ABSTRACT The primitive of Searchable Symmetric Encryption (SSE) allows a server to perform search directly over encrypted documents. Recently, forward secure SSE (FSSE) has attracted significant attention due to its secure property for document update. That is, it can prevent the server from learning whether the newly updated document contains previously queried keywords. Most of the existing FSSE schemes are designed for single keyword search. For the case of conjunctive keyword search, prior dedicated FSSE constructions either require inefficient tools or suffer from significant leakage (i.e., it leaks the documents matching each queried keywords). To deal with this challenge, we propose a general framework to design efficient conjunctive keyword FSSE schemes without significant leakage: a basic scheme FOXT-B and an enhanced one FOXT-E. Specifically, in our basic scheme, we combine XSet structure in the Oblivious Cross-Tags protocol (CRYPTO 2013) with a single keyword FSSE to support conjunctive keyword search. However, this simple combination is limited to support document update. That is, the user cannot perform search until all the keyword-document pairs in a document have been updated. To support flexible keyword-document update, we design a new XSet for our enhanced scheme, in which the user can perform search at any time. Finally, we present a formal security analysis and implement our schemes on a real-world database, which demonstrates that our proposed schemes can achieve the desired security goals with practical efficiency.

INDEX TERMS Searchable encryption, forward security, conjunctive keyword search.

I. INTRODUCTION Cloud computing provides seemingly unbounded computation and storage resources over the internet in the matter of pay-as-you-go basis. More and more users prefer to outsource their data to the cloud server for enjoying superior data storage and computing services. However, outsourcing data to remote server suffers from security and privacy concerns [1]–[5]. For example, famous social networking Facebook may have leaked data for millions of users to a political firm Cambridge Analytica. Simple encryption can provide confidentiality of the outsourced data, however, it is a hindrance to search over the encrypted data.

Searchable Symmetric Encryption (SSE), as a promising solution, enables to search over the encrypted data while protecting the confidentiality of the data and queries. Specifically, a user first encrypts the data and generates a search index, which are together outsourced to the server. Latter, the user submits a query to the server, and then the server performs search on the search index and finally returns results to the user. Since song et al. [6] proposed the seminal SSE scheme, a lot of progress [7]–[11] has been made in efficiency, security and expressiveness. Recently, an important line of work focuses on the security of SSE schemes. For example, a passive attacker in [12] and [13] can leverage the common leakages in SSE schemes to reveal the user’s query. Recently, Zhang et al. [14] proposed an even more powerful attack, in which the attacker can adaptively inject...
new documents. With this power, the attacker can recover the content of user’s query by learning which added documents match it.

This work emphasizes the necessity of constructing forward secure SSE scheme, in which the former queries cannot be used to search in the newly added documents. The notion of forward secure SSE was first introduced by Stefanov et al. [15] who also designed an FSSE scheme. However, their ORAM-like construction results in large communication overhead for an update. Recently, Bost et al. [16] only used simple cryptography tool (a public key primitive trapdoor permutation) to propose an efficient FSSE scheme named \( \Sigma_{\text{off}} \) with an optimal update and search complexity. Following, Song et al. [17] transformed the trapdoor permutation from a public key primitive to a symmetric key one, which further improved the search and update efficiency. Other researchers also focused on how to improve the efficiency of FSSE scheme [18]–[20]. However, all the above FSSE schemes are severely limited to perform single keyword search.

In practice, a single keyword query always retrieves a large number of documents, and then the user herself has to filter the desired ones. To provide more practical query, conjunctive keyword search should be supported. That is, the returned documents contain all the queried keywords. A trivial way to design a conjunctive keyword FSSE scheme is that the server conducts intersection between the resultant document sets for each queried keyword. However, this naive solution results in inefficiency (e.g., if the resultant document set is very large for a keyword) and significant leakage (e.g., it reveals the resultant document sets for each queried keyword). [9]

Very recently, some researchers have focused on constructing non-trivial conjunctive keyword FSSE schemes. Hu et al. [21] adopted function-hiding Inner Product Encryption (IPE) to propose an elegant conjunctive keyword FSSE scheme without significant leakage. The main idea is that the server first retrieves the search results for a keyword in a single keyword forward structure, and then uses IPE to test which result contains all the other queried keywords. However, this method does not have the significant leakage at the cost of degrading much search and update efficiency, because of the primitive of IPE. Wu and Li [22] constructed a tree-based structure to achieve forward secure conjunctive keyword search efficiently. The server can top-down perform conjunctive keyword search and return a leaf as the search result when all the nodes from the leaf to the root contain all the queried keywords. Nevertheless, the server can directly obtain the significant leakage by performing search for each queried keyword. To the best of our knowledge, how to design an efficient conjunctive keyword FSSE scheme without significant leakage is still a challenge.

A. OUR CONTRIBUTION

In this paper, we focus on constructing efficient conjunctive keyword FSSE schemes without significant leakage. Our contribution can be summarized as follows:

- We design a general framework to construct two conjunctive keyword FSSE schemes: a basic scheme FOXT-B and an enhanced one FOXT-E. The main idea is that we can combine any single keyword FSSE scheme with a similar structure XSet in Cash et al’s conjunctive keyword scheme to design conjunctive keyword FSSE schemes. Both of our proposed schemes are based on simple cryptographic primitives, which do not reveal the significant leakage while enjoying good efficiency.

- The basic scheme FOXT-B is somewhat a trivial combination, which is limited to update a document. That is, the user is not allowed to perform search until all the keyword-document pairs in a document have been updated. This limitation comes from the structure of XSet. To this end, we construct a new structure of XSet in our enhanced scheme FOXT-E which can support more flexible keyword-document pair update. That is, the user can perform search at any time.

- We present a formal security analysis and provide a thorough implementation on a real-world database. The results demonstrate that our proposed schemes can achieve the desired goals with a comparable computation overhead.

B. RELATED WORK

The first SSE scheme was introduced by Song et al. [6], in which the encryption is based on the documents and thus the search complexity is linear in the number of documents. Then Curtmola et al. [7] constructed a new SSE scheme based on inverted index achieving sublinear search complexity, and gave a formal security definition. After that, a line of work [9], [23]–[26] has been done for SSE security, expressiveness, and performance.

There have been plenty of works on enriching query expressiveness [9], [27]–[31]. The first SSE scheme supporting conjunctive keyword search was proposed by Golle et al. [27], however, the search complexity is linear in the whole number of documents. Subsequently, Cash et al. [9] proposed a notable conjunctive keyword search scheme named OXT, with support for sublinear search complexity. Thus it can be well applied in large database. Since then, several extensions of OXT have been made. Sun et al. [28] made an extension to multi-user setting, in which only the authorized users are allowed to perform search. Following, Wang et al. [29] further improved decryption efficiency. Considering the malicious model, Wang et al. [30] proposed a verifiable scheme based on a simulator to verify the correctness and completeness of the conjunctive search results. Recently, Lai et al. [31] reduced the leakage of OXT and improved the security.

In dynamic setting, an SSE scheme should support data addition and deletion. Kamara et al. [32] proposed the first
sublinear dynamic SSE scheme. The forward and backward security for addition and deletion was introduced by Stefanov et al. [15], and they proposed the first ORAM-inspired forward secure scheme. A powerful file-injection attack [14] leveraged some leakage information during addition operation to efficiently recover the queried keywords, which further motivates the study of forward security. Bost et al. [16] constructed a forward secure SSE scheme based on trapdoor permutation (a public key primitive) with an optimal update and search complexity. Song et al. [17] replaced the trapdoor permutation by symmetric cryptographic primitives to improve efficiency. Kim et al. [20] further improved efficiency when considering document deletion. Note that all the above FSSE constructions support only single keyword search. Very recently, some researchers have focused on conjunctive keyword FSSE schemes. Hu et al. [21] constructed a conjunctive key-word FSSE scheme based on function-hiding IPE. Although this scheme does not have significant leakage, it is inefficient because of the IPE operation. Wu and Li [22] proposed a virtual binary tree structure to design a new conjunctive keyword FSSE scheme. It is efficient for search and update operation, however, it has significant leakage.

C. ORGANIZATION
The rest of this paper is organized as follows. In Section II, we present some preliminaries. In Section III, we propose new forward secure SSE schemes supporting conjunctive keyword search. The security and comparison analysis of the proposed schemes are given in Section IV, and the performance evaluation is presented in Section V. Finally, conclusions will be made in Section VI.

II. PRELIMINARIES
In this section, we present some notations (as shown in Table I) and basic primitives which are used in this work.

A. TRAPDOOR PERMUTATIONS
Similar to $\Sigma_{\text{trap}}$ [16], we adopt a trapdoor permutation (TDP) to generate a chained forward structure. Specifically, a TDP $\pi$ is a permutation over a set $D$. It is easy to evaluate $\pi$ (resp. $\pi^{-1}$) based on the secret key $sk$ (resp. $pk$). $\pi$ is one way while for every $x \in D$

$$\pi_{pk}(\pi_{sk}^{-1}(x)) = x \text{ and } \pi_{sk}^{-1}(\pi_{pk}(x)) = x.$$  

B. SYMMETRIC SEARCHABLE ENCRYPTION
An SSE scheme consists of two entities, a user and a server. The user wants to outsource documents to the server. In order to protect the document and preserve search ability, the user encrypts the documents and generates a search index. Then the user outsources the encrypted documents and search index to the server. When the user wants to perform a query, she submits a search token to the server. Once receiving the search token, the server performs search over the search index and returns the search results to the user. In dynamic SSE scheme, the user is allowed to update a document or a keyword-document pair to the server. In the following, we give the details of a dynamic SSE scheme. A dynamic searchable encryption scheme $\Pi = \{\text{Setup}, \text{Search}, \text{Update}\}$ consists of one algorithm and two protocols.

- $\text{Setup}(\lambda, \text{DB})$: this is an algorithm which takes as input a security parameter $\lambda$ and a database $\text{DB}$, and outputs (EDB, K, $\sigma$), where EDB is the encrypted database, K is a secret key and $\sigma$ is the state of client.
- $\text{Search}(K, \bar{w}, \sigma; \text{EDB})$: this is a protocol between the client and server. If the client would like to perform a conjunctive search $\bar{w} = w_1 \wedge \cdots \wedge w_n$, she takes as input the (K, $\bar{w}$, $\sigma$), and outputs the search token. The server performs search over the EDB based on the search token, and then returns the matching results R to the client. Each document in the results R contains all the queried keywords $w_i$ for $1 \leq i \leq n$.
- $\text{Update}(K, \sigma, \text{op}, \text{in}; \text{EDB})$: this is a protocol between the server with input EDB and the client with input (K, $\sigma$, op, in), where op is the update operation of adding or deleting a document (or a keyword-document pair) and in is an input (id, W(id)) or ($w$, i.d).

C. SECURITY OF SSE
The security of SSE is defined by a real-world and ideal-world based on a leakage function $L$. The definition expresses that an adversary can get no more information about the database other than the information inferred from the leakage function. Specifically, the leakage function $L$ is parameterized as $L = (L^{\text{Spp}}, L^{\text{Seck}}, L^{\text{Updt}})$, the components of which express the leakages from $\text{Setup}, \text{Search}$ and $\text{Update}$ algorithm, respectively. We define two games $\text{Real}$ and $\text{Ideal}$, and the goal of the adversary is to distinguish them. In the following, we give a formal definition.

<table>
<thead>
<tr>
<th>Table 1. Notations in this work.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
</tr>
<tr>
<td>$id_i$</td>
</tr>
<tr>
<td>W(id)</td>
</tr>
<tr>
<td>$W = \cup_{i=1}^d W_{id_i}$</td>
</tr>
<tr>
<td>$w_i$</td>
</tr>
<tr>
<td>DB</td>
</tr>
<tr>
<td>$DB(w) = {id : w \in W_{id}}$</td>
</tr>
<tr>
<td>$\pi$</td>
</tr>
<tr>
<td>$H, H_1, H_2, H_3$</td>
</tr>
<tr>
<td>$F_1, F_2$</td>
</tr>
<tr>
<td>$G$</td>
</tr>
<tr>
<td>$g$</td>
</tr>
<tr>
<td>$Z_p^*$</td>
</tr>
<tr>
<td>$F_p$</td>
</tr>
<tr>
<td>$s \leftarrow S$</td>
</tr>
<tr>
<td>$[T]$</td>
</tr>
</tbody>
</table>
Definition 1: A dynamic SSE scheme $\Pi$ is $\mathcal{L}$-adaptively-secure if for all PPT adversaries $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that

$$\Pr[\text{Real}_A[\lambda] = 1] - \Pr[\text{Ideal}_A[\lambda, S[\lambda]] = 1] \leq \text{negl}(\lambda),$$

where $\text{Real}_A[\lambda]$ and $\text{Ideal}_A[\lambda, S[\lambda]]$ is defined as follows:

- $\text{Real}_A[\lambda]$: $\mathcal{A}$ chooses a database DB and receives EDB by running $\text{Setup}(\lambda^k, \text{DB})$. Then $\mathcal{A}$ repeatedly performs search and update queries with input $w$ and $(\text{op}, \text{in})$ respectively, and receives transcripts by running $\text{Search}(w)$ and $\text{Update}(\text{op}, \text{in})$. Finally, $\mathcal{A}$ gathers all the real transcripts and outputs a bit $b$.

- $\text{Ideal}_A[\lambda, S[\lambda]]$: $\mathcal{A}$ chooses a database DB and receive EDB by running the simulator $\text{Setup}(\lambda^k, \text{DB})$. Then $\mathcal{A}$ repeatedly performs search and update queries with input $w$ and $(\text{op}, \text{in})$ respectively, and receives transcripts by running $\text{Search}(\mathcal{L}[\text{op}])$ and $\text{Search}(\mathcal{L}[\text{op}])$. Finally, $\mathcal{A}$ gathers all the simulated transcripts and outputs a bit $b$.

D. FORWARD SECURITY OF SSE

The notion of forward secure SSE was first introduced by Stefanov et al. [15] and formalized by Bost [16]. Loosely speaking, forward security means the updated keywords cannot be revealed during an update. That is, the server cannot learn whether the updated document contains a specific keyword queried before. At a starting point, we give an example of forward secure SSE scheme in Figure 1. Specifically, the user first outsources three encrypted documents to the server, in which only the document $F_1$ contains keyword $w$. Thus the server returns $F_1$ with the search token for $w$. Later, the user adds a new document $F_4$ which also contains $w$. However, the server cannot find out the new document $F_4$ with the previous search token for $w$. In the following, we borrow the formal definition of forward secure SSE scheme from [16].

Definition 2: A $\mathcal{L}$-adaptively secure dynamic SSE scheme $\Pi = (\text{Setup}, \text{Search}, \text{Update})$ is forward secure if the update leakage can be written as

$$\mathcal{L}[\text{op}, (w, \text{id})] = \mathcal{L}'(\text{op}, \text{id})$$

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

III. THE FORWARD SECURE FSSE CONSTRUCTION

In this section, we present a general framework to construct conjunctive keyword FSSE scheme. In essence, our framework can extend any single keyword FSSE scheme to a conjunctive keyword FSSE scheme without significant leakage. The main idea is that we combine any single keyword FSSE scheme with a structure XSet from the OXT protocol [9]. Specifically, we construct two forward secure conjunctive keyword FSSE schemes: the basic scheme FOXT-B and the enhanced one FOXT-E. The basic scheme is a simple combination, which is limited to only support the update of a whole document $(\text{id}, W(\text{id}))$. That is, the user is allowed to perform search if all the keyword-document pairs in the document are updated. To support more flexible update, we modify the structure of XSet to support keyword-document pair $(w, \text{id})$ update. Thus the user can perform search at any time.

A. GENERAL IDEAS

For simplicity, we adopt the idea of forward secure scheme in [16] to construct a forward secure structure TSet which is used to perform single keyword search. The TSet is a state-based chain to perform single keyword search. In the following, we take keyword $w$ with a list of matching documents $(\text{id}_1, \ldots, \text{id}_n)$ as an example to describe the update and search operation. Specifically, when the client wants to update document $\text{id}_i$, she first uses trapdoor permutation $\pi$ and secret key $sk$ to generate a new state $ST_{e}(w)$ from $ST_{e-1}(w)$. Then she encrypts $\text{id}_i$ as $(e_i, y_i)$ which stores at location $UT_i(w)$ derived from the state $ST_i(w)$. When performing search on keyword $w$, given state $ST_i(w)$ as a search token, the server can generate all the former ones $ST_{e}(w)$ for $1 \leq i < c$ with public key $pk$, and thus retrieve all the results stored in $UT_i(w)$ for $1 \leq i < c$. Note that with $ST_{e}(w)$, the server cannot generate the latter ones $ST_i(w)$ for $i > c$. In this way, the server cannot perform search in the newly updated documents $\text{id}_i$ with a previous search token $ST_{e}(w)$ for $c < i$.

To extend this single keyword FSSE scheme to a conjunctive keyword FSSE scheme, we combine the forward secure structure TSet with structure XSet in [9]. The main idea is that the server can first retrieve the search results DB($w_1$) for the least frequent keyword $w_1$ from TSet, then filter the ones which contain the other queried keywords based on XSet. In essence, XSet contains a function of each pair $(w, \text{id})$, representing as $g^{w}(w, \text{id})$. The server can use a token $\text{token}[:i:j]$ to obviously test whether $\text{id}_i$ contains the keyword $w_j$. If $\text{token}[:i:j] \in \text{XSet}$, the answer is yes.

It is worth noting this method can only support document update. That is, the user cannot perform search until all the pairs $(\text{id}, W(\text{id}))$ are updated. Otherwise, the server can test

2The main idea of performing conjunctive keyword search is to use the structure of XSet. The user can use the cross tag $\text{token}$ to obviously test the existence of a keyword-document pair based on XSet. Thus it is called Oblivious Cross-Tags Protocol (OXT). Since our proposed schemes adopt the idea of OXT, so we call them Forward Oblivious Cross-Tags Protocol (FOXT).
whether a result document id for w₁ is a newly added document for keyword w₁. A detailed example will be presented in the basic construction.

Our enhanced scheme FOXT-E is designed to support more flexible (w, id) pair update. The idea is to prevent the server from testing the membership of (wᵢ, id) in XSet if id is the newly added document for wᵢ. On one hand, we change the XSet structure from a set to a map. That is, only with a correct location, the server can find the right element in the XSet to be tested. On the other hand, we design a shell for each element in the XSet. Only with a proper value, the server can peel the shell. Instead of sending all the correct locations and values used to peel the retrieved elements, we construct a map CR. For each keyword wᵢ, 2 ≤ i ≤ n, wrᵢ servers as a part of search token. With wrᵢ, the server can iteratively generate desired locations and retrieve all the corresponding elements in XSet, then peel off the elements, and finally add the peeled elements in a set XSubSet. The XSubSet is used to test the membership of xtoken. Thus we guarantee the forward privacy when updating pairs.

Figure 2 gives an example of our proposed schemes. We take a query w₁ ∧ w₂ as an example. The basic scheme is presented in (a). The server first retrieves the results of w₁ based on TSet, {(e₁, y₁), · · · , (eᵦ, yᵦ)}, and filters the ones containing w₂ based on XSet. The enhanced scheme is presented in (b). The server uses wrᵢ to iteratively find out the elements in XSet and peels them. The peeled elements generate a new set XSubSet which is used to filter the desired results.

B. BASIC CONSTRUCTION

In this section, we give the construction of our basic forward secure FSSE scheme, FOXT-B. The main idea is that we combine the single keyword FSSE scheme φς [16] with the set XSet in the OXT [9]. Specifically, when adding a document id, we add encrypted tuple (e, y) for each pair (w, id) in the single keyword forward structure TSet, and stores a function of (wᵢ, id) in the set XSet. In this basic scheme, the client cannot perform search until the updates for all the (w, id) pairs are completed, where w ∈ W(id). The details of our basic scheme are given in Algorithm 1.

- **Setup(λ):** on input a security parameter λ, the client generates (pk, sk) for the trapdoor permutation π, chooses keys Kₛ, Kₐ, Kₓ, Kᵦ, K₂, and initializes empty maps Σ, TSet and empty set XSet, where Σ is in the client side, TSet and XSet are in the server side. At last, the client outputs K = (Kₛ, Kₐ, Kₓ, Kᵦ, K₂, sk), EDB = (pk, TSet, XSet), and σ = Σ.
- **Search(w = (w₁, · · · , wₙ); σ; EDB):** to perform a conjunctive keyword search w₁ ∧ · · · ∧ wₙ, the client looks up the local storage Σ to find the latest state STₙ and

---

3 In a map, each element is associated with a location.
Algorithm 1 FOXT-B: Forward Secure Conjunctive SSE with Document Update

Setup($1^k$)

1. $(sk, pk) \leftarrow \text{KeyGen}(1^k)$
2. Select keys $K_S$ and $K_D$ for PRF’s $F_1$ and $F_2$
3. Select keys $K_X, K_I, K_Z$ for PRF $F_p$
4. $\Sigma, \text{TSet} \leftarrow \text{empty map}$
5. $\text{XSet} \leftarrow \emptyset$
6. \text{return}((\text{TSet}, \text{XSet}, sk), (K_S, K_D, K_X, K_I, K_Z, sk), $\Sigma$)

Search($\vec{w} = (w_1, \ldots, w_n), \sigma; \text{EDB}$)

\textbf{Client:}

1. $K_{w_1} \leftarrow F(K_S, w_1)$
2. $(ST_c, c) \leftarrow \Sigma[w_1]$
3. if $(ST_c, c) = \bot$ then return $\emptyset$
4. end if
5. for $1 \leq i \leq c$ do
6. for $2 \leq j \leq n$ do
7. $\text{xtoken}[i, j] \leftarrow g^{F_p(K_S, w_1)(i)} F_p(K_S, w_j)$
8. end for
9. Set $\text{xtoken}[i] = \text{xtoken}[i, 2], \ldots, \text{xtoken}[i, n]$
10. end for
11. Set $(K_{w_1}, ST_c, c, \text{xtoken}[1], \ldots, \text{xtoken}[c])$ to server

\textbf{Server:}

12. $R \leftarrow \emptyset$
13. for $i = c$ to 1 do
14. $UT_i \leftarrow H(K_{w_1}, ST_i)$
15. $(e, y) \leftarrow \text{TSet}[UT_i]$
16. if $\text{xtoken}[i, j]^y \in \text{XSet}$ for all $j$ then
17. $R \leftarrow R \cup \{e\}$
18. end if
19. end for

Update(\text{add}, \text{id}, \sigma; \text{EDB})

\textbf{Client:}

1. $\text{WSet} \leftarrow W(id)$
2. $\text{AddSet} \leftarrow \emptyset$
3. while $|\text{WSet}| \neq \emptyset$ do
4. $w \leftarrow \text{WSet}; \text{WSet} \leftarrow \text{WSet} \setminus \{w\}$
5. $K_w \leftarrow F_1(K_S, w); K_e \leftarrow F_2(K_D, w)$
6. $(ST_c, c) \leftarrow \Sigma[w]$
7. if $(ST_c, c) = \bot$ then
8. $ST_0 \leftarrow ST_0 \cdot \Sigma, c \leftarrow 0$
9. else
10. $ST_{c+1} \leftarrow \pi_{sk}^{-1}(ST_c)$
11. end if
12. $\Sigma[w] \leftarrow (ST_{c+1}, c + 1)$
13. $e \leftarrow \text{Enc}(K_e, \text{id}); xind \leftarrow F_p(K_1, \text{id})$
14. $z \leftarrow F_p(K_Z, w \parallel c + 1); y \leftarrow xind \cdot z^{-1}$
15. $\text{xtag} \leftarrow g^{F_p(K_E, w \cdot xind)}$
16. $UT_{c+1} \leftarrow H(K_w, ST_{c+1})$
17. $\text{AddSet} \leftarrow \{UT_{c+1}, (e, x, \text{xtag})\}$
18. end while
19. Send AddSet to the server

\textbf{Server:}

20. for each $(UT_{c+1}, (e, y, \text{xtag})) \in \text{AddSet}$ do
21. $\text{TSet}[UT_{c+1}] \leftarrow (e, y)$
22. $\text{XSet} \leftarrow \text{xtag}$
23. end for

counter $c$ for the least frequent keyword $w_1$. In addition, the client generates the search token $\text{xtoken}[i, j]$ where $1 \leq i \leq c, 2 \leq j \leq n$. Then the client sends $(ST_c, c)$ and all the xtoken’s to the server. During search protocol, the server first retrieves the encrypted indices for $w_1$ from TSet, then tests whether the indices include other keywords $w_2, \ldots, w_n$ from XSet, and finally returns the encrypted indices which include all the conjunctive keywords.

- **Update(\text{add}, \text{id}, \sigma; \text{EDB}):** to insert a new document $\text{id}$, all the entries $(w, id)$ in the $\text{id}$ need to be updated, where $w \in W(id)$. Specifically, for each keyword $w$, client uses a new state $ST_{c+1}$ to generate a location $UT_{c+1}$. Then client updates the TSet by adding the encrypted tuple $(e, y)$ in the location $UT_{c+1}$. In addition, the client updates XSet by adding $\text{xtag}$ in it.

This basic scheme only supports to update a document. That is, the user cannot perform search until all the keyword-document pairs in the document are updated. The following example shows the reason. First, the user outsourced $id_1$ and $id_2$ to the server, where $DB(w_1) = (id_1)$ and $DB(w_2) = (id_1, id_2)$. Then, the user wants to update a document $id_3$, where $W(id_3) = (w_1, w_2)$. In the following, we will show how the forward privacy is compromised, if the server performs search before updating all the pairs. Specifically, when the pair $(w_1, id_3)$ is updated, the user first submits $(ST_2(w_1), 2, \text{xtoken}[1, 2], \text{xtoken}[2, 2])$ to perform search $w_1 \land w_2$. Then the server uses $(ST_2(w_1), 2)$ to retrieve $(e_1, y_1), (e_2, y_2)$, and tests whether $\text{xtoken}[1, 2]^y_1, \text{xtoken}[2, 2]^y_2$ belong to the XSet. In this case, only $\text{xtoken}[1, 2]^y_1 \in \text{XSet}$ and then $e_1$ is the search result. After the user updates pair $(w_2, id_3)$, the server can test the previous search token $\text{xtoken}[2, 2]^y_2$ belongs to the updated XSet. Thus the server learns that the newly inserted document $id_3$ matches the previous query $w_1 \land w_2$. As a result, the forward security is compromised.

**C. ENHANCED CONSTRUCTION**

In our enhanced construction, it is flexible to support a pair $(w, id)$ update. That is, the user is allowed to
Algorithm 2 FOXT-E: Forward Secure Conjunctive SSE
With Pair Update

\textbf{Setup()}

1: \((sk, pk) \leftarrow \text{KeyGen}(1^k)\)
2: Select keys \(K_S\) and \(K_D\) for PRF’s \(F_1\) and \(F_2\)
3: Select keys \(K_X, K_I, K_Z, K_R\) for PRF \(F_p\)
4: \(K = \{K_S, K_D, K_X, K_I, K_Z, K_R\}\)
5: \(\Sigma, TSet, XSet, CR \leftarrow \text{empty map}\)
6: return((TSet, XSet, CR, pk), (K, sk), \(\Sigma\))

\textbf{Search(\(\tilde{w} = (w_1, \ldots, w_n), \sigma; \text{EDB}\))}

\textbf{Client:}
1: \(K_{w_1} \leftarrow F(K_S, w_1)\)
2: \((ST_{c}, c) \leftarrow \Sigma[w_1]\)
3: if \((ST_{c}, c) = \perp\) then
   return \(\emptyset\)
4: end if
5: for \(1 \leq i \leq c\) do
6: for \(2 \leq j \leq n\) do
7: \(\text{xtoken}[i, j] \leftarrow g^{F_p(K, w_1 || i)} - F_p(K_S, w_j)\)
8: end for
9: end for
10: \(\text{Send} (K_{w_1}, ST_{c}, c, \text{wr}, \text{xtoken})\) to \(\text{server}\)

\textbf{Server:}
11: \(R \leftarrow \{\}
12: \text{XSubSet} \leftarrow \{\}
13: \text{USet } \leftarrow \{\}
14: \text{for } 2 \leq i \leq n \text{ do}
15: \text{while } \text{wr}_i \neq \perp \text{ do}
16: \text{if }\text{wr}_i \neq \perp \text{ do}
17: \text{end while}
18: \text{end for}
19: \text{end for}
20: \text{Return }\text{XSubSet} \cup \{\text{xtag}\}
21: \text{Return }\text{XSubSet} \cup \{\text{xtag'}\}
22: \text{Return }\text{CR}[\text{XSubSet}]

\textbf{Update(add, w, id, \sigma; \text{EDB})}

\textbf{Client:}
1: \(K_w \leftarrow F_1(K_S, w); K_e \leftarrow F_2(K_D, w);\)
2: \((ST_{c}, c) \leftarrow \Sigma[w]\)
3: \(K_r \leftarrow F_p(K_r, w)\)
4: if \((ST_{c}, c) = \perp\) then
5: \(ST_0 \leftarrow \mathcal{M}, c \leftarrow 0, \text{wr}_c \leftarrow \perp\)
6: end if
7: \(\text{st}_{c+1} \leftarrow n_{sk}^{-1}(ST_{c}), \text{wr}_c \leftarrow F_p(K_r, w \parallel c)\)
8: \text{end if}
9: \(\Sigma[w] \leftarrow (ST_{c+1}, c + 1)\)
10: \(\text{wr}_{c+1} \leftarrow F_p(K_r, w \parallel c + 1)\)
11: \(\mathcal{L}_{c+1} \leftarrow H_1(\text{wr}_{c+1})\)
12: \(\text{cr}_{c+1} \leftarrow H_3(\mathcal{L}_{c+1}) \oplus \text{wr}_c\)
13: \(\text{xtag} \leftarrow \text{xtag} \oplus \mathcal{L}_{c+1}\)
14: \(\text{ut}_{c+1} \leftarrow H(K_w, \text{ST}_{c+1})\)
15: \(\text{AddSet} \leftarrow (\text{ut}_{c+1}, (e, y); \mathcal{L}_{c+1}, \text{xtag}; \mathcal{L}_{c+1}, \text{cr}_{c+1})\)
16: \text{Send} \text{AddSet} \text{to the server}
17: \text{Server:}
18: \text{UT}_{c+1} \leftarrow H(K_w, \text{ST}_{c+1})\)
19: \(\text{XSet[XL}_{c+1}] \leftarrow \text{xtag}\)
20: \text{CR}[\mathcal{L}_{c+1}] \leftarrow \text{cr}_{c+1}\)
21: \text{end for}
22: \text{end if}
23: \text{end for}
24: \text{end while}
25: \text{end for}
26: \text{end for}

perform search at any time. On one hand, we change the structure of XSet from a set to a map. On the other hand, we design a shell for each element in XSet. Only with correct locations, the server can find proper elements in XSet and then peel them. Instead of sending all the locations, we design a map CR which is stored in the server side. With search token, the server can use CR to iteratively locate proper elements in XSet and peel them. Finally, the peeled elements generate a XSubSet set which serves as the XSet in our basic construction. The details are in Algorithm 2. The differences between the basic one are of blue color.

- **Setup**\((1^k)\): we additionally generates a key \(K_R\) for \(F_p\) and initialize a map CR. Besides, we change the set XSet to a map.
- **Search**\((\tilde{w} = (w_1, \ldots, w_n), \sigma; \text{EDB})\): the search token contains an extra part, \(\text{wr}_i\), for each non-frequent keyword \(w_i\). On one hand, each \(\text{wr}_i = F_p(K_r, w_i \parallel c)\) is used to find a location of XSet and retrieve the xtag. On the other hand, the \(\text{wr}_i\) is used to remove the shell of xtag and get xtag’ which is stored in a set XSubSet. Then the server uses the \(\text{wr}_i\) and CR to get the historical \(\text{wr}_{i-1} = F_p(K_r, w_i \parallel c_i-1)\), and obtains the corresponding xtag’. After retrieving all the xtag’, the server obtains
IV. ANALYSIS OF OUR PROPOSED SCHEME

A. SECURITY ANALYSIS

In this section, we give the formal security analysis of our enhanced scheme FOXT-E. Before going analysis, we present the leakage function \( \mathcal{L} \) of our scheme \( \Pi \). There is nothing leaked during the \texttt{Setup} and \texttt{Update} function. That is, \( \mathcal{L}^{\text{Sp}} = \perp \) and \( \mathcal{L}^{\text{Update}} = \perp \). In the following, we give the leakage function \( \mathcal{L}^{\text{Search}} \) during search. Assuming that a sequence of \( T \) conjunctive queries \( q = (s, x) \) are performed. Specifically, \( s_t \) and \( x_t \) denote the stem (least frequent keyword) and xterms (not least frequent keywords) in the \( t \)-th query, \( t \in \{1, 2, \ldots, T\} \). In the following, we present the outputs of the leakage function.

Algorithm 3 Simulator \( S \)

1. (sk, pk) \( \leftarrow \) KeyGen(1^k)
2. TSet, XSet, CR, \( \Sigma \) \( \leftarrow \) empty map
3. \( \lambda \leftarrow 0 \)
4. return((TSet, XSet, CR, pk), (K, sk), \( \Sigma \))

\texttt{Search(} \( \mathcal{L}^{\text{Search}} \) \texttt{)}

1. \( \delta_t \leftarrow \min \text{sp}(s_t) \)
2. \( K_{\delta_t} \leftarrow \text{Key}_1[\delta_t], K_e \leftarrow \text{Key}_2[\delta_t], K_r \leftarrow \text{Key}_3[\delta_t] \)
3. \( ST_0 = \Sigma[\delta_t] \)
4. Parse Hist(s) as [(u_1, add, (e_1, y_1)), \cdots, (u_c, add, (e_c, y_c))] \[ \]
5. if \( c = 0 \) then \[ \]
6. return \( \emptyset \)
7. end if
8. for \( i = 1 \) to \( c \) do \[ \]
9. Program \( H \) s.t. \( H(K_{\delta_t}, ST_i) \leftarrow UT_{ui} \)
10. \( ST_{i+1} \leftarrow \pi_{\delta_t}(ST_i) \)
11. end for
12. for \( 1 \leq i \leq XT[t] \) do \[ \]
13. for \( 1 \leq j \leq \text{SXP}[t, i] \) do \[ \]
14. Program \( H_1 \) s.t. \( H_1(u_{i,j}) \leftarrow \text{CL}_{u_{i,j}} \)
15. Program \( H_2 \) s.t. \( H_2(K_{\delta_t}, ST_j) \leftarrow \text{XL}_{u_{i,j}} \)
16. if \( j = 1 \) then \[ \]
17. \( H_2(\text{wr}_{u_{i,j}-1}) \leftarrow \perp \)
18. \( H_2(\text{wr}_{u_{i,j}}) \leftarrow \hat{H}(\text{wr}_{u_{i,j}-1}) \)
\( \oplus \text{CR} [\text{CL}_{u_{i,j}}] \)
19. else \[ \]
20. \( H_2(\text{wr}_{u_{i,j}-1}) \leftarrow \mathbb{Z}_p \)
21. \( H_2(\text{wr}_{u_{i,j}}) \leftarrow \hat{H}(\text{wr}_{u_{i,j}-1}) \)
\( \oplus \text{CR} [\text{CL}_{u_{i,j}}] \)
end if
end for
end for

\texttt{Update(} \[ \]
1. \( UT_u \leftarrow \{0, 1\}^\mu, XL_u \leftarrow \{0, 1\}^\mu, CL_u \leftarrow \{0, 1\}^\mu \)
2. \( e_u \leftarrow \text{Enc}(K_e, 0^\mu), y_u \leftarrow \mathbb{Z}_p, xtag_u \leftarrow \mathbb{G}, cr_u \leftarrow \mathbb{Z}_p \)
3. AddSet \( \leftarrow \langle UT_{c+1}, (e, y), XL_{c+1}, xtag; CL_{c+1}, cr_{c+1} \rangle \)
4. Send AddSet to the server

- \( \delta_t \in \mathbb{N}_T \) is the search pattern, indicating whether two stems are the same. It is represented as a vector. For example, \( s = (a, b, a, c, b) \), then \( \delta_t = (1, 2, 1, 3, 2) \).
- \( \text{SRP}[t] \) is the access pattern for the stem of the \( t \)-th query, \( \text{SRP}[t] = R(s_t) \).
- \( \text{SP}[t] \) is the number of the search results for the stem of the \( t \)-th query, \( \text{SP}[t] = |R(s_t)| \).
- \( \text{Hist}(s_t) \) as \([u_1, (e_1, y_1)), \cdots, (u_c, (e_c, y_c))]\), where \( c = \text{SP}[t] \) and all \((e, y) \in \text{SRP}[t] \).
- \( \text{XT}[t] \) is the number of xterms of the \( t \)-th query, \( \text{XT}[t] = |x_t| \).
- \( \text{SXP}[t, i] \) is the number of the search results for the \( i \)-th xterm of the \( t \)-th query, \( \text{SXP}[t, i] = |R(x_t)| \).
- \( \text{Hist}(x_t) \) as \([u_{t,1}, \cdots, u_{t,c}, c] \) is the timestamp for the xterm \( x_t \), where \( c = \text{SXP}[t, i] \).
- \( \text{RP}[t, a] \) is the search result of the stem and any xterm \( x_{t,a} \) in the \( t \)-th query, \( \text{RP}[t, a] = [(u_{t,1}, u_{t,1}'), (e_1, y_1)), \cdots, (u_{t,c}, u_{t,c}', (e_c, y_c))]\), where \((e_i, y_i) \in R(s_t) \cap R(x_{t,a}) \) and \( c = |R(s_t) \cap R(x_{t,a})| \).
TABLE 2. Functionality comparison.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Conjunctive</th>
<th>Forward secure</th>
<th>Index size</th>
<th>Search time</th>
<th>Update time</th>
<th>Update type</th>
<th>Owner storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best [16]</td>
<td>√</td>
<td>✓</td>
<td>O(N)</td>
<td>O(t \cdot r_w)</td>
<td>O(t)</td>
<td>Pair</td>
<td>O(m \cdot \log d)</td>
</tr>
<tr>
<td>Cash et al. [9]</td>
<td>√</td>
<td>×</td>
<td>O(N')</td>
<td>O((Q') \cdot r_w)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Hu et al. [21]</td>
<td>√</td>
<td>✓</td>
<td>O(N \cdot m)</td>
<td>O((Q \cdot r_w) \cdot \log d)</td>
<td>O(L)</td>
<td>Pair</td>
<td>O(m \cdot \log d)</td>
</tr>
<tr>
<td>Wu and Li [22]</td>
<td>√</td>
<td>✓</td>
<td>O(N \cdot \log d)</td>
<td>O(Q \cdot (t \cdot r_w))</td>
<td>O(t)</td>
<td>Document</td>
<td>O(m \cdot \log d)</td>
</tr>
<tr>
<td>Our basic scheme</td>
<td>√</td>
<td>√</td>
<td>O(N)</td>
<td>O((Q \cdot (t \cdot r_w))</td>
<td>O(t)</td>
<td>Pair</td>
<td>O(m \cdot \log d)</td>
</tr>
<tr>
<td>Our enhanced scheme</td>
<td>√</td>
<td>√</td>
<td>O(N)</td>
<td>O((Q \cdot (t \cdot r_w))</td>
<td>O(t)</td>
<td>Pair</td>
<td>O(m \cdot \log d)</td>
</tr>
</tbody>
</table>

1 d is the number of documents, N is the number of keyword-document pairs in the database, t is the time cost for a RSA-based asymmetric encryption operation, r_w is the number of matched documents for keyword w, |Q| is the number of queried keywords, L is the height of the tree in Wu and Li scheme, m is the number of keywords in the database.

- IP[t_1, t_2, \alpha, \beta] is the condition insertion pattern. If s_{t_1} \neq s_{t_2} and x_{t_1, \alpha} = x_{t_2, \beta}, IP[t_1, t_2, \alpha, \beta] = R(s_{t_1}) \cap R(s_{t_2}). Otherwise, IP[t_1, t_2, \alpha, \beta] = \emptyset.

The main difference from the static conjunctive keyword search scheme is that this dynamic one reveals the timestamps of updating. Besides, our scheme additionally reveals the number of the search results for the xterms, which comes from the size of XSubSet during the search. In the following, we give formal security analysis.

**Theorem 1:** Let F_1, F_2 and F_3 be secure PRFs, \pi be a secure one-way trapdoor permutation, and H, H_1, H_2 and H_3 be secure hash functions modeled as random oracles. Then our FOXT-E scheme is L-semantically secure against adaptive attacks.

**Proof:** The proof works by a sequence of indistinguishable games. The first game is the real-world game and the last one is the ideal-world game which is simulated by an efficient simulator. Here we mainly introduce how the simulator works. The goal of the simulator is to use the leakages to simulate the transcripts between the user and the cloud, including the update items of TSet, XSet, CR and the search token for a query. In the Update protocol, the simulator randomly generates locations and elements for TSet, XSet and CR. In the Search protocol, the simulator uses the leakage function to simulate the search token. On the one hand, we program the hash functions H, H_1, H_2 and H_3. It is easy to program H, H_1 and H_2 using the timestamps. For programming H_3, we leverage the timestamps and the elements in CR to program. On the other hand, we need to simulate xtoken. There are three types of element (e_i, y) in the Hist. Since the one belonging to RP, we use the timestamps to locate the corresponding xtag and wr, and then compute the xtoken. For the one belonging to IP, we need to randomly select xtoken and record it. For the one which is neither in RP nor in IP, the xtoken is randomly selected from the group \mathbb{G}. In this way, the xtoken is simulated. The details of the simulator are presented in Algorithm 3. We omit the search operation in the server side, because it is easy for the server to perform search using the simulated search token.

Then we construct successive indistinguishable games. The first one is designed to have the same distribution as the Real\_A(k), and the last one has the same distribution as the Ideal\_A(k) which is generated by the simulator. In the games, we model the hash functions as random oracles, replace pseudo-random functions by random functions. By showing that the distributions of all games are distinguishable, our scheme meets the security goals. Since the games can be constructed easily, we omit the details here.

**B. COMPARISON**

In this section, we compare our proposed schemes with some related constructions [9], [16], [21], [22]. In general, our schemes can be seen as a combination of single keyword FSSE scheme [16] and a conjunctive keyword SSE scheme [9] to efficiently achieve conjunctive keyword search with forward privacy. Specifically, both forward secure single keyword search and conjunctive keyword search are well studied in [16] and [9]. Recently, non-trivial conjunctive keyword FSSE schemes have also attracted considerable attention. Hu et al.’s conjunctive FSSE scheme is designed based on the primitive of Inner Product Encryption. Their scheme can successfully avoid the significant leakage. However, plenty of bilinear paring operations result in inefficiency of the scheme. Wu and Li’s tree-based scheme can also achieve conjunctive keyword search with forward security. Unfortunately, this scheme leaks the undesired significant leakage. Compared with these two schemes, our proposed conjunctive keyword FSSE schemes are efficient without significant leakage. Our basic scheme only supports document update like Hu et al.’s scheme. To support more flexible update, our enhanced scheme can support keyword-document pair update. Table 2 provides a detailed comparison of all the schemes. Note that, for the update time, we consider the efficiency for updating a keyword-document pair in all the compared schemes.

**V. PERFORMANCE EVALUATION**

In this section, we provide a thorough experimental evaluation of our proposed schemes. We mainly focus on the size of EDB, the efficiency of search and update process. First, we give a description of the prototype to implement the algorithms. Second, we analyze the experimental results and compare them with VBTree scheme in [22].

We leverage crypto++ \(^4\) to realize the cryptographic primitives. Specifically, we use AES in CTR mode for the PRFs, SHA256 for the hash functions. For all those schemes, we use Rocksdb \(^5\) to store the data in the client and server. We evaluate our scheme using two LINUX machines, one is a service node and the other is a client node. Both of them have 4 cores

\(^4\)Crypto++: A free C++ class library of cryptographic schemes, https://cryptopp.com/
\(^5\)RocksDB: A Persistent Key-Value Store for Flash and RAM Storage, http://rocksdb.org
FIGURE 3. The size of EDB.

FIGURE 4. The search efficiency.

FIGURE 5. The update efficiency.

with 8 threads (Intel Xeon E5-1620 v3, 3.50 GHz), 16 GB RAM and 1 TB disk, running on Ubuntu 14.04 LTS. We consider the real-world database from Wikimedia Download\(^6\) to implement. The size of database (the number of distinct keyword-document pairs) ranges from \(2^{15}\) to \(2^{20}\). Note that both schemes in [22] and [21] are conjunctive keyword FSSE schemes. Here we just compare our proposed schemes with Wu and Li’s scheme [22]. Because it is obvious that our schemes are superior to Hu et al.’s scheme [21] in efficiency. Besides, our framework can extend any single keyword FSSE scheme, so we adopt Song et al.’s scheme [17] as the single keyword FSSE scheme to implement because of its good efficiency.

First, we focus on the size of search index EDB which is stored in the server side and used to perform search. As shown in Figure 3, the size of EDB increases with the size of database for the three compared schemes. The size of EDB in FOXT-E is larger than the FOXT-B, because we additionally introduce a CR map in FOXT-E scheme. The size of EDB in the VBTree is the largest among the compared schemes. The reason is that the number of encrypted keyword-document pairs in the EDB is much more than that in the database.

Then, we focus on the search efficiency in the compared schemes. We perform arbitrary two keywords conjunctive search in the real-world database to evaluate the search efficiency. As shown in Figure 4, the search time is not linear in the size of database. Specifically, all the three schemes have sublinear search efficiency. Even though our FOXT-E is slower than the other two, it is also very efficient and acceptable in practice.

Finally, we focus on the update efficiency in our compared schemes. As shown in Figure 5, the time costs for an update in the three schemes increase with the number of update pairs. Our proposed schemes are a little less efficient than VBTree, because of the modular exponential operation. Fortunately, we can see that it is very efficient for all the three schemes to update a keyword-document pair.

VI. CONCLUSION

In this paper, we focus on the forward secure conjunctive keyword search over encrypted database. The main contribution is to present a general framework to construct two FSSE schemes supporting conjunctive keyword search. Our schemes can simultaneously achieve conjunctive search and forward privacy. Both of our schemes, FOXT-B and FOXT-E, reduce the significant leakage which exists in almost all existing schemes. That is, our schemes do not reveal the set of document matching each queried keyword. In our basic scheme, FOXT-B, we can extend a single keyword FSSE scheme to support conjunctive keyword search by combining an XSet structure in an Oblivious Cross-Tags protocol (a conjunctive keyword search protocol). This simple combination results in inflexible document update. That is, the user cannot perform search until all the pairs in a document are updated. In the enhanced scheme, FOXT-E, we change the structure of XSet to support more flexible pair update, in which the user can perform search at any time. We provide a formal security proof and thorough experiment on a real-world database, which demonstrates that our scheme can achieve the desired security goals with practical efficiency.

REFERENCES


\(^6\)https://dumps.wikimedia.org


