Libertas: Backward Private Dynamic Searchable Symmetric Encryption Supporting Wildcards

Jeroen Weener[1], Florian Hahn[1], and Andreas Peter[1,2(✉)]

[1] Services and Cybersecurity, University of Twente, Enschede, The Netherlands
{f.w.hahn,a.peter}@utwente.nl
[2] Safety-Security-Interaction, University of Oldenburg, Oldenburg, Germany

Abstract. When outsourcing data, *Searchable Symmetric Encryption (SSE)* allows clients to query the server for their encrypted files without compromising data confidentiality. Several attacks against searchable encryption schemes have been proposed that leverage information leakage the schemes emit when operating. Schemes should achieve *Forward and Backward Privacy* to mitigate these types of attacks. Despite the variance of query types across SSE schemes, most forward and backward private schemes only support exact keyword search. In this research, we extend backward privacy notions and their underlying leakage functions to the *Wildcard Search* domain. Additionally, we present Libertas; a construction that provides backward privacy to any wildcard supporting SSE scheme. If the scheme is forward private, this property is inherited. We prove security in the established $\mathcal{L}$-adaptive security model with respect to a leakage function $\mathcal{L}$. We show that the performance overhead scales linearly with the number of deletions.

Keywords: Searchable Encryption · Backward Privacy · Wildcard Search

1 Introduction

Motivation and Related Work. The demand for Cloud Service Providers (CSPs) has increased in recent years. They offer convenient, scalable and on-demand data storage and processing. Sharing data with a CSP can be inappropriate, however, as the provider is not fully trusted. Encryption prevents them from accessing the data but in doing so, obstructs their ability to process it. Searchable Symmetric Encryption (SSE) allows clients to first encrypt and later search their data once placed at the CSP, allowing for selective data retrieval. SSE was first explored by Song et al. [16]. The server hosts data of the client in encrypted form. Later, the client can search the data for keywords and retrieve relevant data from the server without revealing the searched keywords or the content of the data. Goh [11] introduced the concept of an *index* to speed up searches. An index is a data structure where keyword identifiers are stored

Published by Springer Nature Switzerland AG 2022
S. Sural and H. Lu (Eds.): DBSec 2022, LNCS 13383, pp. 215–235, 2022.
https://doi.org/10.1007/978-3-031-10684-2_13

per document identifier. Searches use the index to find the matching document identifiers instead of using the documents themselves. Most SSE schemes offer a trade-off between efficiency, e.g. achieving search times sublinear in the number of indexed keywords, and security, e.g. leaking the result set of an encrypted search query in form of the access pattern. Curtmola et al. [9] gave a security notion formalizing this trade-off for static datasets. Later, Kamara et al. [14] extended this notion for *Dynamic SSE* (DSSE) schemes that allow documents to be added and removed.

To allow for more query flexibility, several extensions to the basic single keyword search have been proposed. *Boolean queries* enable different kinds of boolean operators such as conjunctions, disjunctions and negations. Cash et al. [7] describe a scheme featuring boolean queries. *Substring queries* match keywords that contain the query as a substring. Prefix and suffix queries match keywords that either start or end with the query. Chase and Shen [8] describe a substring supporting scheme. *Wildcard queries* allow the client to insert different kinds of wildcard characters in the search query. For example, a wildcard character can replace exactly one character or multiple characters. The search query 'com*' matches keywords 'computer' and 'company', while the search query 'c_t' matches 'cat' and 'cut'. Several schemes using several constructions have been proposed that allow for wildcard queries. One such construction is by storing keywords in Bloom filters. Suga et al. [18] consider Bloom filters allowing for substring, fuzzy and wildcard queries. Hu et al. [12] introduce a scheme that is more efficient compared to Suga et al. and allows clients to update the database. The scheme by Bösch et al. [3] operates in the dynamic environment and implement wildcard support by generating and inserting all wildcard variants of a keyword upon database insertion. This transforms the problem of wildcard search into exact keyword search, but heavily burdens server storage depending on the type and number of allowed wildcards in queries. Zhao and Nishide [20] describe a wildcard supporting scheme capable of supporting two types of wildcards by cleverly storing keyword characteristics in Bloom filters. Faber et al. [10] propose a matching algorithm based on the conjunctive search scheme by Cash et al. [7]. Among other query types, Faber et al.'s scheme supports single keyword search, substring and wildcard queries, and supports any combination of these query types using boolean operators.

Unfortunately, search queries and updates potentially leak information such as the matching documents or the affected keywords. As shown by previous lines of research, despite a scheme conforming to the aforementioned security definitions by Curtmola et al. [9] and by Kamara et al. [14], this information leakage can allow for powerful *Leakage Abuse Attacks (LAAs)*. Islam et al. [13] describe the first so-called query recovery attack exploiting full background knowledge about indexed data to retrieve the keyword hidden in search queries. Cash et al. [6] extended this work by describing several LAAs, both passive and active. They improve the attack of Islam et al. and introduce plaintext recovery attacks; attacks that aim to recover the content of the stored documents. Zhang et al. [19] describe efficient file-injection attacks aiming to recover keywords from search queries, assuming little knowledge of stored content.

To defend against query recovery attacks, the security notion *forward privacy* was informally defined by Stefanov et al. [17] and later formalized by Bost [4]. Forward private schemes do not leak which keywords are considered during updates, making it impossible to link newly added data to earlier search queries. Later, Bost et al. [5] introduced *Backward Privacy (BP)* as another security notion for DSSE schemes. In backward private schemes, search queries cannot be executed over deleted entries, limiting the potential of (future) attacks. They introduce three levels of BP of decreasing strength: 1. BP with insertion pattern leakage, 2. BP with update pattern leakage, and 3. weak BP. However, despite all the advancements in flexible search queries as mentioned above, most forward and backward private schemes only consider the exact keyword search setting.

Our Contribution. This paper introduces Libertas; a construction for providing BP with update pattern leakage to any wildcard supporting DSSE scheme. It is proven secure against adaptive adversaries. We implement Libertas with the wildcard supporting DSSE scheme by Zhao and Nishide (Z&N) [20] and evaluate its performance. The source code of our implementation is publicly available on GitHub.[1] Our results show that Libertas' search performance overhead is hardly affected by increases in index size, result set size or the number of wildcards in a query. Libertas does experience noticeable overhead during searches when the index contains entries of removed document-keyword pairs. This overhead scales linearly with the number of deletions.

Organization of the Paper. In Sect. 2 we introduce searchable symmetric encryption and recall its security definitions - including forward and backward privacy. We then discuss the extension of the search functionality from exact-keyword towards wildcards in Sect. 3. Afterwards, in Sect. 4 we give our construction Libertas that provides backward privacy for any dynamic wildcard supporting SSE scheme and present a practical evaluation for a concrete instantiation in Sect. 5. In Sect. 6 we discuss real-world applications and possible further improvements before we conclude our research in Sect. 7.

2 Preliminaries

2.1 Searchable Symmetric Encryption (SSE)

Searchable symmetric encryption schemes allow clients to store documents at a third party in encrypted form and later search for them using queries. Search functionality is typically achieved by the use of an *index*. The exact implementation of the index differs per scheme, but it is typically a look-up table that links keyword identifiers to the identifiers of matching documents. The client can search these keyword identifiers to find the document identifiers of matching documents. These document identifiers can then be used to send the matching documents to the client. SSE schemes can be static or dynamic. Dynamic

[1] https://github.com/LibertasConstruction/Libertas.

SSE (DSSE) schemes differ from static schemes as they additionally allow for updates to the index after the initial setup phase. In this work, we only consider dynamic SSE schemes. Encryption (decryption) and uploading (downloading) of documents is often not relevant for the security analysis and thus treated as an independent step in the process. Typically, documents are encrypted using AES in CBC mode and stored on the server. SSE schemes consist of eight algorithms.

$K \leftarrow \mathsf{Setup}(\lambda)$ is run one time by the client, at the start of the scheme. It takes as input the security parameter λ and outputs the scheme's key K.

$\gamma \leftarrow \mathsf{BuildIndex}(\lambda)$ is run one time by the server, at the start of the scheme. It takes as input the security parameter λ and outputs an (at that point empty) index γ.

$\tau^{\mathsf{srch}} \leftarrow \mathsf{SrchToken}(K, w)$ is run by the client during search operations. It takes as input the scheme's key K and a keyword w that is to be searched for. The output is a search token τ^{srch}.

$\tau^{\mathsf{add}} \leftarrow \mathsf{AddToken}(K, \mathsf{ind}, w)$ is run by the client during add operations. It takes as input the scheme's key K and a document-keyword pair, consisting of a document identifier ind and a keyword w. The output is an add token τ^{add}.

$\tau^{\mathsf{del}} \leftarrow \mathsf{DelToken}(K, \mathsf{ind}, w)$ is run by the client during delete operations. It takes as input the scheme's key K and a document-keyword pair, consisting of a document identifier ind and a keyword w. The output is a delete token τ^{del}.

$R \leftarrow \mathsf{Search}(\gamma, \tau^{\mathsf{srch}})$ is run by the server after receiving the search token τ^{srch} from the client. Together with the index γ, this results in a result set R, which is a list of document identifiers: $R : (\mathsf{ind}_1, \ldots, \mathsf{ind}_n)$. Usually, the server sends back the encrypted documents corresponding to these document identifiers.

$\gamma' \leftarrow \mathsf{Add}(\gamma, \tau^{\mathsf{add}})$ is run by the server after receiving the add token τ^{add} from the client. This token is used to update index γ to a new index γ'.

$\gamma' \leftarrow \mathsf{Del}(\gamma, \tau^{\mathsf{del}})$ is run by the server after receiving the delete token τ^{del} from the client. This token is used to update index γ to a new index γ'.

SrchToken and Search together form the **Search** protocol of the SSE scheme. In the same way, AddToken and Add, and DelToken and Delete form the **Add** and **Delete** protocol of the SSE scheme, respectively.

Result-Hiding SSE Schemes. Result-hiding SSE schemes hide the document identifiers, normally uncovered during the Search algorithm, from the server. An example of such a scheme is the Masked Index Scheme by Bösch et al. [3]. Results are hidden by altering the **Search** protocol, adding new algorithms DecSearch and FetchDocuments in a second round. In these schemes, Search outputs encrypted document identifiers at the server that have to be sent to the client for decryption. The client, therefore, has control over what happens with the document identifiers and does not necessarily have to reveal them to the server. The server can, however, identify when the same document identifier is sent multiple times, as its encryption in the index does not change if no additional measures are taken. The modified algorithm Search (now having two rounds), and the new algorithms DecSearch and FetchDocuments are formally defined as

$R^* \leftarrow \mathsf{Search}(\gamma, \tau^{\mathsf{srch}})$ is run by the server, taking as input the index γ and a search token τ^{srch}, resulting in an encrypted result set R^*.

$R \leftarrow \mathsf{DecSearch}(K, w, R^*)$ is run by the client, taking as input the scheme's key K, the keyword that is searched for w and the encrypted result set R^*. The output of the algorithm is the list of identifiers of matching documents $R : (\mathsf{ind}_1, \ldots, \mathsf{ind}_n)$.

$D \leftarrow \mathsf{FetchDocuments}(R)$ is run by the server, taking as input the document identifiers revealed by DecSearch. The server outputs documents D corresponding to the document identifiers in R.

Note that, in this extended 2-round **Search** protocol, document identifiers are first revealed to the client rather than the server.

2.2 Leakage Functions

A *leakage function* $\mathcal{L}$ describes what information is leaked by an SSE scheme. Leakage can be abused to mount an attack. Schemes should therefore aim to leak as little as possible. Typically, there exists a trade-off between the security and the efficiency of the scheme. By allowing some leakage, the scheme can achieve greater efficiency, and to achieve higher security, one should restrict the leakage, which incurs a penalty for efficiency. The total leakage of a dynamic SSE scheme consists of $\mathcal{L}^{\mathsf{Srch}}$, $\mathcal{L}^{\mathsf{Add}}$ and $\mathcal{L}^{\mathsf{Del}}$, which are the leakage functions corresponding to the Search protocol, Add protocol and Delete protocol, respectively. Leakage functions keep an internal state Q. The Search protocol inserts (u, w) tuples in Q, where u is the timestamp of the operation and w is the searched keyword. Update operations append $(u, \mathsf{op}, (\mathsf{ind}, w))$ tuples to Q, where op is an indicator of the nature of the operation (add or delete) and (ind, w) is the document-keyword pair to either add or delete. The security of SSE schemes is typically measured by the amount of information they leak during operations. To describe this leakage, multiple leakage functions are often considered in the literature. The most common functions are the *search pattern* and *access pattern*, which both relate to search operations.

$$\mathsf{sp}(w) = \{u \mid (u, w) \in Q\},$$
$$\mathsf{ap}(w) = \{\mathsf{ind} \mid (u, \mathsf{add}, (\mathsf{ind}, w)) \in Q \wedge \nexists\, u' > u, \text{ s.t. } (u', \mathsf{del}, (\mathsf{ind}, w)) \in Q\}.$$

The search pattern $\mathsf{sp}(w)$ leaks the timestamps u at which the keyword w has been searched for. If a scheme leaks the search pattern, one is able to infer which search queries pertain to the same keyword. The access pattern $\mathsf{ap}(w)$ leaks the document identifiers ind of documents that contain keyword w at the time of the search.

2.3 Security Model

The security model for SSE schemes often considered in the literature is called $\mathcal{L}$-*adaptive security* [9]. An $\mathcal{L}$-adaptively-secure SSE scheme Σ leaks only explicitly defined leakage $\mathcal{L}$. In this model, an adversary $\mathcal{A}$ can adaptively trigger the

different algorithms that make up the scheme with inputs of choice and observe their outputs. We define a real world game $\text{SSE}^{\Sigma}_{\text{Real}_{\mathcal{A}}}(\lambda, n)$ and an ideal world game $\text{SSE}_{\text{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}}(\lambda, n)$, where λ is the security parameter and n is the number of queries that are executed. In $\text{SSE}^{\Sigma}_{\text{Real}_{\mathcal{A}}}(\lambda, n)$, Σ is executed honestly, while in $\text{SSE}_{\text{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}}(\lambda, n)$, a simulator $\mathcal{S}$ simulates Σ using $\mathcal{L}$ as input. The task of the adversary is to output a bit b, distinguishing between a real transcript and a simulated one. Σ is $\mathcal{L}$-adaptively secure if the transcripts are indistinguishable. Algorithm 1 describes the security games $\text{SSE}^{\Sigma}_{\text{Real}_{\mathcal{A}}}(\lambda, n)$ and $\text{SSE}_{\text{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}}(\lambda, n)$, adapted for result-hiding SSE schemes. We use these games in the security proof of Libertas, which is a result-hiding scheme, in Sect. 4.3.

Definition 1 ($\mathcal{L}$-Adaptive Security). *An SSE scheme Σ is **$\mathcal{L}$-adaptively-secure** with respect to a leakage function $\mathcal{L}$, if for any polynomial-time adversary $\mathcal{A}$ issuing a polynomial number of queries $n(\lambda)$, there exists a probabilistic polynomial time simulator $\mathcal{S}$ such that:*

$$\left|\mathbb{P}[SSE^{\Sigma}_{Real_{\mathcal{A}}}(\lambda, n) = 1] - \mathbb{P}[SSE_{Ideal_{\mathcal{A},\mathcal{S},\mathcal{L}}}(\lambda, n) = 1]\right| \leq negl(\lambda).$$

$\underline{\text{SSE}^{\Sigma}_{\text{Real}_{\mathcal{A}}}(\lambda,n)}$

$K \leftarrow \mathsf{Setup}(\lambda)$
$\gamma \leftarrow \mathsf{BuildIndex}(\lambda)$
for $i = 1$ to n **do**
  $(\mathsf{type}_i, \mathsf{params}_i, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i(\mathsf{st}_{\mathcal{A}}, \gamma, \tau, \mathbf{R}^*, \mathbf{R})$, where τ, $\mathbf{R}^*$ and $\mathbf{R}$ consist of all tokens, encrypted result sets and result sets, respectively, generated in previous iterations.
  if $\mathsf{type}_i = \mathsf{Search}$ **then**
    $w_i \leftarrow \mathsf{params}_i$
    $\tau_i^{\mathsf{srch}} \leftarrow \mathsf{SrchToken}(K, w_i)$
    $R_i^* \leftarrow \mathsf{Search}(\gamma, \tau_i^{\mathsf{srch}})$
    $R_i \leftarrow \mathsf{DecSearch}(K, w_i, R_i^*)$
  else if $\mathsf{type}_i = \mathsf{Add}$ **then**
    $(\mathsf{ind}_i, w_i) \leftarrow \mathsf{params}_i$
    $\tau_i^{\mathsf{add}} \leftarrow \mathsf{AddToken}(K, \mathsf{ind}_i, w_i)$
    $\gamma \leftarrow \mathsf{Add}(\gamma, \tau_i^{\mathsf{add}})$
  else
    $(\mathsf{ind}_i, w_i) \leftarrow \mathsf{params}_i$
    $\tau_i^{\mathsf{del}} \leftarrow \mathsf{DelToken}(K, \mathsf{ind}_i, w_i)$
    $\gamma \leftarrow \mathsf{Del}(\gamma, \tau_i^{\mathsf{del}})$
  end if
end for
$b \leftarrow \mathcal{A}_{n+1}(\mathsf{st}_{\mathcal{A}}, \gamma, \boldsymbol{\tau}, \boldsymbol{R}^*, \boldsymbol{R})$
Return b

$\underline{\text{SSE}_{\text{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}}(\lambda,n)}$

$(\widetilde{\gamma}, \mathsf{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_0(\lambda)$
for $i = 1$ to n **do**
  $(\mathsf{type}_i, \mathsf{params}_i, \mathsf{st}_{\mathcal{A}}) \leftarrow \mathcal{A}_i(\mathsf{st}_{\mathcal{A}}, \widetilde{\gamma}, \widetilde{\boldsymbol{\tau}}, \widetilde{\boldsymbol{R}}^*, \widetilde{\boldsymbol{R}})$
  if $\mathsf{type}_i = \mathsf{Search}$ **then**
    $w_i \leftarrow \mathsf{params}_i$
    $(\widetilde{\tau}_i^{\mathsf{srch}}, \widetilde{R}_i^*, \widetilde{R}_i, \mathsf{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_i(\mathsf{st}_{\mathcal{S}}, \mathcal{L}^{\mathsf{Srch}}(w_i))$
  else if $\mathsf{type}_i = \mathsf{Add}$ **then**
    $(\mathsf{ind}_i, w_i) \leftarrow \mathsf{params}_i$
    $(\widetilde{\tau}_i^{\mathsf{add}}, \widetilde{\gamma}, \mathsf{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_i(\mathsf{st}_{\mathcal{S}}, \mathcal{L}^{\mathsf{Add}}(\mathsf{ind}_i, w_i))$
  else
    $(\mathsf{ind}_i, w_i) \leftarrow \mathsf{params}_i$
    $(\widetilde{\tau}_i^{\mathsf{del}}, \widetilde{\gamma}, \mathsf{st}_{\mathcal{S}}) \leftarrow \mathcal{S}_i(\mathsf{st}_{\mathcal{S}}, \mathcal{L}^{\mathsf{Del}}(\mathsf{ind}_i, w_i))$
  end if
end for
$b \leftarrow \mathcal{A}_{n+1}(\mathsf{st}_{\mathcal{A}}, \widetilde{\gamma}, \widetilde{\boldsymbol{\tau}}, \widetilde{\boldsymbol{R}}^*, \widetilde{\boldsymbol{R}})$
Return b

Fig. 1. Adaptive Semantic Security Games for Result-Hiding DSSE Schemes

2.4 Forward Privacy

Forward privacy has been introduced by Stefanov et al. [17] and is further explored by Bost et al. [4]. Informally, a forward private scheme's update algorithm does not leak whether a newly inserted element matches previous search queries. Formally, forward privacy is defined as follows.

Definition 2 (Forward Privacy). *An $\mathcal{L}$-adaptively-secure SSE scheme is **forward-private** iff the add leakage function $\mathcal{L}^{\mathsf{Add}}$ and delete leakage function $\mathcal{L}^{\mathsf{Del}}$ can be written with stateless functions $\mathcal{L}'$, $\mathcal{L}''$ as:*

$$\mathcal{L}^{\mathsf{Add}}(\mathsf{ind}, w) = \mathcal{L}'(\mathsf{ind}) \qquad \textit{and} \qquad \mathcal{L}^{\mathsf{Del}}(\mathsf{ind}, w) = \mathcal{L}''(\mathsf{ind}),$$

where ind *is the document identifier, w is the updated keyword.*

2.5 Backward Privacy (BP)

In addition to forward privacy, Bost et al. specify backward privacy (BP) [5]. BP limits what one can learn regarding updates on keyword w from a search query on that keyword. Informally, search queries in backward private schemes only reveal document-keyword pairs that have been added, but not subsequently deleted. Limiting the leakage on search queries alone is not sufficient, however, as observing the document-keyword pairs during update queries would trivially grant the server the information on whether a document has been deleted. Therefore, backward private schemes limit the leakage of both search and update queries. Obtaining a full backward private scheme requires hiding the update pattern (see $\mathsf{Updates}(w)$ hereafter), resulting in expensive SSE schemes. Bost et al. have defined three notions of BP with decreasing strength, depending on the amount of information that is leaked [5]. We consider the two strongest notions.

Backward privacy with insertion pattern leakage: Upon a search query for keyword w, leaks the document identifiers currently matching w, the timestamps at which they were inserted and the total number of updates on w.

Backward privacy with update pattern leakage: Upon a search query for keyword w, leaks the document identifiers currently matching w, the timestamps at which they were inserted and the timestamps of all the updates on w (but not their content).

The differences between these notions become clear when considering an example with the following updates: $(\mathsf{add}, \mathsf{ind}_1, w_1)$, $(\mathsf{add}, \mathsf{ind}_1, w_2)$, $(\mathsf{add}, \mathsf{ind}_2, w_1)$, $(\mathsf{del}, \mathsf{ind}_1, w_1)$. Upon a search query for keyword w_1, the first notion reveals ind_2, that it was inserted at time slot 3 and that there were three updates to w_1. The second notion additionally reveals that updates regarding w_1 occurred at time slot 1, 3 and 4. To formally define these notions, Bost et al. define the leakage functions $\mathsf{UpHist}(w)$, $\mathsf{TimeDB}(w)$ and $\mathsf{Updates}(w)$. $\mathsf{UpHist}(w)$ contains the timestamp, operation and document identifier of every update. $\mathsf{TimeDB}(w)$ outputs

all documents currently matching w and the timestamp of insertion. $\mathsf{Updates}(w)$ results in a list of timestamps of updates on keyword w.

$$\begin{aligned}\mathsf{UpHist}(w) = &\ \{(u, \mathsf{op}, \mathsf{ind}) \mid (u, \mathsf{op}, (\mathsf{ind}, w)) \in Q\},\\ \mathsf{TimeDB}(w) = &\ \{(u, \mathsf{ind}) \mid (u, \mathsf{add}, (\mathsf{ind}, w)) \in Q \wedge\\ &\ \nexists\ u' > u \text{ s.t. } (u', \mathsf{del}, (\mathsf{ind}, w)) \in Q\},\\ \mathsf{Updates}(w) = &\ \{u \mid (u, \mathsf{op}, (\mathsf{ind}, w)) \in Q\}.\end{aligned}$$

Note how the access pattern $\mathsf{ap}(w)$ can be constructed from $\mathsf{TimeDB}(w)$ and how $\mathsf{TimeDB}(w)$ and $\mathsf{Updates}(w)$ can be derived from $\mathsf{UpHist}(w)$. This means that $\mathsf{UpHist}(w)$ leaks strictly more than those leakage functions and that $\mathsf{TimeDB}(w)$ leaks strictly more than $\mathsf{ap}(w)$. A scheme leaking $\mathsf{UpHist}(w)$ therefore inherently also leaks $\mathsf{TimeDB}(w)$, $\mathsf{ap}(w)$ and $\mathsf{Updates}(w)$.

$$\begin{aligned}\mathsf{ap}(w) = &\ \{\mathsf{ind} \mid (u, \mathsf{ind}) \in \mathsf{TimeDB}(w)\},\\ \mathsf{TimeDB}(w) = &\ \{(u, \mathsf{ind}) \mid (u, \mathsf{add}, \mathsf{ind}) \in \mathsf{UpHist}(w) \wedge\\ &\ \nexists\ u' > u \text{ s.t. } (u', \mathsf{del}, \mathsf{ind}) \in \mathsf{UpHist}(w)\},\\ \mathsf{Updates}(w) = &\ \{u \mid (u, \mathsf{op}, \mathsf{ind}) \in \mathsf{UpHist}(w)\}.\end{aligned}$$

The different notions of BP can be formally described using these leakage functions.

Definition 3 (Backward Privacy). *An $\mathcal{L}$-adaptively-secure SSE scheme is **insertion pattern revealing backward-private** iff the search, add and delete leakage functions $\mathcal{L}^{\mathsf{Srch}}$, $\mathcal{L}^{\mathsf{Add}}$ and $\mathcal{L}^{\mathsf{Del}}$ can be written as:*

$$\mathcal{L}^{\mathsf{Srch}}(w) = \mathcal{L}'(\mathsf{TimeDB}(w), a_w) \quad \text{and} \quad \mathcal{L}^{\mathsf{Add}}(\mathsf{ind}, w) = \perp \quad \text{and} \quad \mathcal{L}^{\mathsf{Del}}(\mathsf{ind}, w) = \perp,$$

where a_w denotes the number of updates on w and $\mathcal{L}'$ is stateless.

*An $\mathcal{L}$-adaptively-secure SSE scheme is **update pattern revealing backward-private** iff the search and update leakage functions $\mathcal{L}^{\mathsf{Srch}}$, $\mathcal{L}^{\mathsf{Add}}$ and $\mathcal{L}^{\mathsf{Del}}$ can be written with three stateless functions $\mathcal{L}'$, $\mathcal{L}''$ and $\mathcal{L}'''$ as:*

$$\begin{aligned}\mathcal{L}^{\mathsf{Srch}}(w) &= \mathcal{L}'(\mathsf{TimeDB}(w), \mathsf{Updates}(w)),\\ \mathcal{L}^{\mathsf{Add}}(\mathsf{ind}, w) &= \mathcal{L}''(w),\\ \mathcal{L}^{\mathsf{Del}}(\mathsf{ind}, w) &= \mathcal{L}'''(w).\end{aligned}$$

3 Wildcards

Different SSE schemes support different kinds of search queries. The simplest search query consists of one keyword. This is called *exact keyword search*: clients can search for one keyword and receive all documents containing this keyword. In our research, we consider DSSE schemes supporting *single keyword wildcard search*. This setting extends exact keyword search by additionally allowing that the searched keyword can contain wildcards. We consider two types of wildcards:

'_' and '*'. The first wildcard type, '_', is used to indicate the presence of a single character. The second wildcard type, '*', is used to indicate the presence of zero or more characters. Suppose we upload $(\mathsf{ind}_1, \text{'cat'})$ and $(\mathsf{ind}_2, \text{'cut'})$. The query $q =$ 'c_t' would match both ind_1 and ind_2. Consider additionally uploading another document-keyword pair $(\mathsf{ind}_3, \text{'catering'})$. The query $q_2 =$ 'cat*' matches with ind_1 and ind_3.

3.1 Wildcard Security

As searches of wildcard supporting SSE schemes operate on queries q rather than keywords w, we first describe a natural extension of the aforementioned leakage functions to the wildcard setting. We introduce the following notation: let w be a keyword and q be a query that can contain wildcards. If keyword w is contained in query q we denote this as $w \dot{\subseteq} q$. 'cat' $\dot{\subseteq}$ 'c_t'. We change the definition of the internal state Q of leakage functions to the following: the list Q stores every search query as a (u, q) pair, where u is the timestamp and q is the search string (a keyword, possibly containing wildcard characters). Update queries remain the same: a $(u, \mathsf{op}, (\mathsf{ind}, w))$ tuple, where op is the operation (add or del) and (ind, w) is the document-keyword pair. We define $\mathsf{sp}(q)$, $\mathsf{ap}(q)$, $\mathsf{UpHist}(q)$, $\mathsf{TimeDB}(q)$ and $\mathsf{Updates}(q)$ as wildcard adaptations of $\mathsf{sp}(w)$, $\mathsf{ap}(w)$, $\mathsf{UpHist}(w)$, $\mathsf{TimeDB}(w)$ and $\mathsf{Updates}(w)$, respectively.

$$
\begin{aligned}
\mathsf{sp}(q) = \ & \{u \mid (u, q) \in Q\}, \\
\mathsf{ap}(q) = \ & \{\mathsf{ind} \mid (u, \mathsf{add}, (\mathsf{ind}, w)) \in Q \wedge \\
& \quad \nexists\, u' > u \text{ s.t. } (u', \mathsf{del}, (\mathsf{ind}, w)) \in Q \ \wedge \ w \dot{\subseteq} q\}, \\
\mathsf{UpHist}(q) = \ & \{(u, \mathsf{op}, \mathsf{ind}) \mid (u, \mathsf{op}, (\mathsf{ind}, w)) \in Q \ \wedge \ w \dot{\subseteq} q\}, \\
\mathsf{TimeDB}(q) = \ & \{(u, \mathsf{ind}) \mid (u, \mathsf{add}, (\mathsf{ind}, w)) \in Q \wedge \\
& \quad \nexists\, u' > u \text{ s.t. } (u', \mathsf{del}, (\mathsf{ind}, w)) \in Q \ \wedge \ w \dot{\subseteq} q\}, \\
\mathsf{Updates}(q) = \ & \{u \mid (u, \mathsf{op}, (\mathsf{ind}, w)) \in Q \ \wedge \ w \dot{\subseteq} q\}.
\end{aligned}
$$

Similarly to their non-wildcard counterparts, $\mathsf{ap}(q)$, $\mathsf{TimeDB}(q)$ and $\mathsf{Updates}(q)$ can be constructed from $\mathsf{UpHist}(q)$. We can extend the notions of BP introduced earlier to the wildcard setting by using the leakage functions we defined.

Definition 4. *A wildcard supporting, $\mathcal{L}$-adaptively-secure SSE scheme is* ***insertion pattern revealing backward-private*** *iff the search, add and delete leakage functions $\mathcal{L}^{\mathsf{Srch}}$, $\mathcal{L}^{\mathsf{Add}}$ and $\mathcal{L}^{\mathsf{Del}}$ can be written as:*

$$
\begin{aligned}
\mathcal{L}^{\mathsf{Srch}}(q) &= \mathcal{L}'(\mathsf{TimeDB}(q), a_q), \\
\mathcal{L}^{\mathsf{Add}}(\mathsf{ind}, w) &= \bot, \\
\mathcal{L}^{\mathsf{Del}}(\mathsf{ind}, w) &= \bot,
\end{aligned}
$$

where a_q denotes the number of updates on q and $\mathcal{L}'$, $\mathcal{L}''$ and $\mathcal{L}'''$ are stateless.

Definition 5. *A wildcard supporting, $\mathcal{L}$-adaptively-secure SSE scheme is* ***update pattern revealing backward-private*** *iff the leakage functions $\mathcal{L}^{\mathsf{Srch}}$, $\mathcal{L}^{\mathsf{Add}}$ and $\mathcal{L}^{\mathsf{Del}}$ can be written with stateless functions $\mathcal{L}'$, $\mathcal{L}''$ and $\mathcal{L}'''$ as:*

$$\begin{aligned}\mathcal{L}^{\mathsf{Srch}}(q) &= \mathcal{L}'(\mathsf{TimeDB}(q), \mathsf{Updates}(q)),\\ \mathcal{L}^{\mathsf{Add}}(\mathsf{ind}, w) &= \mathcal{L}''(w),\\ \mathcal{L}^{\mathsf{Del}}(\mathsf{ind}, w) &= \mathcal{L}'''(w).\end{aligned}$$

4 Libertas: Constructing Wildcard Supporting Update Pattern Revealing Backward Private Schemes

Libertas is a construction for creating the first backward private, wildcard supporting DSSE schemes. Its idea is similar to that of the scheme $B(\Sigma)$ proposed by [5]. Rather than being an SSE scheme on its own, Libertas encapsulates an existing SSE scheme Σ that supports wildcards and document-keyword additions, to provide BP. The idea is as follows: rather than storing document identifiers, store encryptions of document-update pairs, regardless of whether the update was an insertion or a deletion. During searches, send all encrypted document-update pairs to the client for decryption. The client can select relevant document identifiers (those that are added, but not subsequently deleted) and send them to the server to retrieve the documents. This approach makes Libertas result-hiding.

4.1 Construction

Libertas is built from an encryption scheme E and an SSE scheme Σ. E is *which-key concealing* (sometimes referred to as *key-private encryption*), meaning that two encryptions do not leak whether they are encrypted using the same key [1]. Σ supports add operations and wildcard queries, and is $\mathcal{L}_\Sigma$-adaptively secure, where $\mathcal{L}_\Sigma = (\mathcal{L}_\Sigma^{\mathsf{Srch}}, \mathcal{L}_\Sigma^{\mathsf{Add}})$ is defined with stateless functions $\mathcal{L}'$ and $\mathcal{L}''$ as

$$\begin{aligned}\mathcal{L}_\Sigma^{\mathsf{Srch}}(q) &= \mathcal{L}'(\mathsf{sp}_\Sigma(q), \mathsf{UpHist}_\Sigma(q)),\\ \mathcal{L}_\Sigma^{\mathsf{Add}}(\mathsf{ind}, w) &= \mathcal{L}''(\mathsf{ind}, w).\end{aligned}$$

Libertas is described in Algorithm 1. Here, $E_{K_{\mathsf{Lib}}}$ denotes an encryption using E under key K_{Lib}. Returned values are sent over the network.

4.2 Analysis

We analyze the theoretical cost of running Libertas in terms of storage, operations and communication. We compare these components with Σ, as most costs are identical to, or dependent on, Σ.

Storage. The client stores one extra key K_{Lib} and the counter c. The server stores an encryption in its index for every update (including deletions), rather than a document identifier for document-keyword pairs currently in the database.

Algorithm 1. Libertas

$\mathsf{Setup}(\lambda)$

1: $K_\Sigma \leftarrow \Sigma.\mathsf{Setup}(\lambda)$
2: $K_{\mathsf{Lib}} \xleftarrow{\$} \{0,1\}^\lambda$
3: $K = (K_\Sigma, K_{\mathsf{Lib}})$
4: $c \leftarrow 0$

$\mathsf{BuildIndex}(\lambda)$

1: $\gamma \leftarrow \Sigma.\mathsf{BuildIndex}(\lambda)$

$\mathsf{SrchToken}(K, q)$

1: $\tau^{\mathsf{srch}} \leftarrow \Sigma.\mathsf{SrchToken}(K_\Sigma, q)$
2: Return τ^{srch}

$\mathsf{AddToken}(K, \mathsf{ind}, w)$

1: $\tau^{\mathsf{add}} \leftarrow \Sigma.\mathsf{AddToken}(K_\Sigma, E_{K_{\mathsf{Lib}}}(c, \mathsf{add}, \mathsf{ind}, w), w)$
2: $c \leftarrow c + 1$
3: Return τ^{add}

$\mathsf{DelToken}(K, \mathsf{ind}, w)$

1: $\tau^{\mathsf{del}} \leftarrow \Sigma.\mathsf{AddToken}(K_\Sigma, E_{K_{\mathsf{Lib}}}(c, \mathsf{del}, \mathsf{ind}, w), w)$
2: $c \leftarrow c + 1$
3: Return τ^{del}

$\mathsf{Search}(\gamma, \tau^{\mathsf{srch}})$

1: $R^* \leftarrow \Sigma.\mathsf{Search}(\gamma, \tau^{\mathsf{srch}})$
2: Return R^*

$\mathsf{DecSearch}(K, R^*)$

1: Decrypt R^* using K_{Lib} and sort the entries in ascending order based on the value of c, resulting in $((c_1, \mathsf{op}_1, \mathsf{ind}_1, w_1), \ldots, (c_n, \mathsf{op}_n, \mathsf{ind}_n, w_n))$.
2: Let W be the set of distinct keywords in R^*.
3: For all $w \in W$, let $R_w = \{\mathsf{ind} \mid \exists\ i \text{ s.t. } (\mathsf{op}_i, \mathsf{ind}_i, w_i) = (\mathsf{add}, \mathsf{ind}, w) \wedge \nexists\ j > i,\ (\mathsf{op}_j, \mathsf{ind}_j, w_j) = (\mathsf{del}, \mathsf{ind}, w)\}$.
4: $R = \bigcup_{w \in W} R_w$
5: Return R

$\mathsf{FetchDocuments}(R)$

1: Return all documents corresponding to the document identifiers in R.

$\mathsf{Add}(\gamma, \tau^{\mathsf{add}})$

1: $\gamma \leftarrow \Sigma.\mathsf{Add}(\gamma, \tau^{\mathsf{add}})$

$\mathsf{Delete}(\gamma, \tau^{\mathsf{del}})$

1: $\gamma \leftarrow \Sigma.\mathsf{Add}(\gamma, \tau^{\mathsf{del}})$

Operations. During the setup phase, the client generates an extra key K_{Lib}. For add and delete operations, the client performs an extra encryption and addition. For searches, rather than receiving the documents from the server, the client gets the encryptions of all relevant updates. The client decrypts the fetched updates and selects relevant document identifiers by going over the updates linearly.

Communication. In Σ, searches result in communication between client and server regarding the search token and the resulting documents. During searches in Libertas, between sending the search token and receiving the matching documents, client and server exchange additional information. The server sends all updates regarding keywords matching the searched query and the document identifiers of the matching documents. The client, in turn, sends the identifiers of matching documents to the server. This requires an extra round of communications. This can be a problem in specific settings where communication is slow, unstable, expensive, subject to time constraints or otherwise limited. In some cases, round trips can be combined. Suppose that Σ itself is result-hiding and its DecSearch algorithm only requires the client to decrypt an AES encryption for every result. This process can be done in the DecSearch algorithm of Libertas, therefore combining the second rounds of Σ and Libertas, requiring a total of two round trips rather than three.

4.3 Security

Theorem 1. *Let E_{K_Σ} be an IND-CPA secure, which-key concealing encryption scheme and Σ be a wildcard supporting, $\mathcal{L}_\Sigma$-adaptively secure scheme that supports add operations, with $\mathcal{L}_\Sigma = (\mathcal{L}_\Sigma^{\mathsf{Srch}}, \mathcal{L}_\Sigma^{\mathsf{Add}})$ defined as*

$$\begin{aligned}\mathcal{L}_\Sigma^{\mathsf{Srch}}(q) &= \mathcal{L}'(\mathsf{sp}_\Sigma(q), \mathsf{UpHist}_\Sigma(q)),\\ \mathcal{L}_\Sigma^{\mathsf{Add}}(\mathsf{ind}, w) &= \mathcal{L}''(\mathsf{ind}, w),\end{aligned}$$

where $\mathcal{L}'$ and $\mathcal{L}''$ are stateless. Then, Libertas *is $\mathcal{L}_{\mathsf{Lib}}$-adaptively secure, with $\mathcal{L}_{\mathsf{Lib}} = (\mathcal{L}_{\mathsf{Lib}}^{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Lib}}^{\mathsf{Add}}, \mathcal{L}_{\mathsf{Lib}}^{\mathsf{Del}})$ defined as*

$$\begin{aligned}\mathcal{L}_{\mathsf{Lib}}^{\mathsf{Srch}}(q) &= (\mathsf{sp}_{\mathsf{Lib}}(q), \mathsf{TimeDB}_{\mathsf{Lib}}(q), \mathsf{Updates}_{\mathsf{Lib}}(q)),\\ \mathcal{L}_{\mathsf{Lib}}^{\mathsf{Add}}(\mathsf{ind}, w) &= w,\\ \mathcal{L}_{\mathsf{Lib}}^{\mathsf{Del}}(\mathsf{ind}, w) &= w.\end{aligned}$$

Libertas *is therefore update pattern revealing backward-private.*

If Σ is additionally forward private, meaning it is $\mathcal{L}_{\Sigma_{fp}}$-adaptively secure, where $\mathcal{L}_{\Sigma_{fp}} = (\mathcal{L}_\Sigma^{\mathsf{Srch}}, \mathcal{L}_{\Sigma_{fp}}^{\mathsf{Add}})$, with $\mathcal{L}_{\Sigma_{fp}}^{\mathsf{Add}}$ defined with stateless function $\mathcal{L}'''$ as

$$\mathcal{L}_{\Sigma_{fp}}^{\mathsf{Add}}(\mathsf{ind}, w) = \mathcal{L}'''(\mathsf{ind}),$$

and where Libertas *is $\mathcal{L}_{\mathsf{Lib}_{fp}}$-adaptively secure, where $\mathcal{L}_{\mathsf{Lib}_{fp}} = (\mathcal{L}_{\mathsf{Lib}}^{\mathsf{Srch}}, \mathcal{L}_{\mathsf{Lib}_{fp}}^{\mathsf{Add}}, \mathcal{L}_{\mathsf{Lib}_{fp}}^{\mathsf{Del}})$, with $\mathcal{L}_{\mathsf{Lib}_{fp}}^{\mathsf{Add}}$ and $\mathcal{L}_{\mathsf{Lib}_{fp}}^{\mathsf{Del}}$ defined as*

$$\mathcal{L}_{\mathsf{Lib}_{fp}}^{\mathsf{Add}}(\mathsf{ind}, w) = \bot, \qquad \textit{and} \qquad \mathcal{L}_{\mathsf{Lib}_{fp}}^{\mathsf{Del}}(\mathsf{ind}, w) = \bot,$$

meaning Libertas *is forward private as well.*

Proof. We describe a polynomial-time simulator $\mathcal{S}_{\mathsf{Lib}}$ such that for all probabilistic polynomial-time adversaries $\mathcal{A}$, the output of $\mathrm{SSE}_{\mathrm{Real}}^{\mathsf{Lib}}_{\mathcal{A}}(\lambda, n)$ and the output

of $\mathrm{SSE}^{\mathrm{Ideal}}_{\mathcal{A},\mathcal{S}_{\mathsf{Lib}},\mathcal{L}_{\mathsf{Lib}}}(\lambda, n)$ are equal. Since Σ is $\mathcal{L}_\Sigma$-adaptively secure, there exists a polynomial-time simulator $\mathcal{S}_\Sigma$ that can simulate operations in Σ using $\mathcal{L}_\Sigma$. Consider the simulator $\mathcal{S}_{\mathsf{Lib}}$ that adaptively simulates a sequence of n simulated tokens $(\widetilde{\tau}_1, \ldots, \widetilde{\tau}_n)$, a sequence of m simulated encrypted result sets $(\widetilde{R}^*_1, \ldots, \widetilde{R}^*_m)$ and a sequence of m simulated decrypted result sets $(\widetilde{R}_1, \ldots, \widetilde{R}_m)$, where $m \leq n$, as follows:

- (Setup) the simulator generates a random key $K_{\mathcal{S}_{\mathsf{Lib}}}$.
- (Simulating τ^{srch}) given $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{Lib}}(q) = (\mathsf{sp}_{\mathsf{Lib}}(q), \mathsf{TimeDB}_{\mathsf{Lib}}(q), \mathsf{Updates}_{\mathsf{Lib}}(q))$, construct $\widetilde{\mathcal{L}}^{\mathsf{Srch}}_\Sigma(q) = \mathcal{L}(\widetilde{\mathsf{sp}}_\Sigma(q), \widetilde{\mathsf{UpHist}}_\Sigma(q))$ with $\widetilde{\mathsf{sp}}_\Sigma(q) = \mathsf{sp}_{\mathsf{Lib}}(q)$ and $\widetilde{\mathsf{UpHist}}_\Sigma(q) = \{(u, \mathsf{add}, E_{K_{\mathcal{S}_{\mathsf{Lib}}}}(\perp_c, \perp_{\mathsf{op}}, \perp_{\mathsf{ind}}, \perp_w)) \mid u \in \mathsf{Updates}_{\mathsf{Lib}}(q)\}$.
  Then, rather than running $\Sigma.\mathsf{SrchToken}(K_\Sigma, q)$, run $\mathcal{S}_\Sigma(\mathsf{st}_{\mathcal{S}_\Sigma}, \widetilde{\mathcal{L}}^{\mathsf{Srch}}_\Sigma(q))$. Since every search for query q in $\mathsf{Libertas}$ results in a search for query q in Σ, the search patterns for $\mathsf{Libertas}$ and Σ are identical. $\mathsf{UpHist}_\Sigma(q)$ can be generated as the timestamps are identical to those of $\mathsf{Updates}_{\mathsf{Lib}}(q)$, the operation is always add and the encryption of meaningless data is indistinguishable from that of meaningful data, since E is IND-CPA secure. $(\perp_c, \perp_{\mathsf{op}}, \perp_{\mathsf{ind}}, \perp_w)$ are generated based on u, maintaining consistency between simulated search tokens of identical queries. By taking constructed leakage $\widetilde{\mathcal{L}}^{\mathsf{Srch}}_\Sigma$ as input, $\mathcal{S}_\Sigma$, and in turn $\mathcal{S}_{\mathsf{Lib}}$, can simulate search tokens $\widetilde{\tau}^{\mathsf{srch}}$ that are indistinguishable from real tokens τ^{srch}.
- (Simulating τ^{add}) given $\mathcal{L}^{\mathsf{Add}}_{\mathsf{Lib}}(\mathsf{ind}, w) = w$, construct $\widetilde{\mathcal{L}}^{\mathsf{Add}}_\Sigma(\mathsf{ind}, w) = \mathcal{L}(\widetilde{\mathsf{ind}}, \widetilde{w})$ with $\widetilde{\mathsf{ind}} = E_{K_{\mathcal{S}_{\mathsf{Lib}}}}(\perp_c, \perp_{\mathsf{op}}, \perp_{\mathsf{ind}}, \perp_w)$, and $\widetilde{w} = w$.
  Then, instead of running algorithm $\Sigma.\mathsf{AddToken}(K_\Sigma, E_{K_{\mathsf{Lib}}}(c, \mathsf{add}, \mathsf{ind}, w))$, run $\mathcal{S}_\Sigma(\mathsf{st}_{\mathcal{S}_\Sigma}, \widetilde{\mathcal{L}}^{\mathsf{Add}}_\Sigma(\mathsf{ind}, w))$. To clarify, $\widetilde{\mathsf{ind}}$ is viewed as a document identifier from Σ's perspective, but as an encrypted tuple from $\mathsf{Libertas}$'s perspective. Since $E_{K_{\mathcal{S}_{\mathsf{Lib}}}}$ is CPA-secure, $\perp_c$, $\perp_{\mathsf{op}}$, $\perp_{\mathsf{ind}}$ and $\perp_w$ can be anything, as the resulting encryption will be indistinguishable from an encryption where an actual timestamp, update operation, document identifier and keyword are considered. Therefore, $\mathcal{S}_\Sigma$, and in turn $\mathsf{Libertas}$, will be able to create add tokens $\widetilde{\tau}^{\mathsf{add}}$ that are indistinguishable from real tokens τ^{add}. We do not maintain consistency for add tokens as we did for search tokens, as add tokens are distinct by nature.
  In case Σ is forward private, we are given $\mathcal{L}^{\mathsf{Add}}_{\mathsf{Lib}_{fp}}(\mathsf{ind}, w) = \perp$ and we construct $\widetilde{\mathcal{L}}^{\mathsf{Add}}_{\Sigma_{fp}}(\mathsf{ind}, w) = \mathcal{L}(\widetilde{\mathsf{ind}})$ with $\widetilde{\mathsf{ind}} = E_{K_{\mathcal{S}_{\mathsf{Lib}}}}(\perp_c, \perp_{\mathsf{op}}, \perp_{\mathsf{ind}}, \perp_w)$.
- (Simulating τ^{del}) $\mathcal{S}_{\mathsf{Lib}}$ can construct a delete token $\widetilde{\tau}^{\mathsf{del}}$ that is indistinguishable from τ^{del} in the same way as it constructs add tokens.
- (Simulating R^*) given $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{Lib}}(q) = (\mathsf{sp}_{\mathsf{Lib}}(q), \mathsf{TimeDB}_{\mathsf{Lib}}(q), \mathsf{Updates}_{\mathsf{Lib}}(q))$, construct $\widetilde{R}^* = \{E_{K_{\mathcal{S}_{\mathsf{Lib}}}}(\perp_c, \perp_{\mathsf{op}}, \perp_{\mathsf{ind}}, \perp_w) \mid u \in \mathsf{Updates}_{\mathsf{Lib}}(q)\}$, where $\perp_c$ is a fake timestamp, $\perp_{\mathsf{op}}$ is a fake update operation, $\perp_{\mathsf{ind}}$ is a fake document identifier and $\perp_w$ is a fake keyword. Since $E_{K_{\mathcal{S}_{\mathsf{Lib}}}}$ is IND-CPA secure, items in R^* and $\widetilde{R}^*$ are indistinguishable. As both result sets have the same length as well, R^* and $\widetilde{R}^*$ are indistinguishable. To maintain consistency of simulated

sets between identical search queries, we generate values $(\perp_c, \perp_{\mathsf{op}}, \perp_{\mathsf{ind}}, \perp_w)$ based on u, akin to what we did for simulating search tokens.
– (Simulating R) given $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{Lib}}(q) = (\mathsf{sp}_{\mathsf{Lib}}(q), \mathsf{TimeDB}_{\mathsf{Lib}}(q), \mathsf{Updates}_{\mathsf{Lib}}(q))$, construct $\widetilde{R} = \{\mathsf{ind} \mid (u, \mathsf{ind}) \in \mathsf{TimeDB}_{\mathsf{Lib}}(q)\}$.

5 Evaluation

In order to empirically evaluate the cost of BP in Libertas, we implemented it with the wildcard supporting scheme by Zhao and Nishide (Z&N) [20] that stores one Bloom filter [2] for each document-keyword pair in the index. An overview of the scheme's algorithms, including the generation of the Bloom filters, can be found in Appendix A. Z&N is forward private and allows for updates on a document-keyword pair level rather than considering complete documents, making integration with Libertas easy. Also, it supports two wildcard types, allowing for greater query flexibility.

5.1 Setup

Implementation. A single-core implementation is written and tested in Python 3.8 with code available at https://github.com/LibertasConstruction/Libertas.

Hardware. The experiments were carried out on a laptop computer running Windows 10 with 8 GB of RAM and 4 Intel i7-4700MQ cores, operating at 2.4 GHz each. The implementation only used a single CPU core, however. Both the scheme's client and server ran in the same process, communicating directly via the Python script.

Parameters. We set the false positive rate of the Bloom filters to 0.01 and used keywords of length 5. The length of the keyword determines the size of the keyword characteristic set and thus the number of elements in the Bloom filter. With these settings, Bloom filters consist of 240 bits and use 7 hash functions. We used 2048 bit keys for all Z&N instances and 256 bit keys for AES encryptions in Libertas.

Dataset. For the experiments, we generated document-keyword pairs of the form $[(0, \text{'00000'}), (1, \text{'00001'}), \ldots, (99999, \text{'99999'})]$.

5.2 Experiments

We devised four experiments that measure the effect of changes to the index size, the wildcard query, the result set and the number of deletions, respectively. We measured the execution time of the search protocol of both schemes, averaged over 10 queries and 10 instances of the schemes. We considered the Search operation for Z&N and both the Search and DecSearch operations for $\mathsf{Libertas}_{\mathsf{Z\&N}}$. We disregarded the SrchToken operation as it is identical for both schemes.

Table 1. Avg. search times in *seconds* for (a) exact keyword search, (b) wildcard search with index size 10^4, and (c) varying result set size with index size 10^4.

	(a) Index size				(b) # of wildcards					(c) Result set size				
Scheme	10^2	10^3	10^4	10^5	0	1	2	3	4	10^0	10^1	10^2	10^3	10^4
Z&N	0.02	0.11	0.67	2.90	0.67	0.69	0.76	1.02	2.39	0.63	0.63	0.74	1.91	14.34
$\text{Libertas}_{\text{Z\&N}}$	0.02	0.11	0.67	2.91	0.66	0.69	0.76	1.05	2.78	0.63	0.63	0.74	1.93	15.34

Table 2. Avg. search times in *seconds* per number of deletions (index size 10^4).

	Number of deletions										
Scheme	0	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Z&N	0.65	0.57	0.51	0.44	0.38	0.32	0.25	0.19	0.12	0.06	0.00
$\text{Libertas}_{\text{Z\&N}}$	0.63	0.70	0.76	0.81	0.88	0.95	1.01	1.07	1.12	1.20	1.24

Basic Search. To measure the basic search time, we inserted the first n_i pairs of the generated data set for different index sizes n_i. We measured the search time of a random keyword present in the index.

Wildcard Query Search. To measure the effect of wildcards, we considered a fixed index size of 10,000 but increasingly replaced more query characters with '_' wildcards, to increase the number of matching keywords. We chose not to include '*' wildcards, as the construction of Z&N uses the same concept for both wildcard types. While there is a measurable performance difference depending on the wildcard type, this effect will be identical for Z&N and $\text{Libertas}_{\text{Z\&N}}$. We are only interested in the number of matching keywords as this influences the performance of the DecSearch operation in Libertas.

Varying Result Set Size. We investigated the effect of matching multiple documents. The generated data set is modified slightly for this experiment. The last n_r pairs that are inserted consider the same keyword. This is the keyword we query for. We measured the search time for increasing n_r, with a fixed index size of 10,000.

Varying Number of Deletions. To evaluate the effect of deletions growing the index of Libertas, we measured search times for an increasing number of deletions. For this experiment, both schemes started out with their index containing the first 10,000 pairs of the generated data set. Then, we deleted pairs from the index using the delete protocol of the scheme.

5.3 Results

Tables 1 and 2 summarize our experimental results.

Basic Search. For exact keyword searches, Table 1(a) shows that $\text{Libertas}_{\text{Z\&N}}$ experiences no overhead compared to Z&N, regardless of the index size.

Wildcard Query Search. Table 1(b) shows that the overhead of $\mathsf{Libertas}_{\mathsf{Z\&N}}$ barely increases when considering queries containing wildcards such that they match multiple keywords. Note that, for the given dataset, every additional wildcard increases the number of matching keywords ten-fold.

Varying Result Set Size. Table 1(c) indicates that $\mathsf{Libertas}_{\mathsf{Z\&N}}$ and Z&N have a comparable performance regardless of the result set size.

Varying Number of Deletions. Table 2 shows the downside of an index that grows with deletions. Typically, search times decrease as items are deleted, as can be seen for Z&N. Due to Libertas' nature, however, its index increases, slowing down searches linearly with the number of deletions.

6 Discussion

Query Similarity. The wildcard leakage functions we introduced in Sect. 3.1 allow for query similarity leakage. We consider Definition 4. Here, information on query similarity is leaked in the following way. We consider queries q_1 and q_2. If $q_2 \dot{\subseteq} q_1$, then $\mathsf{TimeDB}(q_2) \subseteq \mathsf{TimeDB}(q_1)$. Note that the relation is not reversible; if an observer sees that $\mathsf{TimeDB}(q_2) \subseteq \mathsf{TimeDB}(q_1)$, it does not necessarily mean that $q_2 \dot{\subseteq} q_1$. An adversary can try to link result sets that are subsets of each other and assume that the corresponding queries are related; the query corresponding to the larger result set is likely a more general form of the query of the smaller set. This query similarity leakage might be abusable and compromise wildcard security. We leave it for future work to determine if this leakage undermines wildcard security and if so, to develop an LAA.

Real-World Application. $\mathsf{Libertas}_{\mathsf{Z\&N}}$ is ready for deployment in systems that require a backward private, wildcard supporting DSSE scheme today. The implementation provided with this paper uses a single CPU core. The implementation can easily be parallelized, however. During the Search algorithm, the server goes through all updates in the index (see line 2–3 in Search in Algorithm 2). This search can be split up between cores. If we assume a computer with 8 CPU cores, we can effectively cut search times by a factor of 8. Searches will take less than a second even with a large index or many deletions. Only when considering very large databases or environments where two round trips are undesirable would Libertas not provide a proper solution.

Clean-Up Procedure. The major drawback of Libertas is that its index grows with every update, as deletions in Libertas translate to insertions in Σ. This increases search times for both the Search algorithm run at the server and the DecSearch algorithm run at the client. We propose a clean-up procedure similar to that of Bost et al. [5] to combat this problem. During Search, the server removes all results from the index. Then, when running the FetchDocuments algorithm, the client additionally runs the AddToken algorithm for every relevant document-keyword pair. That is, every pair that was added, but not subsequently deleted. The server runs the Add algorithm to re-add the relevant document-keyword pairs to the index. This procedure cleanses the index during searches,

removing updates that cancel each other out. If Libertas is constructed from a forward private scheme, we believe this procedure incurs no additional leakage, as additions do not leak information. This clean-up procedure restricts the choice of Σ, as the scheme should be able to remove individual entries from the index. A common example of a valid index structure is a list containing entries for every document-keyword pair, such as in Z&N. An example of a DSSE scheme with an unsuitable index structure is the scheme by Kamara et al. [14].

Insertion Pattern Revealing Backward Privacy. Libertas can achieve *insertion pattern revealing backward privacy* if Σ is forward private and does not leak $\mathsf{UpHist}_{\Sigma}(q)$ during search operations, but only $\mathsf{ap}_{\Sigma}(q)$. In our scenario, the difference between leakage functions $\mathsf{UpHist}_{\Sigma}(q)$ and $\mathsf{ap}_{\Sigma}(q)$ consists of only the timestamps of all updates, as update operations are always additions. If Σ does not leak these timestamps, then Libertas does neither. In the proof, rather than using $\mathsf{Updates}_{\mathsf{Lib}}(q)$ to construct $\mathsf{UpHist}_{\Sigma}(q)$, we can use $a_{q_{\mathsf{Lib}}}$ to construct $\mathsf{ap}_{\Sigma}(q)$ by generating $a_{q_{\mathsf{Lib}}}$ encryptions of $(\perp_c, \perp_{\mathsf{op}}, \perp_{\mathsf{ind}}, \perp_w)$ tuples. Hiding $\mathsf{UpHist}(q)$ in SSE schemes remains a challenge, however. Current solutions use ORAM but are not efficient [15].

7 Conclusion

In this research, we extended commonly used leakage functions and, in turn, backward privacy definitions, to consider wildcard queries as opposed to just exact keyword queries. We presented Libertas; a construction providing *update pattern revealing backward privacy* to any wildcard supporting scheme Σ. We proved the security of Libertas in the $\mathcal{L}$*-adaptive security model* and evaluated its performance compared to its underlying scheme Σ. We found that Libertas experiences an overhead that is linear in the number of deletions. The resulting scheme requires an additional round of communication during searches and its index grows with every update. Nonetheless, searches are fast, making Libertas suitable for real-world applications.

A Z&N: Construction

We provide the construction of Z&N: a DSSE scheme supporting wildcard search. It is proposed by Zhao and Nishide in [20] and uses Bloom filters [2] and a regular index. The algorithms are described in Algorithm 2. An implementation of Z&N can be found at https://github.com/LibertasConstruction/Libertas. The scheme uses a hash function g. $\overline{g}$ denotes the first bit of a hash using g. The scheme uses g with r different keys to effectively create r different hash functions to use for Bloom filters. $\mathsf{BF}[p]$ denotes the bit in a Bloom filter at position p. $\mathsf{ind} \parallel w$ indicates a concatenation of ind and w. The scheme uses keyword characteristic and token (query) characteristic sets, $S_K(w)$ and $S_T(q)$, to capture the structure of keywords and queries to support both '*' and '_' wildcard symbols. Every keyword characteristic set is stored in a Bloom filter.

A.1 Keyword Characteristic Set

$S_K(w)$ is made up of the two sets $S_K^{(o)}(w)$ and $S_K^{(p)}(w)$. The set $S_K^{(o)}(w)$ contains characters of a keyword w together with their position. For example, $S_K^{(o)}(\text{'diana'}) = \{\text{'1:d'}, \text{'2:i'}, \text{'3:a'}, \text{'4:n'}, \text{'5:a'}, \text{'6:\0'}\}$. Note the terminator symbol indicating the end of the keyword. The set $S_K^{(p)}(w)$ consists of the sets $S_K^{(p1)}(w)$ and $S_K^{(p2)}(w)$. These sets consider pairs of characters. Let us take a look at these sets when using the keyword 'diana'.

$$S_K^{(p1)}(\text{'diana'}) = \{\text{'1:1:d,i'}, \text{'2:1:d,a'}, \text{'3:1:d,n'}, \text{'4:1:d,a'}, \text{'5:1:d,\0'},$$
$$\text{'1:1:i,a'}, \text{'2:1:i,n'}, \text{'3:1:i,a'}, \text{'4:1:i,\0'},$$
$$\text{'1:1:a,n'}, \text{'2:1:a,a'}, \text{'3:1:a,\0'},$$
$$\text{'1:1:n,a'}, \text{'2:1:n,\0'},$$
$$\text{'1:1:a,\0'}\}$$

Here, the element '3:1:d,n' comes from the character pair 'diana', where 3 is the distance between the characters and 1 indicates that it is the first occurrence of the pair with the given distance in this set.

$$S_K^{(p2)}(\text{'diana'}) = \{\text{'-:1:d,i'}, \text{'-:1:d,a'}, \text{'-:1:d,n'}, \text{'-:2:d,a'}, \text{'-:1:d,\0'},$$
$$\text{'-:1:i,a'}, \text{'-:1:i,n'}, \text{'-:2:i,a'}, \text{'-:1:i,\0'},$$
$$\text{'-:1:a,n'}, \text{'-:1:a,a'}, \text{'-:1:a,\0'},$$
$$\text{'-:1:n,a'}, \text{'-:1:n,\0'},$$
$$\text{'-:2:a,\0'}\}$$

Here, the element '-:2:i,a' comes from the character pair 'diana'. Distances are not considered in this set. The 2 indicates that this is the second occurrence of the pair in the set.

A.2 Token Characteristic Set

Next, we will show how to construct the token characteristic set $S_T(q)$ of a search query q. As this scheme does not support conjunctive keyword queries, q can be thought of as a keyword containing wildcards. Similar to $S_K(w)$, $S_T(q)$ is made up of the sets $S_T^{(o)}(q)$, $S_T^{(p1)}(q)$ and $S_T^{(p2)}(q)$. The construction of the sets is illustrated by an example with the query 'di*a_a*\0'.

The set $S_T^{(o)}(q)$ is constructed by extracting characters from q with a specified appearance order. $S_T^{(o)}(\text{'di*a_a*\0'}) = \{\text{'1:d'}, \text{'2:i'}\}$.

We define a *character group* as a group of subsequent characters that do not contain wildcards. 'di*a_a*\0' consists of the character groups 'di', 'a', 'a' and '\0'. For $S_T^{(p1)}(q)$, we consider the character group to the left and to the right of '_' wildcards. We generate all possible character pairs with their corresponding distance. Then, we do mostly the same for '*' wildcards: we consider

Algorithm 2. Z&N

$\mathsf{Setup}(\lambda)$

1: $k_t \xleftarrow{\$} \{0,1\}^\lambda$, for $t \in [1,r]$
2: $K_H = \{k_t\}_{t\in[1,r]}$
3: $K_G \xleftarrow{\$} \{0,1\}^\lambda$
4: $K = (K_H, K_G)$

$\mathsf{BuildIndex}(\lambda)$

1: $\gamma \leftarrow$ empty list

$\mathsf{SrchToken}(K, q)$

1: $S_{T_q} \leftarrow S_T(q)$
2: For each element e_j of S_{T_q}:
3: $\quad p_t \leftarrow g(k_t, e_j)$, for $t \in [1,r]$
4: $\quad \tau^{\mathsf{srch}}_{e_j,1} = (p_1, p_2, \ldots, p_r)$
5: $\quad \tau^{\mathsf{srch}}_{e_j,2} = (g(K_G, p_1), g(K_G, p_2), \ldots, g(K_G, p_r))$
6: $\quad \tau^{\mathsf{srch}}_{e_j} = (\tau^{\mathsf{srch}}_{e_j,1}, \tau^{\mathsf{srch}}_{e_j,2})$
7: $\tau^{\mathsf{srch}} = (\tau^{\mathsf{srch}}_{e_1}, \tau^{\mathsf{srch}}_{e_2}, \ldots, \tau^{\mathsf{srch}}_{e_\ell})$
8: Return τ^{srch}

$\mathsf{AddToken}(K, \mathsf{ind}, w)$

1: $b_{\mathsf{id}} \leftarrow g(K_G, \mathsf{ind} \parallel w)$
2: $S_{K_w} \leftarrow S_K(w)$
3: For each element e_j of $S_K(w)$:
4: $\quad p_t \leftarrow g(k_t, e_j)$, for $t \in [1,r]$
5: $\quad$ Initialize a Bloom filter BF of length b and set the bits at positions p_t to 1
6: For $p \in [1,b]$:
7: $\quad \mathsf{mb}[p] \leftarrow \overline{g}(b_{\mathsf{id}}, g(K_G, p))$
8: $\quad \mathsf{BF}[p] \leftarrow \mathsf{BF}[p] \oplus \mathsf{mb}[p]$
9: $\tau^{\mathsf{add}} = (\mathsf{ind}, \mathsf{BF}, b_{\mathsf{id}})$
10: Return τ^{add}

$\mathsf{DelToken}(K, \mathsf{ind}, w)$

1: $b_{\mathsf{id}} \leftarrow g(K_G, \mathsf{ind} \parallel w)$
2: $\tau^{\mathsf{del}} = b_{\mathsf{id}}$
3: Return τ^{del}

$\mathsf{Search}(\gamma, \tau^{\mathsf{srch}})$

1: τ^{srch} consists of Bloom filter positions and hashes of these positions. We arrange these as $((p_1, g(K_G, p_1)), \ldots, (p_i, g(K_G, p_i)))$
2: For all $(\mathsf{ind}, \mathsf{BF}, b_{\mathsf{id}})$ in γ:
3: $\quad$ Add ind to R if $\mathsf{BF}[p_t] \oplus \overline{g}(b_{\mathsf{id}}, g(K_G, p_t)) = 1$ for all $(p_t, g(K_G, p_t))$, where $t \in [1,i]$.
4: Return R

$\mathsf{Add}(\gamma, \tau^{\mathsf{add}})$

1: Add τ^{add} to γ

$\mathsf{Delete}(\gamma, \tau^{\mathsf{del}})$

1: $\tau^{\mathsf{del}} = b_{\mathsf{id}}$
2: Remove from γ the entry with Bloom filter identifier b_{id}

the character group left and right of the '*' wildcard. This time, however, we *concatenate* the character groups before generating the character pairs, thereby ignoring the wildcard itself in the distance computation. The resulting pairs are added to $S_T^{(p1)}$. The following example illustrates what this means exactly. Consider $S_T^{(p1)}$('di*a_a*\0'). The '_' wildcard is surrounded by 'a' and 'a'. $S_T^{(p1)}$ therefore contains '2:1:a,a'. The first '*' wildcard is surrounded by character group 'di' and character 'a', adding '1:1:d,i', '2:1:d,a' and '1:1:i,a' to the set. In the same fashion, '1:1:a,\0' is added.

To construct the set $S_T^{(p2)}(q)$, consider the search string without wildcard symbols. Then, follow the same procedure as with the construction of $S_K^{(p2)}(w)$.

References

1. Abadi, M., Rogaway, P.: Reconciling two views of cryptography. In: van Leeuwen, J., Watanabe, O., Hagiya, M., Mosses, P.D., Ito, T. (eds.) TCS 2000. LNCS, vol. 1872, pp. 3–22. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44929-9_1
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**, 422–426 (1970)
3. Bösch, C., Brinkman, R., Hartel, P., Jonker, W.: Conjunctive wildcard search over encrypted data. In: Jonker, W., Petković, M. (eds.) SDM 2011. LNCS, vol. 6933, pp. 114–127. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23556-6_8
4. Bost, R.: $\sum o\varphi o\varsigma$: forward secure searchable encryption. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2016)
5. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2017)
6. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Proceedings of the ACM SIGSAC Conference On Computer and Communications Security (2015)
7. Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for Boolean queries. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 353–373. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_20
8. Chase, M., Shen, E.: Substring-searchable symmetric encryption. In: Proceedings on Privacy Enhancing Technologies (2015)
9. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2006)
10. Faber, S., Jarecki, S., Krawczyk, H., Nguyen, Q., Rosu, M., Steiner, M.: Rich queries on encrypted data: beyond exact matches. In: Pernul, G., Ryan, P.Y.A., Weippl, E. (eds.) ESORICS 2015. LNCS, vol. 9327, pp. 123–145. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24177-7_7
11. Goh, E.J.: Secure indexes. IACR Cryptol. ePrint Arch. (2003). https://ia.cr/2003/216
12. Hu, C., Han, L.: Efficient wildcard search over encrypted data. Int. J. Inf. Secur. **15**(5), 539–547 (2015). https://doi.org/10.1007/s10207-015-0302-0

13. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In: NDSS (2012)
14. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (2012)
15. Naveed, M.: The fallacy of composition of oblivious RAM and searchable encryption. IACR Cryptol. ePrint Arch (2015). https://ia.cr/2015/668
16. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of the IEEE Symposium on Security & Privacy. IEEE (2000)
17. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: Proceedings of the Network and Distributed System Security Symposium (2014)
18. Suga, T., Nishide, T., Sakurai, K.: Secure keyword search using bloom filter with specified character positions. In: Takagi, T., Wang, G., Qin, Z., Jiang, S., Yu, Y. (eds.) ProvSec 2012. LNCS, vol. 7496, pp. 235–252. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33272-2_15
19. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: the power of file-injection attacks on searchable encryption. In: Proceedings of the USENIX Security Symposium (2016)
20. Zhao, F., Nishide, T.: Searchable symmetric encryption supporting queries with multiple-character wildcards. In: Chen, J., Piuri, V., Su, C., Yung, M. (eds.) NSS 2016. LNCS, vol. 9955, pp. 266–282. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46298-1_18