically 128 bits, whereas these bits are unencrypted in OXT, leading to up to 128 factor overhead (in bit length) in XSet storage for HXT over OXT. While this factor is quite large, our theoretical analysis shows that for a fixed $P_e = 10^{-6}$, the XSet Bloom filter storage size in OXT is 35 times smaller than the TSet hash table size (which is the same in HXT and OXT). Thus, the *overall* storage overhead factor of HXT over OXT without Bloom filter will typically be only $1 + 128/35 \approx 4.65$, which we believe is quite practical.

In terms of computational cost, the HXT has additional costs compared to OXT due to the use of HVE (whereas the practical implementations of OXT only uses plain Bloom filter). However, our proposed HVE is based on symmetric key primitives and bitwise operations. As a consequence, for typical parameters (e.g. $n = 1$ or $n = 2$ keyword queries with false positive rate $P_e = 10^{-6}$), we estimate that the *overall* server computation time overhead for HXT over OXT in such practical applications is likely to be less than 1% and is fully overlapped by I/O costs. Our implementation results indeed show a server runtime overhead between $2\% - 8\%$ for a two keywords query, and 21% for the six keywords query. The HXT client computational cost overhead factor over OXT is higher at $\approx 0.06 * \log_2(1/P_e) \approx 1.19$ for $P_e = 10^{-6}$, but this is likely acceptable as the overall client search time may be dominated by 'out of protocol' costs such as the communication time for downloading result documents. In terms of communications, HXT adds one extra round of communication over the OXT protocol, in which the server communicates to the client the Bloom filter subset it needs to check, and the client returns a HVE token to allow this search. The communication length is still, as in OXT, only proportional to the number of results for the *least-frequent* query keyword.

Additional Related Work. In practice, efficient unencrypted search algorithms usually use a precomputed database *index*. This allows keyword searches to be performed in essentially sublinear time with respect to the size of the database (or more precisely, in time proportional only to the number of *results* matching the query). A number of index based SSE protocols have been proposed, each more efficient than its predecessor. The first secure encrypted index was proposed in [21], based on the form of *forward* index, storing for each document a Bloom filter containing all the document's keywords. This allows a single document to be searched in $O(1)$ time but requires each document to be checked in turn, with complexity proportional to the number of documents in the database.

Curtmola et al. [15] were the first to propose using an *inverted-index* data structure, storing in a hash table for each keyword, the encrypted IDs of the documents that contain it (while hiding the number of documents matching each keyword), resulting in complexity proportional to the number of matching *results*, even for searching the whole document collection. However, [15] does not support multiple keyword *conjunctive* queries efficiently; it has complexity proportional to the number of documents matching the *most frequent* queried keyword. Later, [12] presented the OXT protocol, extending [15] by adding a 'Cross-Tag Set' (XSet) data structure, which lists hashed pairs of keywords and IDs of documents containing them, and reducing search complexity to be proportional to the number of results matching the *least frequent* queried keyword. Our HXT protocol is an improvement of OXT, replacing the XSet by another encrypted data structure to eliminate KPRP leakage, while preserving a low search complexity.

SE protocols have also been studied extensively in the *public key* setting. Such protocols allow any user with the public key to insert data but only allow the user with the private key to search. The use of public key cryptography makes the protocols less efficient than SSE, but allows more powerful functionality and/or better security properties. The first such protocol was proposed by [5] as a generalization of anonymous Identity-Based Encryption (IBE), and supporting equality queries. It was significantly further generalized in [6] to HVE, applied to conjunctive, subset and range searchable encryption queries. However, it is not clear how to use it to obtain

Table 1: Notations and Terminologies

Notation	Meaning
λ	a security parameter
id_i	the document identifier of the i-th document
d	number of documents in the database
w	a keyword
W_i	forward index {all w contained in id_i}
W	the set of all keywords $\cup_{i=1}^{d}\mathsf{W}_i$
ω	the number of elements in W
DB	database $(\mathsf{id}_i, \mathsf{W}_i)_{i=1}^{d}$
$\mathsf{DB}(w)$	inverted-index {all id : $w \in \mathsf{W}_{\mathsf{id}}$}
N	the number of all pairs of (id, w) in DB
n	max. number of conjunctive keywords per query
sterm	the least frequent term among queried terms
xterm	other queried terms (excluding sterm)
$\mathcal{E}$	result set on server side (encrypted)
R	result set on client side
Q	the number of all queries
$[n]$	the set of integers $\{1, 2, \ldots, n\}$
k	the number of hash functions in a Bloom filter
m	the length of a Bloom filter
P_e	query false positive probability (per sterm result)
$\mathsf{negl}(\lambda)$	a negligible function in λ
$s \xleftarrow{\$} S$	uniformly sampling a random s from S
$a \leftarrow \mathcal{A}(\cdot)$	obtaining a as output of running algorithm $\mathcal{A}$

efficient SSE protocols for conjunctive keyword queries. Our HXT protocol fills this gap by proposing a symmetric-key HVE which is significantly more efficient and suffices for symmetric-key setting of SSE.

2 PRELIMINARIES

We first give a list of notations and definitions needed in our construction and security analysis. A summary of notations and terminologies used in this paper is given in Table 1. All needed hardness assumptions and other necessary cryptographic primitives including symmetric encryption scheme and pseudo-random functions are given in the full version.

2.1 T-set

A T-set is an expanded inverted-index data structure [12] used for efficient SSE. It is a cryptographic data structure that associates a list of fixed-length data tuples to each keyword in a database. Later it enables the owner to issue corresponding tokens to retrieve these lists related to the queried keywords. A syntax, a correctness definition, a security model, and an instantiation of such a hash table is given in [12]. Throughout this paper, we identically adapt the notations, definitions and results (in particular, those of Appendix 2.2 and Sec. 6) of that paper regarding T-sets here. To summarise what we use in this work, we give an instantiation of a T-set along with a result regarding its security.

An instantiation of a T-set consists of three algorithms:

- TSet.Setup($\mathbf{T}$): The input to this algorithm is an array $\mathbf{T}$ and the output is TSet with b buckets of size s each. This procedure places the i-th element of $\mathbf{T}[w]$ into bucket $\mathsf{TSet}[b]$, where b is obtained using two secure pseudorandom functions (PRFs) F and $\bar{F}$ and a hash function H.
- TSet.GetTag(w): The input to this procedure is a keyword w and it returns $\bar{F}(w)$.
- TSet.Retrieve($\bar{F}(w)$): This procedure accepts the output of TSet.GetTag and TSet as its inputs and retrieves the same bucket $\mathsf{TSet}(b)$ and recovers $\mathbf{t} = \mathbf{T}[w]$.

The only leakage of T-set instantiation in [12] is

$$N = \sum_{w \in \mathsf{W}} |\mathbf{T}[w]| = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|, \tag{1}$$

that is the number of all pairs of (id, w) in DB.

Theorem 1 (Th. 7 of [12]). *For any keyword sequence* $\mathbf{q}$*, including an empty sequence, define* $\mathcal{L}_{\mathbf{T}}(\mathbf{T}, \mathbf{q})$ *as* (1)*. The T-set instantiation* Γ *is* $\mathcal{L}_{\mathbf{T}}$*-adaptively-secure assuming that* F *and* $\bar{F}$ *are secure* PRF*s and that* H *is a random oracle.*

2.2 Searchable Encryption: Definition and Security

In our single-writer single-reader setting, there are two parties: the data owner (called *client*) of the plaintext database and a cloud service provider (called *server*) that stores the encrypted database. The client can interactively perform search queries over the database. In more details, the client outsources her search service to the server. When she wants to perform a search query, she generates the search token by herself using her private key and forwards the token to the server. With the token, the server retrieves the encrypted identifier or documents for the client.

Formally, the syntax of our SSE protocol Π consists of the following algorithms:

- $\mathsf{SE.EDBSetup}(1^\lambda, \mathsf{DB}) \rightarrow (param, mk, \mathsf{EDB})$: Run by client, takes 1^λ and DB as inputs and returns the system public parameters $param$, master key mk and encrypted database EDB. $param$ is publicly known. mk is kept by client and EDB is stored in the server.
- $\mathsf{SE.Search}(param, mk, \psi(\mathbf{w}), \mathsf{EDB}) \rightarrow \mathsf{DB}(\psi(\mathbf{w}))$: A protocol, runs between client and server interactively. Client's inputs are $param$, mk and query $\psi(\mathbf{w})$, while server's inputs are $param$ and EDB. At the end of the protocol, client outputs document identifiers $\mathsf{DB}(\psi(\mathbf{w}))$ matching query $\psi(\mathbf{w})$, and server outputs nothing.

We say that Π is *computationally correct with false positive rate* P_e if for any database DB of size $\mathsf{poly}(\lambda)$ and any conjunctive query $\psi(\mathbf{w}) = w_1 \wedge \cdots \wedge w_n$, the following game $\mathbf{Cor}^{\Pi}_{\mathcal{A}}(\lambda)$ is won with probability at most $|\mathsf{DB}(w_1)| \cdot P_e + \mathsf{negl}(\lambda)$. In this game, the challenger runs SE.EDBSetup to get EDB from DB and simulates SE.Search on EDB and query $\psi(\mathbf{w})$ to compute the client search result S. The game is won (and returns 1) if $S \neq \mathsf{DB}(\psi(\mathbf{w}))$. We remark that this relaxed correctness definition, allowing a 'per sterm result' false positive rate P_e, is also required for the Bloom filter based practical implementation of OXT (though not formalised in [12]).

We consider the following security model for SSE, which is exactly the one from [12], except that our leakage function will

reveal WRP instead of the KPRP (known as RP) in [12]. The model is parameterised by a leakage function $\mathcal{L}$, as described below, which captures information allowed to learn by an adversary from the interaction with a secure searchable encryption protocol. Loosely speaking, the security says that the server's view during a non-adaptive attack can be properly simulated given only the output of the leakage function $\mathcal{L}$.

Let Π = (SE.EDBSetup, SE.Search) be a searchable encryption protocol and $\mathcal{A}$ and $\mathcal{S}$ be two efficient algorithms. The security is formally defined via a real experiment $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$ and an ideal experiment $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda)$ as follows:

- $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$: The adversary $\mathcal{A}(1^{\lambda})$ chooses a database DB and a query list $\mathbf{q}$. Then the experiment runs SE.EDBSetup(1^{λ}, DB) to get ($param, mk$, EDB) and returns $param$ and EDB to $\mathcal{A}$. After that, for each $i \in |\mathbf{q}|$, the experiment runs the SE.Search on input $\mathbf{q}[i]$, and stores the resulted transcript and the client's output into $\mathbf{t}[i]$. Finally, EDB and $\mathbf{t}$ will be given to $\mathcal{A}$. Eventually, the experiment outputs the bit that $\mathcal{A}$ returns.
- $\mathbf{Ideal}^{\Pi}_{\mathcal{A},\mathcal{S}}(\lambda)$ The adversary $\mathcal{A}(1^{\lambda})$ chooses a database DB and a query list $\mathbf{q}$. Then the experiment runs EDB $\leftarrow \mathcal{S}(\mathcal{L}($ DB, $\mathbf{q}))$ and gives it to $\mathcal{A}$. Eventually, the experiment outputs the bit that $\mathcal{A}$ returns.

Definition 1 (Security). *The searchable encryption protocol Π is called $\mathcal{L}$-semantically secure against non-adaptive attacks if for all PPT adversaries $\mathcal{A}$ there exists an efficient simulator $\mathcal{S}$ such that*

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

An adaptive model can also be defined correspondingly as in [12]. In such a model, the query list $\mathbf{q}$ will not be known to the challenger at the beginning of the real and ideal games. Instead, it selects repeatedly search query $\mathbf{q}$ after each experiment runs SE.EDBSetup. Note that in the real game, the input to SE.Search is only EDB and to generate EDB, the simulator $\mathcal{S}$ has only access to the leakage function $\mathcal{L}$(DB, $\mathbf{q}$).

2.3 Bloom Filters

A Bloom filter (BF) is a probabilistic (indexing) data structure to represent a set $\mathcal{V} = \{s_1, s_2, \ldots, s_N\}$ of N elements. Its main functionality is to support membership queries. The idea is to choose k independent hash functions, $\{H_i\}_{1 \leq i \leq k}$, each with domain $\mathcal{V}$ and range $[m]$. The Bloom filter consists of a binary vector $\mathbf{b}$ of m bits, initially all 0. In order to set up BF for $\mathcal{V}$, for each element $s \in \mathcal{V}$, the bits at positions $\{H_i(s)\}_{1 \leq i \leq k}$ are changed to 1. To test membership of q, we check if $\mathbf{b}$ has 1's in all positions $\{H_i(q)\}_{1 \leq i \leq k}$, and if so, we conclude $q \in \mathcal{V}$ with high probability. Otherwise, we conclude $q \notin \mathcal{V}$ with probability 1. If $q \notin \mathcal{V}$ yet the membership test returns 1, we call it a "false positive" event. Given a BF set up for $\mathcal{V}$, and $q \notin \mathcal{V}$, the false positive probability for q over a uniformly random choice of $\{H_i(q)\}_{1 \leq i \leq k}$ is

$$P_e \leq (1 - \mathrm{e}^{-k \cdot N/m})^k. \tag{2}$$

Parameters are chosen such that P_e is negligible. It can be seen from (2) that there is a trade-off between k and the probability of having a false positive: smaller k are preferred since they reduce the computation cost, but it is also necessary to maintain the desired false positive rate. Given N, P_e, the optimal choice of k is $k \approx \log_2(1/P_e)$, while the required $m \approx 1.44 \cdot \log_2(1/P_e) \cdot N$ (i.e. $k \approx 20$, $m \approx 29 \cdot N$ for $P_e = 10^{-6}$) [8].

2.4 Hidden Vector Encryption and its Security

Predicate encryption offers a new cryptographic mechanism that provides fine-grained access control over an encrypted database. In predicate encryption, decryption keys are associated with boolean predicates $f : \Sigma \longrightarrow \{0, 1\}$ over a pre-defined set of attributes Σ, while each ciphertext is associated with an attribute $I \in \Sigma$, and a payload messages $\mu \in \mathcal{M}$. A decryption key can be used to decrypt a ciphertext only if the attribute I satisfies $f(I) = 1$. A major application of this encryption paradigm is to outsource encrypted data to a server, and yet retain the ability to make queries on the data without revealing more information than absolutely necessary. This is, in principle, similar to the concept of SSE described in the previous subsection. Hidden vector encryption (HVE) is one such predicate encryption scheme that supports conjunctive, equality, comparison, and subset queries on encrypted data. While HVE was formally defined in the public-key setting in [6], we adopt their definition to the symmetric-key setting in order for it to be applicable in the context of SSE.

A symmetric-key HVE may be defined as an ensemble of the following four PPT algorithms:

- HVE.Setup(λ): takes a security parameter λ and outputs a master secret key msk. It also defines the message space $\mathcal{M}$.
- HVE.KeyGen($msk, \mathbf{v} \in \Sigma_*^m$): takes a predicate vector $\mathbf{v}$, the master secret key msk and outputs a decryption key $\mathbf{s}$.
- HVE.Enc($msk, \mu \in \mathcal{M}, \mathbf{x} \in \Sigma^m$): takes as input a message μ, an index vector $\mathbf{x}$, and the master secret key msk and outputs the ciphertext $\mathbf{c}$ associated with $(\mathbf{x}, \mu)$.
- HVE.Query($\mathbf{s}, \mathbf{c}$): takes a ciphertext $\mathbf{c}$ corresponding to the index vector $\mathbf{x}$ and a decryption key $\mathbf{s}$ corresponding to the predicate vector $\mathbf{v}$, and outputs the message μ if $P^{\mathsf{HVE}}_{\mathbf{v}}(\mathbf{x}) = 1$.

We say that a symmetric-key HVE is correct [6] if for all security parameters λ, all $(\mu, \mathbf{x}) \in \mathcal{M} \times \Sigma^m$ and all predicate vectors $\mathbf{v}$, after sequentially running HVE.Setup(λ) to get msk, HVE.KeyGen($msk, \mathbf{v} \in \Sigma_*^m$) to get $\mathbf{s}$, and HVE.Enc($msk, \mu \in \mathcal{M}, \mathbf{x} \in \Sigma^m$) to get $\mathbf{c}$, if $P^{\mathsf{HVE}}_{\mathbf{v}}(\mathbf{x}) = 1$, then HVE.Query($\mathbf{s}, \mathbf{c}$) = μ, otherwise

$$\Pr[\mathsf{HVE.Query}(\mathbf{s}, \mathbf{c}) = \perp] = 1 - \mathsf{negl}(\lambda).$$

The next step is to formally define the notion of semantic security for symmetric-key HVE against PPT adversaries. The definitions are presented in the simulation-setting, which naturally subsumes the traditional security definitions for HVE in the indistinguishability setting. Prior to presenting the formal definition, we present two auxiliary definitions that constitute the trivial leakage from any symmetric-key HVE scheme. Given a predicate vector $\mathbf{v} = (v_1, \ldots, v_m) \in \Sigma_*^m$, its *wildcard pattern* $\alpha(\mathbf{v})$ is a vector of the same size as the predicate vector $\mathbf{v}$, which is 1 if $v_j \neq *$, and 0 otherwise. Also, given a predicate vector $\mathbf{v} \in \Sigma_*^m$ and an index vector $\mathbf{x} \in \Sigma$, the decryption pattern $\beta(\mathbf{v}, \mathbf{x})$ is a boolean value such that $\beta(\mathbf{v}, \mathbf{x}) = 1$ if $P^{\mathsf{HVE}}_{\mathbf{v}}(\mathbf{x}) = 1$, and 0 otherwise. With these definitions in place, we now define the real and simulation experiments for a symmetric-key HVE scheme.

The Real Experiment. The real experiment for a symmetric-key HVE scheme involves a challenger and a stateful PPT adversary $\mathcal{A}$, who interact as follows:

- **Setup phase:** During this phase the adversary $\mathcal{A}$ chooses an attribute vector $\mathbf{x} \in \Sigma^m$ and gives it to the challenger. In response, the challenger runs HVE.Setup(λ) and outputs *msk* and message space $\mathcal{M}$. $\mathcal{M}$ is given to the adversary.
- **Query phase 1:** The adversary $\mathcal{A}$ adaptively chooses predicate vectors $\mathbf{v}_j$, for $j \in [q_1]$. The challenger then uses $\mathbf{v}_j$ and *msk* to run HVE.KeyGen, and responds with the corresponding decryption keys $\mathbf{s}_j$.
- **Challenge phase:** The adversary $\mathcal{A}$ outputs a message $\mu \in \mathcal{M}$. The challenger runs HVE.Enc using *msk*, $\mathbf{x}$ and μ and obtains the ciphertext $\mathbf{c}$, which is given to $\mathcal{A}$.
- **Query phase 2:** The adversary runs a protocol identical to **Query phase 1** and receives $\mathbf{s}_j$ for $q_1 + 1 \leq j \leq q_2$.

Let $r_{\mathcal{A}}$ denote the internal random bits used by $\mathcal{A}$ during the real experiment. We use the term $\text{View}_{\mathcal{A},\text{Real}}$ to denote the ensemble $(\mathcal{M}, \mathbf{c}, \{\mathbf{v}_j\}_{j\in[q_2]}, r_{\mathcal{A}})$, which is essentially the view of the adversary $\mathcal{A}$ in the real experiment described above.

The Simulation Experiment. The simulation experiment for a symmetric-key HVE scheme involves a stateful PPT simulator $\mathcal{S}$ and a stateful PPT adversary $\mathcal{A}$, who interact as follows:

- **Setup phase:** During this phase the adversary $\mathcal{A}$ chooses an attribute vector $\mathbf{x} \in \Sigma^m$. In response, the simulator $\mathcal{S}$ provides $\mathcal{A}$ with the message space $\mathcal{M}$. Note that $\mathcal{S}$ does not receive the actual attribute vector $\mathbf{x}$.
- **Query phase 1:** The adversary adaptively chooses predicate vectors $\mathbf{v}_j$, for $j \in [q_1]$. For each such predicate, the simulator $\mathcal{S}$ only receives as input the wildcard pattern $\alpha(\mathbf{v}_j)$ and the decryption pattern $\beta(\mathbf{v}_j, \mathbf{x})$. It responds with the corresponding decryption keys $\mathbf{s}_j$.
- **Challenge phase:** The adversary $\mathcal{A}$ outputs a message $\mu \in \mathcal{M}$. The simulator $\mathcal{S}$ responds with the challenge ciphertext $\mathbf{c}$ corresponding to $\mathbf{x}, \mu$.
- **Query phase 2:** The adversary runs a protocol identical to **Query phase 1** and receives $\mathbf{s}_j$ for $q_1 + 1 \leq j \leq q_2$.

Again, let $r_{\mathcal{A}}$ denote the internal random bits used by $\mathcal{A}$ during the simulation experiment. We use the term $\text{View}_{\mathcal{A},\text{Sim}}$ to denote the ensemble $(\mathcal{M}, \mathbf{c}, \{\mathbf{v}_j\}_{j\in[q_2]}, r_{\mathcal{A}})$, which is essentially the view of the adversary $\mathcal{A}$ in the simulation experiment described above.

We define the advantage of a PPT distinguisher $\mathcal{D}$ in distinguishing the real and simulation experiments described above as $\text{Adv}^{\text{HVE}}_{\mathcal{D},\mathcal{A}}(\lambda) = |\Pr[\mathcal{D}(\text{View}_{\mathcal{A},\text{Real}}) = 1] - \Pr[\mathcal{D}(\text{View}_{\mathcal{A},\text{Sim}}) = 1]|$. A symmetric-key HVE scheme is said to be selectively simulation-secure if for all such PPT distinguishers $\mathcal{D}$ and PPT adversaries $\mathcal{A}$, the function $\text{Adv}^{\text{HVE}}_{\mathcal{D},\mathcal{A}}(\lambda)$ is a negligible in λ.

3 LIGHTWEIGHT SYMMETRIC-KEY HIDDEN VECTOR ENCRYPTION

In this section, we propose a novel HVE scheme in the symmetric-key setting, referred to as SHVE, that entirely avoids the use of pairings. Our construction is predicate-only (implying that the payload message is "True" by default) and is amenable to parallel implementations for high performance. At the same time, it guarantees selective simulation-security against probabilistic polynomial-time adversaries for a single ciphertext query and an unbounded number of decryption key queries.

3.1 Detailed SHVE Construction

We now present the details of our proposed SHVE construction. Let Σ be a finite set of attributes and $*$ be a wildcard symbol ("don't care" value) not in Σ. Define $\Sigma_* = \Sigma \cup \{*\}$. In our framework, Σ is typically a finite field $\mathbb{Z}_p$, where p is a prime. We define a family of predicates $\mathcal{P}^{\text{SHVE}} : \Sigma^m \longrightarrow \{0,1\}$ as follows. For each $\mathbf{v} = (v_1, \ldots, v_m) \in \Sigma_*^m$, there exists a predicate $P^{\text{SHVE}}_{\mathbf{v}} \in \mathcal{P}^{\text{SHVE}}$, such that for $\mathbf{x} = (x_1, \ldots, x_m) \in \Sigma^m$, we have:

$$P^{\text{SHVE}}_{\mathbf{v}}(\mathbf{x}) = \begin{cases} 1 & \forall\, 1 \leq i \leq m\ (v_i = x_i \text{ or } v_i = *), \\ 0 & \text{otherwise.} \end{cases}$$

In other words, the vector $\mathbf{x}$ matches $\mathbf{v}$ in all the coordinates that are not the wildcard character $*$. The parameter m is referred to as the *width* of the SHVE.

Our construction uses a pseudorandom function (PRF) $F_0 : \{0,1\}^{\lambda} \times \{0,1\}^{\lambda+\log\lambda} \longrightarrow \{0,1\}^{\lambda+\log\lambda}$ and an IND-CPA secure symmetric encryption scheme (Sym.Enc, Sym.Dec) with both the key-space and the plaintext-space being $\{0,1\}^{\lambda+\log\lambda}$, where λ is a security parameter. The details of the construction are as follows:

- SHVE.Setup(1^{λ}): On input the security parameter λ, the algorithm uniformly samples $msk \xleftarrow{\$} \{0,1\}^{\lambda}$. It then defines the payload message space $\mathcal{M} = \{\text{'True'}\}$, and outputs $(msk, \mathcal{M})$.
- SHVE.KeyGen($msk, \mathbf{v} \in \Sigma_*^m$): On input a predicate vector $\mathbf{v} = (v_1, \ldots, v_m)$ and the master secret key *msk*, we denote by $S = \{l_j \in [m] \mid v_{l_j} \neq *\}$ the set of all locations in $\mathbf{v}$ that do not contain wildcard characters. Let these locations be $l_1 < l_2 < \ldots < l_{|S|}$. The algorithm samples $K \xleftarrow{\$} \{0,1\}^{\lambda+\log\lambda}$ and sets the following:

$$\begin{aligned} d_0 &= \oplus_{j\in[|S|]} \left(F_0(msk, v_{l_j}||l_j)\right) \oplus K, \\ d_1 &= \text{Sym.Enc}\left(K, 0^{\lambda+\log\lambda}\right). \end{aligned}$$

  The algorithm finally outputs the decryption key:

$$\mathbf{s} = (d_0, d_1, S).$$

- SHVE.Enc($msk, \mu = \text{'True'}, \mathbf{x} \in \Sigma^m$): On input a message μ, an index vector $\mathbf{x} = (x_1, \ldots, x_m)$ and the master secret key *msk*, this algorithm sets $c_l = F_0(msk, x_l||l)$, for each $l \in [m]$, and outputs the ciphertext:

$$\mathbf{c} = (\{c_l\}_{l\in[m]}).$$

- SHVE.Query($\mathbf{s}, \mathbf{c}$): The query algorithm takes as input a ciphertext $\mathbf{c}$ and a decryption key $\mathbf{s}$, and parses them as:

$$\begin{aligned} \mathbf{c} &= \left(\{c_l\}_{l\in[m]}\right), \\ \mathbf{s} &= (d_0, d_1, S). \end{aligned}$$

  where $S = \{l_1, l_2, \ldots, l_{|S|}\}$. The algorithm computes the following:

$$K' = \left(\oplus_{j\in[|S|]} c_{l_j}\right) \oplus d_0.$$

Next the decryption algorithm computes:

$$\mu' = \text{Sym.Dec}\left(K', d_1\right).$$

If $\mu' = 0^{\lambda+\log\lambda}$, the decryption algorithm outputs 'True' else it outputs $\perp$.

The correctness of the aforementioned scheme may be verified as follows. Let $\mathbf{c} = (\{c_l\}_{l\in[m]})$ be a ciphertext corresponding to an index vector $\mathbf{x} = (x_1, \ldots, x_m)$, and let $\mathbf{s} = (d_0, d_1, S)$ be a decryption key corresponding to predicate vector $\mathbf{v} = (v_1, \ldots, v_m)$. Let $S = \{l_1, l_2, \ldots, l_{|S|}\}$. We consider the following scenarios:

- If $P_{\mathbf{v}}^{\mathsf{SHVE}}(\mathbf{x}) = 1$, we must have $v_{l_j} = x_{l_j}$ for each $j \in [|S|]$. In other words, we have $c_{l_j} = F_0(msk, v_{l_j}||l_j)$ for each $j \in [|S|]$. This now immediately leads to the following relation:

$$\begin{aligned} K' &= \left(\oplus_{j\in[|S|]} c_{l_j}\right) \oplus d_0 = K, \\ \mu' &= \text{Sym.Dec}\,(K, d_1) \\ &= 0^{\lambda+\log\lambda}. \end{aligned}$$

- If $P_{\mathbf{v}}^{\mathsf{SHVE}}(\mathbf{x}) = 0$, we must have $v_{l_j} \neq x_{l_j}$, for some $j \in [|S|]$. This in turn implies that for some $j \in [|S|]$, $c_{l_j} \neq F_0(msk, v_{l_j}||l_j)$, and hence, during decryption, $K' \neq K$. This ensures that except with negligible probability, we have $\mu' \neq 0^{\lambda+\log\lambda}$, and the decryption algorithm returns the failure symbol $\perp$.

This establishes the correctness of the SHVE scheme. Quite evidently, in our construction, the key-generation and query algorithms operate only on the secret-key/ciphertext components listed in the subset S, which correspond to the non-wildcard entries in the predicate vector. The speed-up achieved as a result of this property is particularly evident in applications where a majority of the predicate vectors have only sparsely distributed non-wildcard entries. As it turns out, our SSE construction, presented in the following section, presents precisely such an application scenario.

3.2 Security of SHVE

We now state the following theorem for the security of our SHVE construction:

Theorem 2. *Our predicate-only* SHVE *construction is selectively simulation-secure in the ideal cipher model as per the security definitions presented in Sec. 2.4.*

Proof. The proof is given in Appendix A. □

4 HXT CONSTRUCTION

We now give the main construction of our SSE protocol and then prove its correctness and analyse its security.

4.1 Hidden Cross Tags (HXT) Protocol

Our SSE protocol HXT uses (for security parameter λ) (*i*) a cyclic group $\mathbb{G}$ with prime order p and generator g, for which the DDH assumption holds, (*ii*) a symmetric-key HVE (see Sec. 2.4), for which we use (for the rest of the paper) SHVE presented in previous section, (*iii*) a symmetric key encryption scheme Sym with key space $\{0,1\}^\lambda$, (*iv*) a Bloom filter BF with length m and k hash functions $\{H_j\}_{1\le j\le k}$ (see Sec. 2.3), and finally (*v*) PRFs F with range $\{0,1\}^\lambda$ and F_p with range $\mathbb{Z}_p^*$.

Algorithm 1 : SE.EDBSetup

Input: 1^λ, DB
Output: *mk*, *param*, EDB

function SE.EDBSetup(1^λ, DB)
  Initialise $\mathbf{T} \leftarrow \emptyset$ indexed by keywords W.
  Select key κ_S for PRF F.
  Select keys κ_I, κ_Z, κ_X for PRF F_p.
  Select hash functions $\{H_j\}_{1\le j\le k}$ for BF.
  Run HVE.Setup(1^λ) to get *msk*.
  Initialise EDB$\leftarrow$ {}.
  for $w \in$ W **do**
    Initialise $\mathbf{t} \leftarrow$ {}.
    Compute $\kappa_e \leftarrow F(\kappa_S, w)$.
    for id $\in$ DB(w) **do**
      Set a counter $c \leftarrow 1$.
      Compute xid $\leftarrow F_p(\kappa_I, \text{id})$.
      Compute $z_w \leftarrow F_p(\kappa_Z, w||c)$; $y_c \leftarrow \text{xid} \cdot z_w^{-1}$.
      Compute $e_c \leftarrow$ Sym.Enc(κ_e, id).
      Append (y_c, e_c) to $\mathbf{t}$ and set $c \leftarrow c+1$.
    end for
    Set $\mathbf{T}[w] \leftarrow \mathbf{t}$.
  end for
  Compute (TSet, κ_T) $\leftarrow$ TSet.Setup($\mathbf{T}$).
  Let EDB(1) = TSet.
  Initialize BF $\leftarrow 0^m$.
  for $w \in$ W **do**
    for id $\in$ DB(w) **do**
      Compute xid $\leftarrow F_p(\kappa_I, \text{id})$.
      for $j = 1 : k$ **do**
        Let $h_j(\text{id}, w) \triangleq H_j\left(g^{F_p(\kappa_X, w)\cdot\text{xid}}\right)$.
        Set BF$[h_j(\text{id}, w)] \leftarrow 1$.
      end for
    end for
  end for
  Compute $\mathbf{c} \leftarrow$ HVE.Enc(*msk*, μ = 'True', BF).
  Let EDB(2) = $\mathbf{c}$.
  return *param* = $(\{H_j\}_{j=1}^k)$, *mk* = $(msk, \kappa_S, \kappa_I, \kappa_Z, \kappa_X, \kappa_T)$, EDB = (EDB(1), EDB(2)).
end function

The HXT protocol consists of two algorithms: SE.EDBSetup and SE.Search.

The setup algorithm SE.EDBSetup (Algorithm 1) gets the security parameter λ and DB and returns the *param*, *mk* and EDB. The encrypted database EDB has two components: EDB(1) is TSet generated exactly as in OXT, and EDB(2), which is shown as the blue part in Fig. 1: it is an HVE encryption of a carefully designed Bloom filter BF, which is set up for XSet elements of the form $h(\text{id}, w) = g^{F_p(\kappa_X, w)\cdot\text{xid}}$, for encrypted identifiers xid $= F_p(\kappa_I, \text{id})$ over all id $\in$ DB(w). The setup algorithm writes 1's into BF in positions in set

$$S = \{H_j(h(\text{id}, w))\}_{1\le j\le k},$$

Algorithm 2 : SE.Search

Input: $param$, mk, query $\bar{w} = (w_1 \wedge \cdots \wedge w_n)$ with sterm w_1, EDB.
Output: Result R.

1: **function** SE.Search($param, mk, \bar{w}$, EDB)
2: Client's inputs are $(param, mk, \bar{w})$ and server's input is $(param$, EDB).
3: Client initialises $R \leftarrow \{\}$.
4: Client computes stag $\leftarrow$ TSet.GetTag(κ_T, w_1) and sends stag to the server.
5: Server lets TSet = EDB(1).
6: Server computes $\mathbf{t} \leftarrow$ TSet.Retrieve(TSet, stag), sends $|\mathbf{t}|$ to client, and starts accepting xtokens computed by client as follows:
7: **for** $c = 1 : |\mathbf{t}|$ **do**
8: Client computes $\eta_{w_1} \leftarrow F_p(\kappa_Z, w_1||c)$.
9: **for** $\ell = 2 : n$ **do**
10: Client computes $\text{xtoken}[c, \ell] \leftarrow g^{\eta_{w_1} \cdot F_p(\kappa_X, w_\ell)}$.
11: **end for**
12: Client sets $\text{xtoken}[c] \leftarrow (\text{xtoken}[c, 2], \ldots, \text{xtoken}[c, n])$.
13: Client sends $\text{xtoken}[c]$ to server.
14: **end for**
15: Server initialises $\mathcal{E} \leftarrow \{\}$.
16: **for** $c = 1 : |\mathbf{t}|$ **do**
17: Client initialises $\mathbf{v}_c \leftarrow *^m$.
18: Server recovers (y_c, e_c) from the c-th component of $\mathbf{t}$.
19: **for** $\ell = 2 : n$ **do**
20: Server computes $\text{xtag} = \text{xtoken}[c, \ell]^{y_c}$.
21: **for** $j = 1 : k$ **do**
22: Server computes $u_j \leftarrow H_j(\text{xtag})$.
23: Server sends u_j to client.
24: Client sets $\mathbf{v}_c[u_j] \leftarrow 1$.
25: **end for**
26: **end for**
27: Client computes $\text{token}_c \leftarrow \text{HVE.KeyGen}(msk, \mathbf{v}_c)$.
28: Client sends token_c to server.
29: Server lets $\mathbf{c} = \text{EDB}(2)$.
30: Server computes $\text{res}_c \leftarrow \text{HVE.Query}(\text{token}_c, \mathbf{c})$.
31: **if** $\text{res}_c =$ 'True' **then**
32: Server adds e_c to $\mathcal{E}$ (i.e., $\mathcal{E} = \mathcal{E} \cup \{e_c\}$)
33: **end if**
34: **end for**
35: Server sends $\mathcal{E}$ to client.
36: Client computes $\kappa_e \leftarrow F(\kappa_S, w_1)$,
37: Client computes $\text{id}_c \leftarrow \text{Sym.Dec}(\kappa_e, e_c)$, and adds id_c to R for all $e_c \in \mathcal{E}$.
38: **return** R
39: **end function**

over all (id, w) pairs with id $\in$ DB(w), and then encrypts BF with HVE.Enc. The new parts of our protocol compared to OXT are coloured blue.

The search protocol SE.Search is shown in Algorithm 2, where the first 14 lines generate stag and xtokens similar to OXT. The XSet membership test for conjunctions in OXT is replaced by a HVE token generation and query. Namely, the HVE token token_c for

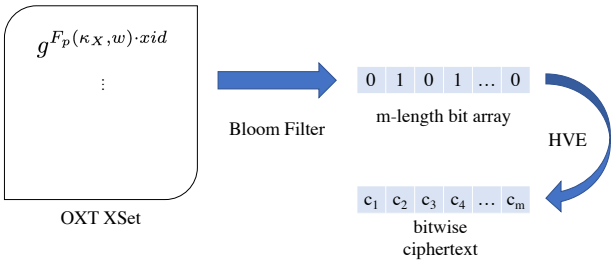

Figure 1: An illustration of difference w.r.t XSet **in** HXT **compare to** OXT.

all $\text{id}_c \in \text{DB}(w_1)$ is generated for a predicate (BF) vector $\mathbf{v}_c$ with 1's in positions in set $S' = \{H_j(h(\text{id}_c, w_i))\}_{1 \le j \le k}^{2 \le i \le n}$ and wildcards in other positions. [1] Consequently (as the message encrypted by HVE was set to 'True' in SE.EDBSetup) the HVE.Query returns 'True' if $S' \subseteq S$, i.e. if *all* $n-1$ xterms w_i are in the document id_c. Otherwise, HVE.Query returns $\perp$, without revealing KPRP information on which w_i are in id_c. Importantly, in step 27 and 30, the HVE.KenGen and HVE.Query algorithm only uses components of $\mathbf{c}$ in the non-wildcard positions of $\mathbf{v}_c$ and token_c, so search runtime is only proportional to $|\text{DB}(w_1)| \cdot n \cdot k$ (similar to OXT), and not to the size m of the BF. We next show that HXT is correct with the Bloom filter's false positive rate P_e.

Theorem 3. *If the underlying* HVE *scheme and T-set scheme* Γ *are correct, and the* PRFs F *and* F_p *are secure, then* HXT *is computationally correct with false positive rate* $P_e \le (1 - e^{-k \cdot N/m})^k$.

Proof. The proof is given in the full version. □

5 SECURITY

A Searchable Symmetric Encryption (SSE) query consists of a Boolean formula ψ and a tuple $\mathbf{w} = (w_1, \ldots, w_n)$ of keywords. Throughout the paper, we only consider conjunctive queries with $\psi(\mathbf{w}) = w_1 \wedge \cdots \wedge w_n$. Without loss of generality, we assume that w_1 is sterm and $(w_2, \ldots, w_n)$ are $n-1$ xterms. For a vector of queries $\mathbf{q} = (\text{s}, \text{x}_2, \ldots, \text{x}_n)$, it consists of a vector s of sterms, and a sequence of vectors $\text{x}_2, \ldots, \text{x}_n$ of xterms.

We define the leakage function of HXT $\mathcal{L}(\text{DB}, \mathbf{q})$ as a tuple(N, EP, SP, WRP, IP) formed as follows:

- $N = \sum_{i=1}^{d} |\text{W}_i|$ is the total number of appearances of keywords in documents.
- EP is the equality pattern of $\text{s} \in \text{W}^Q$ indicating which queries have the equal sterms. In particular, $\text{EP}[i] = |\{\text{s}[1], \ldots, \text{s}[j]\}|$, where j is the least index for which $\text{s}[j] = \text{s}[i]$. Note that $\text{EP} \in [\omega]^Q$ and it is leaked since the client sends stag corresponding to s to server.
- SP is the size pattern of the queries, i.e. the number of documents matching the sterm in each query. Formally, $\text{SP} \in [\omega]^Q$ and $\text{SP}[i] = |\text{DB}(\text{s}[i])|$.

[1] Under the "honest-but-curious" assumption, the server following the protocol can not arbitrarily check the membership of Bloom Filter. Therefore, the server is not able to check XSet like in OXT.

- WRP is the whole result pattern, which is an array computed as follows: $\mathsf{WRP}[i] = \mathsf{DB}(\mathsf{s}[i]) \cap_{j=2}^{n} \mathsf{DB}(\mathsf{x}_j[i])$[2].
- IP is the conditional intersection pattern, which is a 4-dimensional table $\mathsf{IP}[i, j, \alpha, \beta]$ defined as follows:

$$= \begin{cases} \mathsf{DB}(\mathsf{s}[i]) \cap \mathsf{DB}(\mathsf{s}[j]) & \text{if } i \neq j, \alpha \neq \beta, \\ & \text{and } \mathsf{x}_\alpha[i] = \mathsf{x}_\beta[j], \\ \emptyset & \text{otherwise,} \end{cases}$$

for $1 \leq i, j \leq Q$ and $1 \leq \alpha, \beta \leq n$.

The overall leakage function $\mathcal{L}$ consists of the leakage from the HXT protocol $\mathcal{L}_{\mathsf{HXT}}$ and the leakage function of T-set $\mathcal{L}_{\mathbf{T}}$.

5.1 Leakage Function Comparison

Note that WRP is a new component in our leakage function compared to OXT. In their leakage function, they actually had $\mathsf{KPRP}[i] = \cup_{j=2}^{n} \left(\mathsf{DB}(\mathsf{s}[i]) \cap \mathsf{DB}(\mathsf{x}_j[i])\right)$, which is denoted by RP in [12] and obviously a lot more than WRP. The following example illustrates the impact of updating leakage component from KPRP to WRP.

Suppose that a database consists of 6 documents labelled by $\{\mathsf{id}_i\}_{1 \leq i \leq 6}$. Let us assume the following database 'forward index', listing document id's and keywords contained in each:

id	keywords	id	keywords
1	w_1, w_2, w_6, w_7, w_8	4	w_1, w_2, w_3
2	w_2, w_3, w_4, w_5	5	w_1, w_3, w_6
3	w_4, w_5, w_6, w_7	6	w_2, w_3, w_7

Consider the query $w_1 \wedge w_2 \wedge w_3$ for some keywords w_1, w_2, and w_3. By convention, we let w_1 be the least frequent keyword amongst all queried words. The inverted-index listing the document id's containing each of the queried words are $\mathsf{DB}(w_1) = \{\underline{\mathsf{id}_1}, \underline{\mathsf{id}_4}, \underline{\mathsf{id}_5}\}$, $\mathsf{DB}(w_2) = \{\underline{\mathsf{id}_1}, \mathsf{id}_2, \underline{\mathsf{id}_4}, \mathsf{id}_6\}$, and $\mathsf{DB}(w_3) = \{\mathsf{id}_2, \underline{\mathsf{id}_4}, \underline{\mathsf{id}_5}, \mathsf{id}_6\}$.

Table 2: Leakage Comparison for query $w_1 \wedge w_2 \wedge w_3$ between KPRP and WRP.

Leakage Component	Leaked Entries
KPRP (from OXT)	$\{(\mathsf{id}_1, w_2), (\mathsf{id}_4, w_2), (\mathsf{id}_4, w_3), (\mathsf{id}_5, w_3)\}$
WRP (from HXT)	$\{(\mathsf{id}_4, w_2), (\mathsf{id}_4, w_3)\}$

The KPRP leakage component in OXT is computed as follows: $\mathsf{RP} = \cup_{j=2}^{3} \left(\mathsf{DB}(w_1) \cap \mathsf{DB}(w_j)\right) = \{\mathsf{id}_1, \mathsf{id}_4\} \cup \{\mathsf{id}_4, \mathsf{id}_5\}$. As shown in Table 2, The KPRP leakage thus reveals 4 entries of the inverted-index, underlined in the inverted-index above. However, in this paper, we eliminate the 'partial query' (KPRP) leakage induced by RP, leaving only *whole result pattern* (WRP) corresponding to the final query result. By definition, we have $\mathsf{WRP} = \cap_{j=1}^{3} \mathsf{DB}(w_j)$. In our example, WRP reveals the exact result of the query, that is $\{\mathsf{id}_4\}$. Table 2 shows that the WRP only reveals two entries in the inverted-index above, in contrast to 4 entries revealed by KPRP above.

In fact, [12] has not noticed this leakage in their analysis and simply put RP to be the set of all identifiers matching the i-th query.

[2]WRP is defined under the "no false positive" assumption, i.e. Both TSet and Bloom Filter do not have a false positive rate. Otherwise, the false positive should be taking into account to define a refined leakage profile FP-WRP.

5.2 Security Analysis of HXT

Here, we show the security of our protocol against both a non-adaptive and an adaptive adversarial server which is assumed to be "honest-but-curious". Similar to [12], we first give a theorem about the security against non-adaptive attacks, and later discuss the full security. For sake of simplicity, we assume in our proof that no false positive happens in our protocol HXT[3] (i.e., no false positive happening in both TSet and BF).

Theorem 4. *Our protocol* HXT *is $\mathcal{L}$-semantically secure against non-adaptive attacks where $\mathcal{L}$ is the leakage function defined as before, assuming that the* DDH *assumption holds in $\mathbb{G}$, that F and F_p are secure* PRF*s, that* HVE *is a selectively simulation-secure protocol, that* Sym = (Sym.Enc, Sym.Dec) *is an* IND-CPA *secure symmetric encryption scheme, that Γ is a $\mathcal{L}_{\mathbf{T}}$-secure and computationally correct T-set instantiation, and that no false positive happens in our* HXT.

Proof. The proof is given in Appendix A. □

We now show that our theorem is also valid for adaptive models.

Theorem 5. *Our protocol* HXT *is $\mathcal{L}$-semantically secure against adaptive attacks where $\mathcal{L}$ is the leakage function defined as before, assuming that the* DDH *assumption holds in $\mathbb{G}$, that F and F_p are secure* PRF*s, that* HVE *is a selectively simulation-secure scheme, that* Sym *is an* IND-CPA *secure symmetric encryption scheme, and that Γ is a $\mathcal{L}_{\mathbf{T}}$-secure and computationally correct T-set instantiation.*

Proof. The proof is given in Appendix A. □

6 PERFORMANCE COMPARISON

We first give a list of notations needed in this section for our comparison analysis in Table 3.

Table 3: Notations for Comparison Analysis

Notation	Meaning
m, m_p	number of multiplications over $\mathbb{G}$ and $\mathbb{Z}_p$
p, e	number of pairings and exponentiations
$\mathsf{e}_{\mathsf{pre}}$	number of preprocessed exponentiations
G, G_T, Z_p	size of an element from $\mathbb{G}$, $\mathbb{G}_T$, and $\mathbb{Z}_p$ resp.
m'	number of non-wildcard elements in a BF
T_{PRF}	time taken to compute a PRF
T_{hash}	time taken to compute a hash of BF
T_{XOR}	time taken to perform an exclusive-or operation over λ
T_{Enc}	time taken to compute a sym. ciphertext
T_{Dec}	time taken to decrypt a sym. ciphertext
T_{TSet}	time taken to set-up TSet

6.1 Comparison between HVE Schemes

The performance of the proposed HXT protocol depends on the parameters of the underlying employed HVE scheme. Hence, we first give the comparison between the available HVE schemes and our SHVE scheme in terms of their performance parameters. After

[3]Note that the assumption can be relaxed by taking into account the indices introduced due to the false positive.

Table 4: Different HVE schemes and their properties.

Ref.	$\lvert\mathbb{G}\rvert$	Ciphertext Size	Key Size	Enc. Cost	Query Cost	KeyGen. Cost
BW [6]	p_1p_2	$(2m+1)\mathsf{G}+(1)\mathsf{G}_\mathsf{T}$	$(2m+1)\mathsf{G}$	$(6m+2)\mathsf{m}+(8m+2)\mathsf{e}$	$(2m+1)\mathsf{p}$	$(2m+1)\mathsf{p}+(2)\mathsf{m}$
KSW [25]	$p_1p_2p_3$	$2(2m+1)\mathsf{G}+(1)\mathsf{G}_\mathsf{T}$	$(2m+1)\mathsf{G}$	$(4m)\mathsf{m}+(2m)\mathsf{m}_p$ $+(6m+1)\mathsf{e}$	$(2m+1)\mathsf{p}$	$(3m+1)\mathsf{m}+(2m)\mathsf{m}_p$ $+(6m+2)\mathsf{e}$
OT [30]	p_1	$2(5m+1)\mathsf{G}+(1)\mathsf{G}_\mathsf{T}$	$(11)\mathsf{G}+(m)\mathsf{Z}_p$	$(m+1)\mathsf{e}+(m+1)\mathsf{m}$	$(11)\mathsf{p}+5(m-1)\mathsf{e}$	$(12m+10)\mathsf{m}$
IP [23]	p_1	$(2m+1)\mathsf{G}+(1)\mathsf{G}_\mathsf{T}$	$(2m)\mathsf{G}$	$(2m+2)\mathsf{e}+(1)\mathsf{m}$	$(2m)\mathsf{p}+(2m+1)\mathsf{m}$	$(2m)\mathsf{e}+(2m)\mathsf{m}$
SHVE	N/A	$(m)\lambda$	$O(m')+2\lambda$	$(m)T_{\mathsf{PRF}}$	$(m')T_{\mathsf{XoR}}+T_{\mathsf{Dec}}$	$(m')T_{\mathsf{PRF}}+(m')T_{\mathsf{XoR}}$ $+T_{\mathsf{Enc}}$

Table 5: Execution Time Comparison between IP [23] and the proposed SHVE; The width of HVE: $m = 10000$, no wildcard element.

Scheme	KeyGen. Time (s)	Enc. Time (s)	Query Time (s)
IP [23]	51.154	50.901	119.219
SHVE	0.172	0.162	0.004

showing the performance advantage of proposed SHVE, we carry on to derive and analyse the performance of our HXT and compare it to that of OXT.

In Table 4, we summarise 4 well-known paring-based HVE constructions as well as the SHVE scheme we proposed and compare them based on their properties including ciphertext and key (token) sizes as well as encryption, query and token generation computational costs when we use them to encrypt a Bloom filter with the length of m, m is also referred to as the width of the HVE here. Note that the second and third schemes were induced from Inner Product Encryption (IPE), while the first and the fourth constructions were originally obtained for HVE model. All the presented schemes except ours are pairing-based constructions (over groups $\mathbb{G}$ and $\mathbb{G}_T$ as domain and range of a bilinear function) with different group orders ranging from just 1 prime to product of 3 primes, while our construction is based on symmetric key encryption.

It is clear that the HVE scheme with pairings in [23] can provide the most efficient construction with low complexity encryption, query, and key generation algorithms amongst the pairing-based HVEs. Therefore, the first evaluation aims to compare the runtime efficiency of our SHVE scheme with the IP [23]. To evaluate our scheme, we implement our SHVE with Java, and we choose to use AES-CMAC as our PRF function while AES in CBC mode as the symmetric key encryption scheme. All above symmetric cryptographic primitives are from the Legion of Bouncy Castle [35]. We adopt 128-bit key length for symmetric key encryption scheme because symmetric encryption with 128-bit key offers better security than the Elliptic Curve Cryptography (ECC) over a curve with a 160-bit prime order group with a smaller key size according to RFC 4492 [3], and, it can perform encryption/decryption efficiently. For comparison, we use the open source implementation of IP [23] included in Java Pairing based Cryptograyphy (JPBC) [9] library, which is also implemented by Java, and it is constructed on the curve $y^2 = x^3 + x$ over the field $\mathbb{F}_p$ for some prime $p = 3 \mod 4$, the group operations are based on and the 160-bit prime order groups which are generated from above curve. To make the performance of IP [23] consistent with the theoretical analysis from Table 4, we add a preprocessing code for Enc., as it has been implemented for KeyGen. and Query, but missed in Enc..

It is also critical to distinguish between m (the length of the entire Bloom filter which is at least as large as $36 \times N$) and m'. In the HVE schemes based on inner product encryption, the complexity of key generation and query depends on m, while in the SHVE, the complexity depends only on the number of non-wildcard characters m' in the vector. This has a significant impact on the practicality of the scheme since a dependence of m would mean a query complexity that scales with the size of the database. However, our query complexity (e.g., number of pairings and exponentiations) grows with the size of the result set rather than the database.

All evaluations are performed on a server with Intel Xeon E5 2660 2.20 GHz CPU and 128 GB of DDR3 RAM. The width of HVE m is set to 10000, and the predicate vector $\mathbf{v}$ we used doesn't have any wildcard element, which means $m = m'$ in our evaluations. We run IP [23] and proposed SHVE scheme three times respectively to obtain the average execution times of key generation, encryption, and query. The results are shown in Table 5. Compared with IP [23], the proposed SHVE scheme is 314x faster in Enc., 297x faster in KeyGen., and almost 30000x faster in Query. We also evaluate the execution time of SHVE with large m to show the efficiency of SHVE, which is given in the full version.

6.2 Comparison between OXT and HXT

We compare our scheme with that of [12] in terms of computational complexity (of the set-up and search phases), storage size (of the server), and the number of interaction runs and bandwidth for conjunctive query $(w_1 \wedge w_2 \wedge \cdots \wedge w_n)$ with $\mathsf{sterm} = w_1$. Note that e, m, and p are defined in Table 3. The overall comparison is summarized in Table 6.

6.2.1 Set-up computational costs. First, we focus on computational cost spent during the set-up phase. Although both our protocol and the OXT share a lot of similarities, the time taken to generate the encrypted database in HXT is mainly contributed from the computation of the $\mathbf{c}$ in addition to TSet, XSet and BF vector computational costs. The computing of $\mathbf{c}$ adds the computational cost of an HVE.Enc, that is $(m)T_{\mathsf{PRF}}$ if we employ the SHVE. If we let $T_{\mathsf{TSet}} = T_{\mathsf{PRF}} + T_{\mathsf{Enc}}$, then computing TSet and XSet in OXT sums up to $NT_{\mathsf{TSet}} + N\mathsf{e}_{\mathsf{pre}}$, where the first term is obtained since TSet has N components each filled up using a PRF and an encryption of a symmetric encryption scheme. For the XSet computation, we

Table 6: Communication overhead between client and server and their computational costs.

Conjunctive query $\mathbf{q} = (w_1 \wedge w_2 \wedge \cdots \wedge w_n)$.			
		OXT [12] cost	HXT cost
Computation	**set-up comp. cost**	$NT_{\mathsf{TSet}} + N\mathsf{e}_{\mathsf{pre}} + NkT_{\mathsf{hash}}$	$NT_{\mathsf{TSet}} + N\mathsf{e}_{\mathsf{pre}} + NkT_{\mathsf{hash}} + (m)T_{\mathsf{PRF}}$
	search common cost (server) xtag comp.&BF match	$\lvert\mathsf{DB}(w_1)\rvert\,((n-1)(kT_{\mathsf{hash}} + \mathsf{e}))$	
	search additional cost (server) HVE Queries	N/A	$\lvert\mathsf{DB}(w_1)\rvert((m')T_{\mathsf{XOR}} + T_{\mathsf{Dec}})$
	search common cost (client) stag, xtoken comp. & index recover	$\lvert\mathsf{DB}(w_1)\rvert(nT_{\mathsf{PRF}} + (n-1)\mathsf{e}_{\mathsf{pre}}) + T_{\mathsf{PRF}} + T_{\mathsf{Dec}}$	
	search additional cost (client) HVE KeyGen	N/A	$\lvert\mathsf{DB}(w_1)\rvert((m')T_{\mathsf{PRF}} + (m')T_{\mathsf{XOR}} + T_{\mathsf{Enc}})$
Storage	**storage size (server)**	$N\lambda + m$	$N\lambda + (m)\lambda$
Communication	**common comm. bandwidth**	$\lvert\mathbf{t}\rvert + \lvert\mathsf{DB}(w_1)\rvert(n-1)\mathsf{G} + \lvert\mathcal{E}\rvert O(\lambda)$	
	additional comm. bandwidth $token_c$ transmission	N/A	$\lvert\mathsf{DB}(w_1)\rvert(O(m') + 2\lambda)$

make N preprocessed exponentiations, in total of $N\mathsf{e}_{\mathsf{pre}}$. These are shown in the first row computation comparison of Table 6. Note that although we do not entirely generate XSet in HXT, we still compute its components to initiate a BF vector. Bloom filter is employed for keeping the XSet in a reasonably storage space for the practical implementation of OXT [12] and generating HVE for HXT, the BF generation has only the evaluations of hashs H_j, for $1 \le j \le k$ and kN elements.

6.2.2 Storage size. We now investigate the storage size of HXT and compare it to OXT. The latter stores TSet and the Bloom filter of XSet in EDB, while HXT uses TSet and an encryption of an HVE system. Note that the latter is in fact an HVE ciphertext replacing the Bloom filter of XSet of OXT. The size of TSet equals N, where each component contains a $\mathbb{G}$ element of size $O(\lambda)$; and the size of Bloom filter is m. Note that m is approximately $1.44kN$ to attain a negligible probability of false positives, which is 35 times smaller than an XSet with 512-bit base field for $k = 20$ (1024 bits for each element and $1024N$ in total). Therefore, This in total gives $O(N(\lambda + k))$ as the storage size in OXT. The size of $\mathbf{c}$ depends on the size of the corresponding BF (used to generate it). Therefore, the ciphertext size is $(m)\lambda$. Hence, the overall storage size of HXT is of order $O(kN\lambda)$.

6.2.3 Search computational costs. During the search phase, the computational costs are divided between the client and the server. The client in both HXT and OXT has to interact with the server once she wants to send stag and xtokens, where she needs to use a preprocessed element to calculate $(n-1)$ exponentiations for each recovered document in $\mathsf{DB}(w_1)$. The overall computational cost till this stage is $T_{\mathsf{PRF}} + |\mathsf{DB}(w_1)|T_{\mathsf{PRF}} + |\mathsf{DB}(w_1)|(n-1)(T_{\mathsf{PRF}} + \mathsf{e}_{\mathsf{pre}}) = T_{\mathsf{PRF}} + |\mathsf{DB}(w_1)|(n)T_{\mathsf{PRF}} + |\mathsf{DB}(w_1)|(n-1)\mathsf{e}_{\mathsf{pre}}$. In OXT, the server then performs $|\mathsf{DB}(w_1)|(n-1)\mathsf{e}$ many xtag generation and $|\mathsf{DB}(w_1)|(n-1)kT_{\mathsf{hash}}$ membership test in the BF vector. [4] It finally returns recovered encrypted indices to the client, when she has to perform a decryption of what was stored in TSet. The latter costs client T_{Dec}. In HXT, the server evaluates vectors $\mathbf{v}_c$, for $1 \le c \le |\mathsf{DB}(w_1)|$, and sends it to client. The client consequently computes token_c using HVE.KeyGen. The first one again is endowed by $(n-1)kT_{\mathsf{hash}}$ and $(n-1)\mathsf{e}$ evaluations, while the second component requires an HVE key generation, if m' denotes the number of non-wildcard components of $\mathbf{v}_c$, then the cost of generating token_c is $(m')T_{\mathsf{PRF}} + (m')T_{\mathsf{XOR}} + T_{\mathsf{Enc}}$. Finally at the server side, the determination of res through an HVE.Query and a token_c is extra compared to OXT, which costs another $(m')T_{\mathsf{XOR}} + T_{\mathsf{Dec}}$, using the above defined notation. The search computational costs discussed above are presented in the second to fifth row of Table 6.

Finally, we determine the computational cost overhead on server and client side, respectively. For this purpose, we define O^{srvr} as:

$$\frac{|\mathsf{DB}(w_1)|((m')T_{\mathsf{XOR}} + T_{\mathsf{Dec}})}{|\mathsf{DB}(w_1)|((n-1)(kT_{\mathsf{hash}} + \mathsf{e}))}.$$

Based on a micro-benchmark, the computation time of a hash is insignificant, as the computation cost of an exponentiation is comparable to 1300 evaluations of hashs. In addition, the computation time of an xor operation is 3 times faster than hash, 50 times faster than Sym.Dec and 100 times faster than Sym.Enc. If we apply the typical settings from [12]: let $P_e = 10^{-6}$ and $n = 2$, we will have $k = 20$, and m' is equal to the size of $\mathbf{v}_c$, which is $(n-1)k$ (k in above settings), we conclude that HXT only introduces 1% extra cost on server side; We further define O^{client} as

$$\frac{|\mathsf{DB}(w_1)|((m')T_{\mathsf{PRF}} + (m')T_{\mathsf{XOR}} + T_{\mathsf{Enc}})}{|\mathsf{DB}(w_1)|(nT_{\mathsf{PRF}} + (n-1)\mathsf{e}_{\mathsf{pre}}) + T_{\mathsf{PRF}} + T_{\mathsf{Dec}}}.$$

Because the preprocessed element performs exponentiation 17 times slower than PRF and PRF is 43 times slower than exclusive-or operation, the overhead of HXT is about 119% on client side. It is easy to deduce from the micro-benchmark that the above two ratios are inversely proportional to n, which means the computational overhead of HXT is smaller when the query has a longer keyword list. However, our evaluations on subsection 7.3 show that such overhead can be masked by I/O cost on server side.

[4] Note that server doesn't preprocess xtokens, because preprocessing takes more time than exponentiation directly, so it only suits in the case when the same element is reused many times.

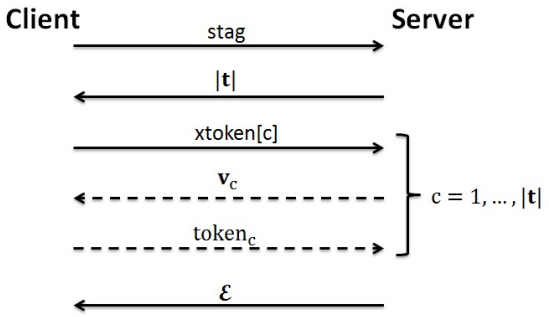

Figure 2: All interactions between a server and a client during a search in HXT (all arrows) and OXT (solid arrows only). Since the message flows corresponding to third, forth, and fifth lines are sent in parallel over $c \in [|\mathbf{t}|]$, the HXT protocol only has 6 message flows (or equivalently 3 rounds). This is in contrast to OXT, which has 4 message flows (2 rounds).

6.2.4 Interaction rounds and bandwidth. In our proposed query protocol SE.Search shown in Algorithm 2 (on page 8), the stag and xtoken generations are the same as that in OXT, which accounts for the first round of interaction between the client and server. The bandwidth of this round is $|\mathbf{t}| + |\text{DB}(w_1)|(n-1)\text{G}$. In the second and third rounds of interactions, the server computes $\mathbf{v}_c$ (using the hashs from the BF) for each encrypted index in vector $\mathbf{t}$ retrieved using stag and TSet and sends it to the client. This interaction costs $|\text{DB}(w_1)|O(m')$ communication overhead. The client then uses the key generation algorithm of the corresponding HVE to form token_c and lets server to use these token_c to check if the result of this query is "True" or not (using the query algorithm of the underlying HVE scheme). This extra interaction round has $|\text{DB}(w_1)|(O(m') + 2\lambda)$ bandwidth. It gathers all encrypted indices e_c that passes the HVE.Query into a set $\mathcal{E}$. This set will be sent to client with bandwidth $|\mathcal{E}|O(\lambda)$ as the final result and the client is further responsible to decrypt the recovered indices using her own secret key to the symmetric encryption Sym. Note that one round of interaction between client and server has been added in our HXT compared to OXT, where the server only checks a set membership against XSet rather than employing HVE. All these are summarized in Fig. 2.

7 EVALUATIONS

7.1 Prototype Implementation

We implement a prototype system for evaluating our HXT protocol. To build this prototype, we firstly implement an OXT prototype, because there is no open source implementation of OXT. Our implementation use the cryptographic primitives outlined in subsection 6.1. Bloom filter is a essential part for both OXT and HXT prototypes, we deploy the Bloom filter from Alexandr Nikitin as it is the fastest Bloom filter implementation for Java [29]. In our OXT prototype, we set the false positive rate to 10^{-6} and it enables the OXT prototype to keep the Bloom filter of XSet on the RAM of our server.

The OXT prototype consists of two main parts: one for encrypted database (EDB) generation and the other for database query. Based on the OXT prototype, we implement the proposed SHVE scheme to enable the prototype to perform HXT protocol. By replacing the Bloom filter of XSet to a SHVE ciphertext in EDB generation, and the Bloom filter query to SHVE key generation and query in database query, the OXT prototype is adapted to an HXT one. All above programs are implemented by a combination of Java and Scala code and it has roughly 2000 lines of code.

The implemented prototypes are designed to fulfil the scalability and the query efficiency characteristics of original protocols. To reach these goals, our prototypes are implemented on distributed platform (i.e. Hadoop [1]). Hadoop is a distributed platform in master-slave structure: It has a master node that manages the resource and monitor the application execution while a group of slave nodes which dedicate their computational resources (e.g. CPU, RAM, disk) to execute the tasks from master node. Hadoop [1] offers a distributed file system HDFS [32], in addition to a distributed database HBase [2]. HDFS allows our prototypes to store TSet and SHVE in multiple hard drives in different slave nodes, and reach them concurrently. As a result, it avoids the heavy I/O overhead on single hard drive; HBase provides efficient in-memory index mechanism over distributed dataset, which can highly reduce the TSet and SHVE access time.

For scalability, we take steps to further improve the read performance of TSet and HVE on HBase. HBase is a column-based database [2], that is, the data in the same column are stored in the same file. Therefore, we follow tall narrow model [20] to design TSet table to avoid a very long column value because such long value incurs extra overhead (i.e. compaction) while loading them into memory. In the tall narrow model, each tuple list $\mathbf{T}[w]$ is split into fixed-size blocks with a stag_w and a block counter. Because HBase stores data in key lexicographically order, it stores above blocks with the same stag_w into contiguous disk area. Hence, retrieving TSet only has one random access following by the sequential accesses. HVE is stored as key/value pairs on HBase. Because the HVE is a vector the ciphertext, we use the index of vector as the key, and the corresponding ciphertext as value. Due to the variety of HVE key, it is difficult to avoid the random access of HVE ciphertext. Therefore, we use the randomised index of HVE ciphertext as the key of HVE, because random keys help to distribute the data into different nodes, which enables the random access in parallel [20].

To accelerate the query phase, we make use of the distributed in-memory computing framework Spark [36]. Spark follows the same data processing flow as MapReduce [16], which distributes the computing tasks and execute them on different slave nodes in parallel. Spark inherits the scalability and fault tolerance of MapReduce [36], but it can execute tasks in-memory without keeping any intermediate data on disk, it means our prototypes don't have any I/O operation during database query except the TSet and SHVE query.

We deploy our prototypes on a shared Hadoop cluster with 13 slave nodes and one master node. Each node has 2x Intel Xeon CPU E5-2660 2.2GHz (each CPU has 8 cores with dual-thread) and 128GB RAM, in addition, we have another node with the same specification above which is served as edge node and client of our prototypes. All nodes are connected by InfiniBand [22] network

Table 7: Statistics of the datasets used in the evaluation

Size	# of documents	Distinct keywords	Distinct (id, w) pairs
2.93GB	$7.8 * 10^5$	$4.0 * 10^6$	$6.2 * 10^7$
8.92GB	$2.7 * 10^6$	$1.0 * 10^7$	$1.6 * 10^8$
60.2GB	$1.6 * 10^7$	$4.3 * 10^7$	$1.4 * 10^9$

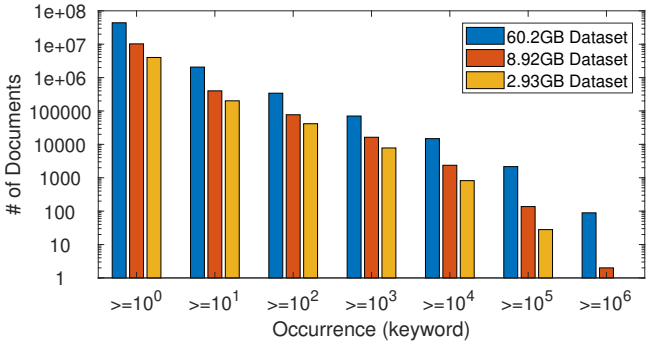

Figure 3: The keyword occurrence distribution of three datasets.

technique. The cluster installs CDH 5.2.6 [14], one of the most complete and popular distribution of Hadoop and its peripheral ecosystem (contains Hadoop Yarn 2.5.0, HBase 0.96.8 and Spark 2.0.2). Based on the setup configuration and scheduling policy, we can use at most 416 virtual cores (32 virtual cores in each slave node) and 1248GB RAM (96GB RAM in each slave node), in addition, each virtual core should have at least 2GB. In real world scenario, 1 virtual core and 2GB RAM are needed for running the monitor program of a distributed application on Hadoop. As a result, our prototypes can start 415 tasks with 1 virtual core and 3GB RAM concurrently at most. However, our following evaluations show that it is not necessary to use all resources to query the database: 100 concurrent tasks with 1 virtual core and 2GB RAM are sufficient to provide a satisfactory result.

7.2 Datasets

We test our implementation on three datasets from Wikimedia Downloads [19]: the original sizes of our datasets are 2.93GB[5], 8.92GB[6] and 60.2GB[7], respectively. A brief summary of the statistical features of the datasets is given in Table 7.

The corresponding EDB and Bloom filter size for above three datasets are 9.3GB and 215MB, 33GB and 575MB, 256GB and 4.76GB. In addition, the HVE size is 28GB, 76GB and 647GB. Fig 3 further gives the frequency of keywords according to the number of documents to depict the keyword occurrence distribution of the generated EDBs.

7.3 Evaluation Results

Our evaluation aims to verify the following: (i) our implementation in distributed platform can ensure the efficiency of queries; (ii) the additional query latency introduced in HXT is small; (iii) HXT keeps the scalability property of OXT.

[5]enwiki-20161220-pages-articles22.xml
[6]enwiki-20161220-pages-articles27.xml
[7]enwiki-20171201-pages-articles.xml

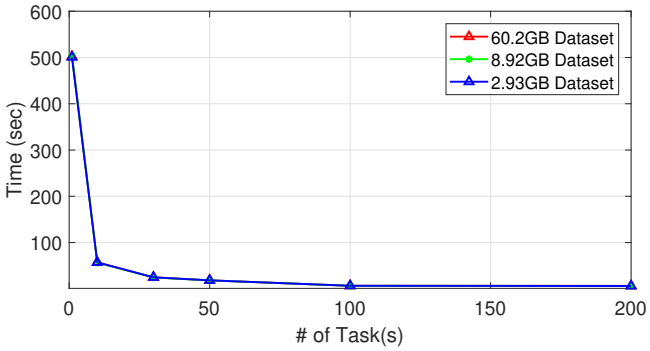

Figure 4: HXT Server query time when # of parallel tasks increases.

7.3.1 The impact of parallelism. First, we study how distributed computing influences the query efficiency of HXT prototype. We choose a keyword with about 330K matched documents respectively in three datasets, and we use the selected keyword as the sterm to perform a two-terms conjunctive query in our HXT prototype. We vary the number of parallel tasks from 1 to 200 before the server start running the query to test the impact of parallelism. As shown in Fig 4, we observe that parallelism successfully improved the efficiency of query by a factor of 100 times on server side. In addition, we conclude the impact in three cases: (i) when increasing the number of parallel tasks from 1 to 10, parallelism can highly improved the server side performance; (ii) when the number of parallel tasks is between 10 to 100, parallelism only can slightly improve the query efficient on server; (iii) after the number of parallel tasks is larger than 100, parallelism doesn't affect the query efficiency.

The reason is that the computational cost is the dominant cost when the server only has a small fraction of resources is allowed to engage the computation. By increasing the parallelism factor in above case, computation cost of each tasks can be highly reduced, which yields a significant performance improvement. However, with the increase of the number of parallel tasks, it incurs more communication cost for task scheduling and monitoring between the master and slave nodes of our cluster, and HBase also has a I/O limits based on the underlying infrastructure. Therefore, the computation cost is overlapped by communication cost and I/O latency after we have more than 100 parallel tasks.

Another observation is the query latency highly depends on the selectivity of sterm, while it is independent of the size of dataset. We examine it deeply in the scalability test at the end of this section.

7.3.2 Performance comparison. We use the parallel factor 100 to further investigate the additional overhead in HXT comparing with OXT protocol. Due to the OXT and HXT protocol having the same behaviour when performing single-keyword search, our evaluation only report the query performance of conjunctive query.

We choose a variable term, named **v**, on the 2.93GB dataset. The selectivity of **v** is from 2 to 337449 documents. We further choose a

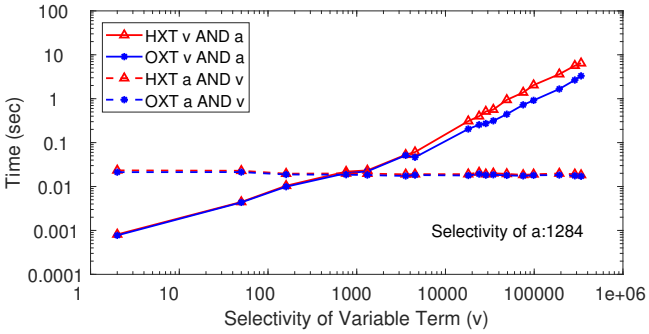

Figure 5: Server performance comparison between HXT and OXT in 2.93GB dataset.

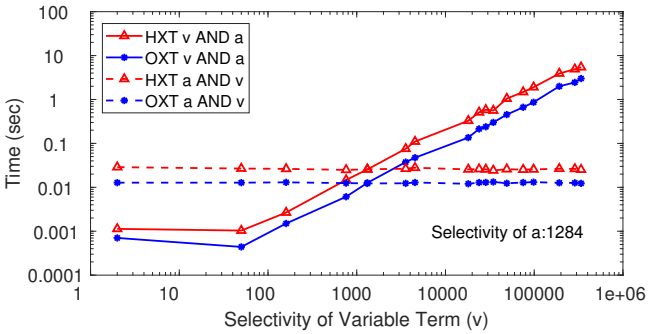

Figure 6: Client performance comparison between HXT and OXT in 2.93GB dataset.

fixed term **a** and perform two types of conjunctive queries on the 2.93GB dataset. Fig 5 shows the time spent by HXT and OXT during the query on server side. The first conjunctive query uses the **v** as sterm and the **a** as xterm. Hence, the OXT server time is linear to the selectivity of **v**, because it needs to do an additional exponentiation for each tuple from the TSet to check against the XSet. When the selectivity of **v** is small, we observe that HXT prototype has $2\% - 8\%$ additional cost comparing with OXT. However, it slows down with the increase of selectivity of **v**. This is because HXT requires to access HBase to get HVE ciphertext, increasing the selectivity also means the server needs to do more HBase access, which increase the load of I/O.

Another conjunctive query use the **a** as sterm and **v** as xterm, the server then runs in a steady constant time regardless of the selectivity of **v**. In above case, HXT has $2\% - 8\%$ overhead against the OXT over time. This also illustrate the importance of choosing the least frequent term as sterm.

The query time on the client side of HXT and OXT is demonstrated in Fig 6. Comparing with server, client doesn't have any I/O operation, so it purely reflects the computation costs, and it fits well with the analysis in subsection 6.2 as HXT is 2 times slower than OXT.

However, as we design our prototypes to perform query in parallel, the computation cost of HVE key generation on client side can be overlapped by the xtag generation (it always slower than xtoken generation because it doesn't use preprocessed elements),

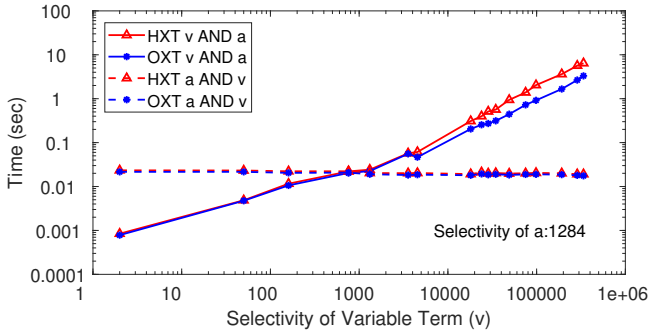

Figure 7: Overall query delay comparison between HXT and OXT in 2.93GB dataset.

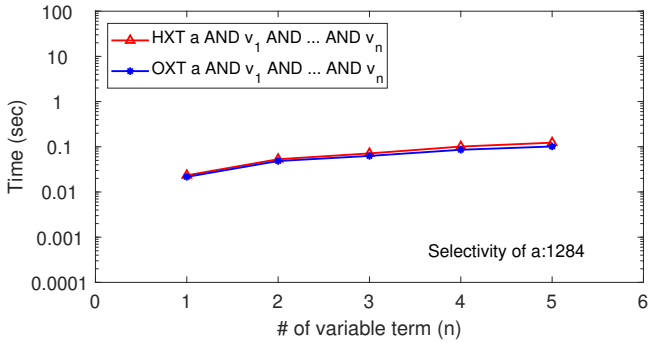

Figure 8: Overall query delay comparison between HXT and OXT under multi-keyword setting in 2.93GB dataset.

as well as the HBase I/O for loading HVE ciphertext on server side. As shown in Fig. 7, the overall performance of HXT is not affected by the computation on client side.

The last evaluation in this part aims to compare the performances of HXT and OXT for querying multiple keywords. In this evaluation, the sterm is identical to the fixed term **a** in previous two-keyword evaluation, but we introduce more variable terms $\{\mathbf{v_n}\}, \mathbf{n} \in [1, 5]$ as xterms in the conjunctive query. As shown in Fig. 8, the query delay increases if the query has more keywords. The underlying reason is that the size of m' is proportional to the size of keyword list, and HXT server is required to have more I/O operation with the increasing size of m'. However, such delay is affordable: the HXT prototype has only 8% and 21% additional cost when the query has two and six keywords, respectively.

7.3.3 Scalability of HXT. We leverage three EDBs we generated from Wikimedia Downloads [19] to demonstrate the scalability of the HXT protocol. We follow the the same evaluation method in [12] to inject artificial query terms to randomly selected documents. Fig. 9 shows that the implementation of HXT preserves the scalability of OXT protocol even when the database sizes exceed the memory we assigned for database query. The query time of HXT is independent of the size of the database. Instead, it is linear to the size of result set.

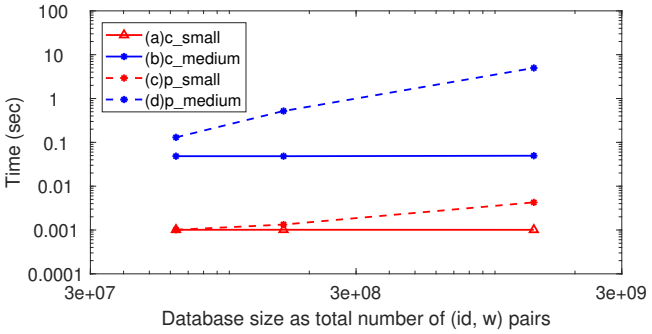

Figure 9: HXT Scalability test in various dataset, the test is running in four cases: (a) constant small (10) result set; (b) constant medium-size (10000) result set; (c) proportional small result set; (d) proportional medium result set;

We also evaluate the query delay for different sizes of datasets as well as the bandwidth communication, which are given in the full version.

8 CONCLUSION

In this paper, we propose a new searchable encryption protocol to obtain better security than to the current existing OXT protocol at the cost of slightly increasing the communication and computation overhead.

In particular, we have introduced hidden cross-tags (HXT) protocol by employing Bloom filters (BF) and newly introduced lightweight symmetric hidden vector encryption (SHVE). It is similar to OXT [12] except that we replace the XSet by an SHVE encryption of BF. The search algorithm re-constructs the search tokens and performs SHVE query algorithm instead of an XSet membership test. It has been shown that our SHVE is selectively simulation-secure, and our HXT is computationally correct, semantically secure against selective adversaries. Implementation and experiments have been conducted to compare the efficiency of the SHVE with those pairing-based HVEs and HXT versus OXT accordingly.

Some possible further research directions are: (*i*) to establish a protocol achieving a better security (by even removing WRP from the leakage profile) robust to the recent attacks [37], while supporting Boolean queries, (*ii*) to apply HXT to other types of queries including rich queries [17, 18], (*iii*) to employ HXT in dynamic SSE scheme with forward/backward security [7].

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and constructive suggestions.

The authors would also like to thank Mr. Piotr Szul from Data61 for his kindly help in the use of High Performance Computing Resource. The work is supported in part by the Data61-Monash CRP, ARC Discovery Project grant DP180102199, Oceania Cyber Security Centre POC scheme, Qualcomm India Innovation Fellowship 2017 and DRDO (Grant: DFTM/02/3111 /M/01/JCBCAT/1288/D(R&D) dated 07 July 2017). Debdeep would also like to thank his DST Swarnajayanti fellowship for partial support during the duration of the project.

REFERENCES

[1] Apache. 2015. Hadoop. https://hadoop.apache.org[online]. (2015).
[2] Apache. 2015. HBase. https://hbase.apache.org[online]. (2015).
[3] S. Blake-Wilson, N. Bolyard, V.Gupta, C. Hawk, and B. Moeller. 2006. RFC4492: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). *RFC4492, Internet Engineering Task Force* (2006).
[4] B.H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.
[5] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. 2004. Public Key Encryption with Keyword Search. In *EUROCRYPT 2004*. 506–522.
[6] D. Boneh and B. Waters. 2007. Conjunctive, Subset, and Range Queries on Encrypted Data. In *TCC'07*. 535–554.
[7] R. Bost, B. Minaud, and O. Ohrimenko. 2017. Forward and backward private searchable encryption from constrained cryptographic primitives. In *ACM CCS'17*. 1465–1482.
[8] A. Broder and M. Mitzenmacher. 2004. Network Applications of Bloom Filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.
[9] A. De Caro and V. Iovino. 2011. JPBC: Java Pairing Based Cryptography. In *IEEE SCC 2011*. 850–855.
[10] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *ACM CCS'15*. 668–679.
[11] D. Cash, J. Jaeger, S. Jarecki, C.S. Jutla, H. Krawczyk, M-C. Rosu, and M. Steiner. 2014. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *NDSS'14*.
[12] D. Cash, S. Jarecki, C.S. Jutla, H. Krawczyk, M-C. Rosu, and M. Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO'13*. 353–373.
[13] C-K. Chu, W.T. Zhu, J. Han, J.K. Liu, J. Xu, and J. Zhou. 2013. Security Concerns in Popular Cloud Storage Services. *IEEE Pervasive Computing* 12, 4 (2013), 50–57.
[14] Cloudera. 2018. CDH Overview. https://www.cloudera.com/documentation/enterprise/5-2-x/topics/cdh_intro.html[online]. (2018).
[15] R. Curtmola, J.A. Garay, S. Kamara, and R. Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS'06*. 79–88.
[16] J. Dean and S. Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[17] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M.N. Garofalakis. 2016. Practical Private Range Search Revisited. In *ACM SIGMOD'16*. 185–198.
[18] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M-C. Rosu, and M. Steiner. 2015. Rich Queries on Encrypted Data: Beyond Exact Matches. In *ESORICS 2015*. 123–145.
[19] Wikimedia Foundation. 2017. Wikimedia Downloads. https://dumps.wikimedia.org[online]. (2017).
[20] L. George. 2011. *Advanced HBase Schema Design*. Technical Report. In Hadoop World 2011.
[21] E. Goh. 2003. Secure Indexes. *IACR Cryptology ePrint Archive* 2003 (2003), 216.
[22] IBTA. 2017. InfiniBand Specification. http://www.infinibandta.org/[online]. (2017).
[23] V. Iovino and G. Persiano. 2008. Hidden-Vector Encryption with Groups of Prime Order. In *Pairing 2008*. 75–88.
[24] M.S. Islam, M. Kuzu, and M. Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *NDSS'12*.
[25] J. Katz, A. Sahai, and B. Waters. 2013. Predicate Encryption Supporting Disjunctions, Polynomial Equations, and Inner Products. *J. Cryptology* 26, 2 (2013), 191–224.
[26] K. Liang, C. Su, J. Chen, and J.K. Liu. 2016. Efficient Multi-Function Data Sharing and Searching Mechanism for Cloud-Based Encrypted Data. In *ASIACCS'16*. 83–94.
[27] J.K. Liu, M.H. Au, W. Susilo, K. Liang, R. Lu, and B. Srinivasan. 2015. Secure Sharing and Searching for Real-time Video Data in Mobile Cloud. *IEEE Network* 29, 2 (2015), 46–50.
[28] M. Naveed, S. Kamara, and C.V. Wright. 2015. Inference Attacks on Property-Preserving Encrypted Databases. In *ACM CCS'15*. 644–655.
[29] A. Nikitin. 2016. Bloom Filter Scala. https://alexandrnikitin.github.io/blog/bloom-filter-for-scala/[online]. (2016).
[30] T. Okamoto and K. Takashima. 2012. Adaptively Attribute-Hiding (Hierarchical) Inner Product Encryption. In *EUROCRYPT 2012*. 591–608.
[31] R.A. Popa, C.M.S. Redfield, N. Zeldovich, and H. Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *ACM SOSP'11*. 85–100.
[32] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In *IEEE MSST'10*. 1–10.
[33] D.X. Song, D. Wagner, and A. Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *IEEE S&P 2000*. 44–55.
[34] S. Sun, J.K. Liu, A. Sakzad, R. Steinfeld, and T.H. Yuen. 2016. An Efficient Non-interactive Multi-client Searchable Encryption with Support for Boolean Queries. In *ESORICS 2016*. 154–172.

[35] The Legion of the Bouncy Castle. 2007. Bouncy Castle Crypto APIs. https://www.bouncycastle.org[online]. (2007).
[36] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud'10*.
[37] Y. Zhang, J. Katz, and C. Papamanthou. 2016. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX Security 16*. 707–720.

A PROOFS

Proof of Theorem 2:

We first show a construction for the simulator $\mathcal{S}$ in the simulation experiment. The simulator models the symmetric encryption scheme (Sym.Enc, Sym.Dec) as an ideal cipher. In particular, the adversary $\mathcal{A}$ either issues encryption queries of the form (κ, μ) or decryption queries of the form (κ, c). The simulator $\mathcal{S}$ maintains a table of the form (κ, μ, c). Upon receipt of an encryption/decryption query, it looks up the table, and either returns an already existing entry, or adds a uniformly random entry to the table and returns the same. The simulator operates as follows:

- **Setup phase:** Suppose the adversary $\mathcal{A}$ chooses an attribute vector $\mathbf{x} \in \Sigma^m$. The simulator $\mathcal{S}$ sets $\mathcal{M} = \{\text{'True'}\}$ and provides the same to $\mathcal{A}$. It additionally randomly chooses $c_l \xleftarrow{\$} \{0,1\}^{\lambda+\log\lambda}$ for $l \in [m]$.
- **Query phase 1:** The adversary adaptively chooses predicates $P^{\mathsf{SHVE}}_{\mathbf{v}_j}$, for $j \in [q_1]$. For each such predicate, the simulator $\mathcal{S}$ receives the corresponding wildcard pattern $\alpha(\mathbf{v}_j) = (\alpha_{j,1}, \ldots, \alpha_{j,m})$ and the decryption pattern $\beta(\mathbf{v}_j, \mathbf{x})$. $\mathcal{S}$ then does the following:
  - $\mathcal{S}$ computes $S_j = \{l_i \in [m] \mid \alpha_{j,l_i} = 1\}$. Let $S_j = \{l_1, l_2, \ldots, l_{|S_j|}\}$.
  - If $\beta(\mathbf{v}_j, \mathbf{x}) = 1$, it randomly samples $K \xleftarrow{\$} \{0,1\}^{\lambda+\log\lambda}$. Next, for $i \in [|S_j|]$, it sets the following :
    $$d_{j,0} = \left(\oplus_{i\in[|S_j|]} c_{l_i}\right) \oplus K.$$
    Finally, it sets $d_{j,1} = \text{Sym.Enc}(K, 0^{\lambda+\log\lambda})$. Note that since (Sym.Enc, Sym.Dec) is modeled as an ideal cipher, all the aforementioned Sym.Enc operations are essentially implemented via table-look-up operations.
  - Otherwise, if $\beta(\mathbf{v}_j, \mathbf{x}) = 0$, the simulator sets $d_{j,0}, d_{j,1} \xleftarrow{\$} \{0,1\}^{\lambda+\log\lambda}$.
  - Finally, the simulator sets the decryption key:
    $$\mathbf{s}_j = \left(d_{j,0}, d_{j,1}, S_j\right).$$
    This decryption key is subsequently provided to the adversary $\mathcal{A}$.
- **Challenge phase:** The simulator $\mathcal{S}$ provides $\mathcal{A}$ with the challenge ciphertext $\mathbf{c} = \left(\{c_l\}_{l\in[m]}\right)$.
- **Query phase 2:** The adversary runs a protocol identical to **Query phase 1**, and $\mathcal{S}$ responds with $\mathbf{s}_j$ for $q_1 + 1 \le j \le q_2$ as described above.

The indistinguishability of the ciphertext $\mathbf{c}$ and the secret keys $\mathbf{s}_j$ for $j \in [q_2]$ from the real experiment follows directly from the following facts:

- The payload message is 'True' by default in the predicate-only version of the scheme. Now, for each $j \in [q_2]$ such that $P^{\mathsf{SHVE}}_{\mathbf{v}_j}(\mathbf{x}) = 1$, decrypting $\mathbf{c}$ using $\mathbf{s}_j$ returns 'True'. On the other hand, for each $j \in [q_2]$ such that $P^{\mathsf{SHVE}}_{\mathbf{v}_j}(\mathbf{x}) = 0$, decrypting $\mathbf{c}$ using $\mathbf{s}_j$ returns 'True' with only negligible probability.
- The encryption and decryption outputs of (Sym.Enc, Sym. Dec) are indistinguishable from uniformly random since they are modeled in the ideal cipher model.
- The outputs of the PRF F_0 are indistinguishable from random to a PPT distinguisher $\mathcal{D}$ that can guess the master secret key *msk* with only negligible probability.

Proof of Theorem 4:

First of all, we describe that leakage function $\mathcal{L}$ which consists of two components: $\mathcal{L}_{\mathsf{HXT}}$, the leakage from the HXT protocol, and $\mathcal{L}_{\mathbf{T}}$, the T-set leakage function. On input a database DB and a set of search queries $(\mathsf{s}, \mathsf{x}_2, \ldots, \mathsf{x}_n) \in \mathsf{W}^n$, the leakage function $\mathcal{L}$ can be computed similarly as in [12]: For every $w \in \mathsf{W}$, randomly choose a key $\kappa \in \{0,1\}^\lambda$ and initiates $\mathbf{t}$ as an empty vector; For a counter c, choose a non-zero random $y \in \mathbb{Z}_p$ and compute an encryption of constant string $\text{Sym.Enc}(\kappa, 0^\lambda)$ and put this ciphertext along with y in the c-th component of $\mathbf{t}$; Once the counter reaches its end ($T_w = |\mathbf{T}[w]|$), put $\mathbf{t}$ into the w-th entry of $\mathbf{T}$. Then the leakage function is output as $\left((\mathcal{L}_{\mathsf{HXT}}(\mathsf{DB}, \mathsf{s}, \mathsf{x}_2, \ldots, \mathsf{x}_n)), \mathcal{L}_{\mathbf{T}}(\mathbf{T}, \mathsf{s}), \mathbf{T}[\mathsf{s}]\right)$.

Next, we show the proof of Theorem 4. The proof is structured through a sequence of games. In all games, the adversary supplies a database DB and a list of search queries $\mathbf{q} = (\mathsf{s}, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ at the beginning, where s and x_i are the list of query sterms and xterms, respectively. The first game Game_0 is designed to have the same distribution as $\mathbf{Real}^{\Pi}_{\mathcal{A}}(\lambda)$, where we neglect all false positives of both TSet and BF for simplifying the proof, and the last one can be easily simulated by an efficient algorithm $\mathcal{S}_{\mathsf{HXT}}$. By showing that the distributions of each two successive games are (computationally) indistinguishable, we get the simulator $\mathcal{S}_{\mathsf{HXT}}$ that meets the requirements of the security definition, thus completing the proof of the theorem. In the following, we use $\Pr[\mathrm{G}_i = 1]$ to denote the probability that Game_i outputs 1.

Game$_0$: this game is slightly modified from the real game to make the analysis easier, the details of which are shown in the full version. With $(\mathsf{DB}, \mathsf{s}, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ as input, the game starts to simulate encrypted database EDB(1), then it computes a vector of size Q of stags called STags. Particularly $\mathsf{STags}[i] \leftarrow \mathsf{TSet.GenTag}(\kappa_{\mathsf{T}}, \mathsf{s}[i])$, for $1 \le i \le Q$. Second, it computes BF similar to that of Alg. 1 and finally inputs $\mathbf{c}$ into EDB(2). It finally computes the transcript array $\mathbf{tr}$, with $\mathbf{tr}[i]$ being $(\mathsf{Res}, \mathsf{ResInds}, ((\mathsf{STags}[\mathsf{i}], \mathsf{xtoken}[i], \mathsf{token}[i])))$ for $1 \le i \le Q$, by running the last loop of Alg. 1. Note that the obtained ResInds in $\mathbf{t}$ is computed by looking up the corresponding id values in $\mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$, instead of decrypting the results returned by SE.Search in the real game. Assuming no false positives happening, the distribution of the explained game is exactly the same as the real game. Therefore, $\Pr[\mathrm{G}_0 = 1] \le \Pr[\mathrm{Real}^{\mathsf{HXT}}_{\mathcal{A}}(\lambda) = 1] + \mathsf{negl}(\lambda)$.

Game$_1$: in this game we replace the PRFs F and F_p with random functions. The details of which are shown in the full version. Note that since $F(\kappa_S, \cdot)$ is only evaluated on the same input once, its evaluations can be replaced with random selections from the appropriate range. As to $F_p(\kappa_X, \cdot)$, $F_p(\kappa_I, \cdot)$ and $F_p(\kappa_Z, \cdot)$, they are replaced by f_X, f_I and f_Z, respectively. A standard hybrid argument

implies that there exist efficient adversaries $\mathcal{B}_{1,1}$ and $\mathcal{B}_{1,2}$ such that $\Pr[G_1 = 1] - \Pr[G_0 = 1] \leq \mathsf{Adv}^{\mathsf{PRF}}_{F,\mathcal{B}_{1,1}}(\lambda) + 3\mathsf{Adv}^{\mathsf{PRF}}_{F_p,\mathcal{B}_{1,2}}(\lambda)$.

Game$_2$: this game replaces only the encryption of document identifiers with that of constant string 0^λ. In the game, the encryption is operated for polynomial, say $\mathsf{poly}(\lambda)$, times, so by a standard hybrid argument we can see that the indistinguishability between these two games can be reduced to IND-CPA security of the symmetric encryption. That is, there exists an efficient adversary $\mathcal{B}_2$, such that $\Pr[G_2 = 1] - \Pr[G_1 = 1] \leq \mathsf{poly}(\lambda) \cdot \mathsf{Adv}^{\mathsf{IND-CPA}}_{\mathcal{B}_2,\mathsf{Sym}}(\lambda)$.

Game$_3$: in this game BF and xtoken are generated in an alternative but equivalent way, which is shown in the full version. Loosely speaking, all possible values $g^{f_X(w)f_I(\mathsf{id})}$ for each identifier $\mathsf{id} \in \mathsf{DB}(w)$ and keyword $w \in \mathsf{W}$ are pre-computed and stored in an array A. Moreover, some xtoken values in transcripts, which correspond to impossible matches, are generated and stored in another array B.

Then arrays A and B are used to compute BF and xtoken. In particular, for a given w and $\mathsf{id} \in DB(w)$ the element $A[w, \mathsf{id}]$ instead of $g^{f_X(w)f_I(\mathsf{id})}$ is added to BF. Note that $A[w, \mathsf{id}]$ is exactly the value $g^{f_X(w)f_I(\mathsf{id})}$, so BF is the same as in previous game. In addition, it is easy to see that the transcript $\mathbf{tr}[\mathsf{i}]$ will be the same only if $\mathsf{xtoken}[i]$ and $\mathsf{token}[i]$ are the same in both games. We note that $\mathsf{token}[i]$ depends on $\mathsf{xtoken}[i]$, so we only focus on the generation of $\mathsf{xtoken}[i]$ array in the following.

In Game$_2$, the $\mathsf{xtoken}[c, \ell]$ for the ℓ-th xterm $\mathsf{x}_\ell[i]$ (of the i-th query) and $c \in [T]$ is set to be $g^{f_Z(\mathsf{s}[i]||c)\cdot f_X(\mathsf{x}_\ell[i])}$. In the current game, however, $\mathsf{xtoken}[c, \ell]$ is generated by first looking up $\mathsf{DB}[\mathsf{s}[i]] = (\bar{\mathsf{id}}_1, \ldots, \bar{\mathsf{id}}_{T_s})$, $\mathsf{WPerms}[\mathsf{s}[i]] = \sigma$ and $\mathbf{t}$, where $\mathbf{t} = (f_I(\bar{\mathsf{id}}_{\sigma(c)})/f_Z(\mathsf{s}[i]||c), e_c)_{c\in[T_s]}$ by the correctness of TSet. Then for $c \in [T_s]$ and $\ell \in [2, n]$, it retrieves (y_c, e_c), such that $y_c = f_I(\bar{\mathsf{id}}_{\sigma(c)})/f_Z(\mathsf{s}[i]||c)$, and sets $\mathsf{xtoken}[c, \ell]$ to be $A[\bar{\mathsf{id}}_{\sigma(c)}, \mathsf{x}_\ell[i]]^{1/y_c} = g^{f_Z(\mathsf{s}[i]||c)\cdot f_X(\mathsf{x}_\ell[i])}$. For $c \in [T] \setminus [T_s]$, $\mathsf{xtoken}[c, \ell]$ is set to be $B[\mathsf{s}[i], \mathsf{x}_\ell[i], c] = g^{f_X(\mathsf{x}_\ell[i])\cdot f_Z(\mathsf{s}[i]||c)}$.

It is easy to observe from the above that the $\mathsf{xtoken}[c, \ell]$ is exactly the same as in Game$_2$. Therefore, we have $\Pr[G_3 = 1] = \Pr[G_2 = 1]$.

Game$_4$: this game is almost identical to the previous one, except that the values y are now drawn randomly from $\mathbb{Z}^*_p$. Due to the modifications made in Game$_3$, the random function f_Z is chosen during the first steps of the algorithm and never evaluated again later, so z is uniformly and independently distributed. Moreover, since $y = \mathsf{xid}\cdot z^{-1}$, for any $w \in \mathsf{W}$ and $c \in [T_w]$, the value of y is also uniform and independent of the rest of the randomness in the game. Thus replacing y with random values does not affect the distribution of the resulted game, so we have $\Pr[G_4 = 1] = \Pr[G_3 = 1]$.

Game$_5$: this game is similar to the previous game, except that all the values of A and B arrays are selected at random from $\mathbb{G}$. Under the DDH assumption, there exists an efficient algorithm $\mathcal{B}_3$ such that $\Pr[G_5 = 1] - \Pr[G_4 = 1] \leq \mathsf{Adv}^{\mathsf{DDH}}_{\mathbb{G},\mathcal{B}_3}(\lambda)$.

To show the indistinguishability between these two games, a simple reduction can be conducted similarly as in [12]. Briefly speaking, the values of X array in G_4 are the g^a values, and the X values are raised to the power of xid when computing A and to the power of $f_Z(w||c)$ when computing B, where xid and $f_Z(w||c)$ act as the b values of the DDH tuple. Thus, A and B in G_4 have values of the form g^{ab}, while in G_5 they are replaced with random values. Differentiating between them can be easily reduced to breaking the DDH assumption, we omit the details here.

Game$_6$: in this game TSet is generated by using simulator $\mathcal{S}_T$, which is shown in the full version. The existence of such a simulator is guaranteed by the security notion of T-Sets. In addition, we remove some irrelevant code (some selecting random functions) and other routines remained the same as G_5. Similar to the analysis shown in [12], there exists an efficient algorithm $\mathcal{B}_4$, under the security definition of TSet, such that $\Pr[G_6 = 1] - \Pr[G_5 = 1] \leq \mathsf{Adv}^{\mathsf{TSet}}_{\mathcal{B}_4}(\lambda)$.

Game$_7$: this game is like the previous one, except that the second part of EDB (i.e., $\mathsf{EDB}(2) = \mathbf{c}$) and the search tokens $\mathsf{token}[i]$ are generated by running the simulator $\mathcal{S}_{\mathsf{HVE}}$ of HVE. To show the indistinguishability between Game$_7$ and Game$_6$, we let $\alpha(\mathbf{v}_c) = [m] \setminus \{H_j[\mathsf{xtoken}[c, \ell]^{y_c}]\}^{j\in[1,k]}_{\ell\in[2,n]}$ and $\beta(\mathbf{v}_c, \mathsf{BF}) = P^{\mathsf{HVE}}_{\mathbf{v}_c}(\mathsf{BF})$. Now we consider the following adversary $\mathcal{B}_5$ against selective simulation security of HVE. $\mathcal{B}_5$ starts to simulate Game$_6$/Game$_7$ by generating TSet, BF and xtoken exactly as in Game$_6$, and then simulates $\mathbf{c}$ and token with the response from the real/ideal game of HVE. Note that, assuming no false positive happens, it holds that $P^{\mathsf{HVE}}_{\mathbf{v}_c}(\mathsf{BF}) = 1$ *iff* $\bar{\mathsf{id}}_{\sigma(c)} \in \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap^n_{\ell=2} \mathsf{DB}(\mathsf{x}_\ell[i])$, hence $\mathcal{B}_5$ can derive the input $(\alpha(\mathbf{v}_c), \beta(\mathbf{v}_c, \mathsf{BF}))$ of $\mathcal{S}_{\mathsf{HVE}}$ from BF and $\mathsf{xtoken}[c, \ell]$.

By the description of Game$_6$ and Game$_7$, we know that the real game of HVE with $\mathcal{B}_5$ perfectly simulates Game$_6$, while the ideal game with $\mathcal{B}_5$ perfectly simulates Game$_7$, so we have that $\Pr[G_7 = 1] - \Pr[G_6 = 1] \leq \mathsf{Adv}^{\mathsf{HVE}}_{\mathcal{B}_5}(\lambda)$.

Game$_8$: To enable the final simulator to work well with its given leakage profile, the way of array A being accessed is changed to an alternative but equivalent way. We note that the array A in Game$_7$ is only accessed when generating the xtoken, and not ever used for producing $\mathbf{c}$ because the simulator of HVE does not receive the actual BF (as mentioned in the ideal game of HVE). More specifically, in this game we replace with a random selection the access of array A during the generation of xtoken for the case of $\bar{\mathsf{id}}_{\sigma(c)} \notin \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap^n_{\ell=2} \mathsf{DB}(\mathsf{x}_\ell[i])$, except for the repeated accesses, which does not affect the distribution of xtoken. Note that, a repeated access to the same position of A happens only if it is called during two distinct search queries, since computing xtoken for one single query touches only unique position of A. More precisely, for an element indexed by (id, w) to be accessed twice, it must hold that $\mathsf{id} \in \mathsf{DB}(\mathsf{s}[i]) \cap \mathsf{DB}(\mathsf{s}[j])$ for some $i \neq j$ and $\mathsf{x}_\alpha[i] = \mathsf{x}_\beta[j]$ for some $\alpha, \beta \in [2, n]$. The condition for such a repeated access is exactly captured by the third "if" statement in the last loop of this game (exactly, the IP leakage component). If this condition does not apply, the xtoken is randomly selected from $\mathbb{G}$. Furthermore, it is easy to observe that both token and $\mathbf{c}$ rely heavily on $(\alpha(\mathbf{v}_c), \beta(\mathbf{v}_c, \mathsf{BF}))$ which are derived from xtoken, so we have that $\Pr[G_8 = 1] = \Pr[G_7 = 1]$.

Simulator: In the following, we present a simulator $\mathcal{S}_{\mathsf{HXT}}$ that takes as input the leakage profile $\mathcal{L}(\mathsf{DB}, \mathsf{s}, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ consisting of $(N, \mathsf{EP}, \mathsf{SP}, \mathsf{WRP}, \mathsf{IP}, \mathcal{L}_T(\mathsf{DB}, \mathsf{s}), \mathbf{T}[\mathsf{s}])$[8] and outputs a simulated EDB and $\mathbf{tr}$. By showing that the simulator produces the same distribution as Game$_8$ and then combining the relations between

[8]Note that $\mathcal{L}_T(\mathsf{DB}, \mathsf{s})$ and $\mathbf{T}[\mathsf{s}]$ are computed in the same way as [12].

the games, we get the simulator satisfying the requirements in Theorem 3.

First of all, our simulator $\mathcal{S}_{\mathsf{HXT}}$ will compute a restricted equality pattern of $\mathsf{x} \doteq (\mathsf{x}_2, \ldots, \mathsf{x}_n)$ as below, denoted by $\hat{\mathbf{x}}$. Then it proceeds to produce its final output through algorithm 3. The restricted equality pattern $\hat{\mathbf{x}}$ can be computed as follows in terms of the leakage IP by defining a table such that $\hat{\mathbf{x}}[t_1, \alpha] = \hat{\mathbf{x}}[t_2, \beta]$ iff $\mathsf{IP}[t_1, t_2, \alpha, \beta] \neq \emptyset$. The table $\hat{\mathbf{x}}$ describes which xterms are "known" to be equal by the adversarial server. In particular, we have that

$$\hat{\mathbf{x}}[t_1, \alpha] = \hat{\mathbf{x}}[t_2, \beta] \implies \mathsf{x}[t_1, \alpha] = \mathsf{x}[t_2, \beta], \text{ and} \tag{3}$$

$$(\mathsf{x}[t_1, \alpha] = \mathsf{x}[t_2, \beta]) \wedge (\mathsf{DB}(\mathsf{s}[t_1]) \cap \mathsf{DB}(\mathsf{s}[t_2]) \neq \emptyset) \implies \hat{\mathbf{x}}[t_1, \alpha] = \hat{\mathbf{x}}[t_2, \beta]. \tag{4}$$

Taking as input the leakage profile $(N, \mathsf{EP}, \mathsf{SP}, \mathsf{WRP}, \mathsf{IP}, \mathcal{L}_T(\mathsf{DB}, \mathsf{s}), \mathbf{T}[\mathsf{s}])$ and the restricted equality pattern $\hat{\mathbf{x}}$ computed as above, the simulator then works as in Alg. 3 to generate the $\mathsf{EDB} = (\mathsf{TSet}, \mathbf{c})$ and the transcript $\mathbf{tr}$.

In the simulation, array A is only filled out for positions $w \in \hat{\mathbf{x}}$ and $\mathsf{id} \in \bigcup_{i=1} \big(\mathsf{WRP}[i] \cup \bigcup_{j \neq i, \alpha, \beta} \mathsf{IP}[i, j, \alpha, \beta]\big)$, which is used to keep the reuse pattern of A during the generation of xtoken. Similarly, the permutations σ's are assigned with respect to EP, the repetition of which captures that of sterms s. When computing the transcript $\mathbf{tr}[i]$ for the i-th query, the simulator sets the "revealed" indices for that query as $R[i] \leftarrow \mathsf{WRP}[i] \cup \bigcup_{j \neq i, \alpha, \beta} \mathsf{IP}[i, j, \alpha, \beta]$ and puts them in canonical order, calling them $\bar{\mathsf{id}}_1, \bar{\mathsf{id}}_2, \cdots, \bar{\mathsf{id}}_{|R[i]|}$. Since $R[i] \subseteq \mathsf{DB}(\mathsf{s}[i])$, the simulator then pads $R[i]$ up to size $\mathsf{SP}[i]$ by setting $\bar{\mathsf{id}}_k$ for $k \in [|R[i]|, \mathsf{SP}[i]]$ to be dummy symbols $\perp$. After that, the simulator uses SP, WRP, IP to simulate xtoken as described in Alg. 3.

Next we show the output of the simulator $\mathcal{S}_{\mathsf{HXT}}$ is identically distributed as that of Game_8. It is clear that the distributions of $\mathbf{t}, (y_c, e_c)$ are identical to Game_8, as (TSet, STags) are computed exactly in the same way. In addition, the permutations σ's have the same distribution, since they are chosen uniformly at random and reused in the same pattern in both cases. Moreover, we can see that identifiers in $\mathsf{DB}(\mathsf{s}[i])/\mathsf{DB}(\mathsf{SP}[i])$ are used in the random order determined by σ, except identifiers not appearing as relevant results are padded with dummy symbols in $\mathsf{DB}(\mathsf{SP}[i])$, and that they follow the same logic in both Game_8 and the simulated game (cf. Alg. 3): if $\sigma(c)$-th identifier is in either $\mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$ or the set of identifiers containing the sterm of another query with some same xterm, then the corresponding position of A is accessed; otherwise, a random group element is selected. At last, what we need to do is to show the accessed entries from A follow the same repetition in both games, which is analyzed as below.

Suppose that $(\mathsf{id}_1, \mathsf{x}_\ell[i])$ and $(\mathsf{id}_2, \mathsf{x}_\nu[j])$ are any two identifier/keyword pair accessed from A in Game_8. Then the simulator $\mathcal{S}_{\mathsf{HXT}}$ will read the positions $(\mathsf{id}_1, \hat{\mathbf{x}}[i, \ell])$ and $(\mathsf{id}_2, \hat{\mathbf{x}}[j, \nu])$ instead. To show the simulation is identical to Game_8, next we argue that

$$(\mathsf{id}_1, \mathsf{x}_\ell[i]) = (\mathsf{id}_2, \mathsf{x}_\nu[j]) \iff (\mathsf{id}_1, \hat{\mathbf{x}}[i, \ell]) = (\mathsf{id}_2, \hat{\mathbf{x}}[j, \nu]).$$

Obviously, the $\Leftarrow$ direction follows readily from (3). As to the other direction, we know that $\mathsf{id}_1 = \mathsf{id}_2$ are members of the following set

$$\big(\mathsf{WRP}[i] \cup \bigcup_{k \neq i, \alpha, \beta} \mathsf{IP}[i, k, \alpha, \beta]\big) \cap \big(\mathsf{WRP}[j] \cup \bigcup_{k \neq j, \alpha, \beta} \mathsf{IP}[j, k, \alpha, \beta]\big),$$

Algorithm 3 : Simulator $\mathcal{S}_{\mathsf{HXT}}$

for $w \in \hat{\mathbf{x}}$ **and** $\mathsf{id} \in \bigcup_{i=1} (\mathsf{WRP}[i] \cup \bigcup_{j \neq i, \alpha, \beta} \mathsf{IP}[i, j, \alpha, \beta])$ **do**
  $A[\mathsf{id}, w] \xleftarrow{\$} \mathbb{G}$
end for
for $w \in \mathsf{EP}$ **do** $\mathsf{WPerms}[w] \xleftarrow{\$} \mathsf{Perm}([\mathsf{SP}[i]])$ **end for**
$(\mathsf{TSet}, \mathsf{STags}) \leftarrow \mathcal{S}_T(\mathcal{L}_T(\mathsf{DB}, \mathsf{s}), \mathbf{T}[\mathsf{s}])$
$\mathbf{c} \leftarrow \mathcal{S}_{\mathsf{HVE}}(\alpha(\mathbf{v}_c), \beta(\mathbf{v}_c, \mathsf{BF}))$
$\mathsf{EDB} \leftarrow (\mathsf{TSet}, \mathbf{c})$
for $i = 1 : Q$ **do**
  $\mathbf{t} \leftarrow \mathsf{TSet.Retrieve}(\mathsf{EDB}(1), \mathsf{STags}[i])$; $\sigma \leftarrow \mathsf{WPerms}[\mathsf{EP}[i]]$; $\mathcal{E}[i] \leftarrow \{\}$
  $R[i] \leftarrow \mathsf{WRP}[i] \cup \bigcup_{j \neq i, \alpha, \beta} \mathsf{IP}[i, j, \alpha, \beta]$; $T' \leftarrow |R[i]|$
  $(\bar{\mathsf{id}}_1, \bar{\mathsf{id}}_2, \cdots, \bar{\mathsf{id}}_{T'}, \underbrace{\perp, \cdots, \perp}_{\mathsf{SP}[i]-T'}) \leftarrow \mathsf{DB}[\mathsf{EP}[i]]$
  for $c = 1 : T$ **do**
    $\mathbf{v}_c \leftarrow *^m$
    for $\ell = 2 : n$ **do**
      if $c \leq \mathsf{SP}[i]$ **then**
        $(y_c, e_c) \leftarrow \mathbf{t}[c]$
        if $\bar{\mathsf{id}}_{\sigma(c)} \neq \perp \wedge \bar{\mathsf{id}}_{\sigma(c)} \in \mathsf{WRP}[i]$ **then**
          $\mathsf{xtoken}[c, \ell] \leftarrow A[\bar{\mathsf{id}}_{\sigma(c)}, \hat{\mathbf{x}}[i, \ell]]^{1/y_c}$
        else if $\bar{\mathsf{id}}_{\sigma(c)} \neq \perp \wedge \bar{\mathsf{id}}_{\sigma(c)} \in \bigcup_{j \neq i, \nu} \mathsf{IP}[i, j, \ell, \nu]$ **then**
          $\mathsf{xtoken}[c, \ell] \leftarrow A[\bar{\mathsf{id}}_{\sigma(c)}, \hat{\mathbf{x}}[i, \ell]]^{1/y_c}$
        else
          $\mathsf{xtoken}[c, \ell] \xleftarrow{\$} \mathbb{G}$
        end if
      else
        $\mathsf{xtoken}[c, \ell] \xleftarrow{\$} \mathbb{G}$
      end if
      for $j = 1 : k$ **do** $\mathbf{v}_c[H_j(\mathsf{xtoken}[c, \ell]^{y_c})] = 1$ **end for**
    end for
    $\mathsf{token}_c[i] \leftarrow \mathcal{S}_{\mathsf{HVE}}(\alpha(\mathbf{v}_c), \beta(\mathbf{v}_c, \mathsf{BF}))$
    $\mathsf{res}_c[i] \leftarrow \mathsf{HVE.Query}(\mathsf{token}_c[i], e_c)$
    if $\mathsf{res}_c[i] = \mathrm{True}$ **then**
      $\mathcal{E}[i] \leftarrow \mathcal{E}[i] \cup \{e_c\}$
    end if
  end for
  $\mathsf{Res} \leftarrow \mathcal{E}[i]$; $\mathsf{ResInds} \leftarrow \mathsf{DB}(\mathsf{s}[i]) \cap \bigcap_{\ell=2}^{n} \mathsf{DB}(\mathsf{x}_\ell[i])$
  $\mathbf{tr}[i] \leftarrow (\mathsf{Res}, \mathsf{ResInds}, ((\mathsf{STags}[i], \mathsf{xtoken}[i], \mathsf{token}[i])))$
end for
return $(\mathsf{EDB}, \mathbf{tr})$

as the games only use identifiers from these sets when computing xtoken. This indicates that $\mathsf{id} \doteq \mathsf{id}_1 = \mathsf{id}_2$ belongs to $\mathsf{DB}(\mathsf{s}[i]) \cap \mathsf{DB}(\mathsf{s}[j])$, and thus we can get from (4) that $\hat{\mathbf{x}}[i, \ell] = \hat{\mathbf{x}}[j, \nu]$.

Finally, regarding the distributions of output $\mathbf{c}$ and token, they rely heavily on the distribution of xtoken and can be simulated by running $\mathcal{S}_{\mathsf{HVE}}$ with $\alpha(\mathbf{v}_c)$ and $\beta(\mathbf{v}_c, \mathsf{BF})$ as input. Recall that $\alpha(\mathbf{v}_c)$ and $\beta(\mathbf{v}_c, \mathsf{BF})$ can be derived from xtoken and WRP directly. Up to now, we get that $\mathcal{S}_{\mathsf{HXT}}$ perfectly simulates Game_8 with its leakage.

Proof of Theorem 5:

The main idea of proving this theorem, as shown in [12], is similar to that of Theorem 4, except that we need to invoke the adaptive TSet simulator and respond queries adaptively. Roughly speaking, to handle the adaptivity, the simulator with input N chooses N random group elements and then adds them to BF. When simulating the response to each query, the simulator adaptively "assigns" elements of the BF to id-keyword pairs. This is in contrast to the non-adaptive simulator, where it first initializes the A array and then adds the elements to the BF, as determined by the leakage. Currently, the simulator first chooses the elements of the BF, and then uses them or independent elements to initialise A adaptively.