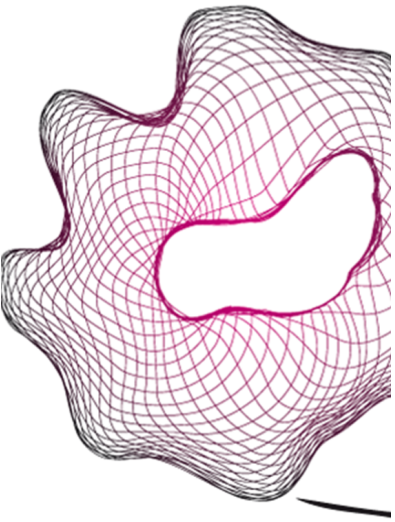
UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**



Libertas: A Backward Private Dynamic Searchable Symmetric Encryption Scheme Supporting Wildcard Search

Jeroen Weener

M.Sc. Thesis
October 2021

Supervisors:
prof.dr. A. Peter
dr.ing. F.W. Hahn

External Committee:
prof.dr.ir. R.M. van Rijswijk-Deij

Services and CyberSecurity
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

ABSTRACT

When outsourcing data, *Searchable Symmetric Encryption* schemes allow 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 scheme. If the scheme is forward private, this property is inherited. We prove security in the $\mathcal{L}$-adaptive security model. We show that the performance overhead scales linearly with the number of deletions.

CCS CONCEPTS

• **Security and privacy → Security protocols**; **Management and querying of encrypted data**.

KEYWORDS

Searchable Encryption, Backward Privacy, Information Leakage, Wildcard Search

1 INTRODUCTION

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 introduced by Song et al. [26], allowing clients to search through a static database of encrypted documents. Later, dynamic SSE (DSSE) schemes have been proposed [21], allowing clients to add and delete data after the scheme's initialization. Non-adaptive and adaptive security definitions for SSE schemes have been defined by Curtmola et al. [15]. Kamara et al. [21] define adaptive security for DSSE schemes. Search queries and updates potentially leak information such as the matching documents or the affected keywords, respectively. As shown by previous lines of research, despite a scheme conforming to the aforementioned security definitions, this information leakage can allow for powerful attacks. Islam et al. [20] propose a passive attack where knowledge of document contents is combined with statistical techniques to recover the content of search queries. Cash et al. [9] propose both passive and active attacks to recover search query content and plaintext content. Zhang et al. [32] describe an active attack where search queries are revealed after injecting few files. To defend against adaptive file injection attacks, new DSSE schemes featuring *forward privacy* have been proposed by Stefanov et al. [27] and Bost et al. [5]. Forward privacy ensures that newly added data cannot be linked to earlier search queries. Forward privacy does not protect against non-adaptive file injection attacks. *Backward privacy* is another security notion that has been proposed by Bost et al. [6]. In backward private schemes, search queries cannot be executed over deleted entries, limiting the potential of (future) attacks. As full backward privacy cannot yet be efficiently achieved, three levels of backward privacy are introduced. The first level is the most secure and leaks the least amount of information. Subsequent levels increase allowed leakage, reducing security. To allow for flexible searches, a variety of query expressiveness extensions have been proposed. One of such extensions is the support for wildcards, where search queries such as 'c_t' can match data containing both 'cat' and 'cut' [8], [16], [33]. Despite the advancements in flexible search queries, most forward and backward private schemes only consider exact keyword search. This paper introduces Libertas: a construction for providing the second level of backward privacy to any wildcard supporting DSSE scheme. It is proven secure against adaptive adversaries. We provide an open-source implementation and evaluate its performance. Our results show that Libertas' search performance overhead is hardly effected 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.

2 RELATED WORK

2.1 Searchable Encryption

SSE schemes were first explored by Song, Wagner and Perrig [26]. The actors of an SSE scheme are the client and server. The server hosts data of the client in encrypted form. The client can search the data for keywords and retrieve relevant data from the server, all without revealing to the server the searched keyword or the content of the data. The stored data is often referred to as documents. The searchable content of documents are called keywords. Despite these naming conventions, SSE schemes often apply to many other forms of data such as emails or DNA genomes [31]. Goh et al. introduce the concept of an *index* to speed up searches [17]. An index is a data structure where keyword identifiers are stored per document identifier. Searches use the index to find the matching document identifiers instead of using the documents themselves. Therefore, by using an index, schemes become indifferent to the cryptographic cipher used for encrypting documents. Most SSE schemes make use of an *inverted index*, first described by Curtmola et al. [15]. Rather than storing keyword identifiers per document identifier, inverted indices store document identifiers per keyword identifier. Standard SSE schemes index data upon initialization and do not allow updates to the index afterwards. *Dynamic SSE* (DSSE) schemes do allow documents to be added and removed. Depending on the implementation, clients can add or remove entire documents, or they perform updates per document-keyword pair. This second approach allows for more fine-grained control over the data but requires the client to send multiple updates if they want to add or

remove an entire document. The first practical DSSE scheme has been proposed by Kamara et al. [21].

2.2 SSE Security and Attacks

Non-adaptive and adaptive security definitions for SSE schemes have been defined by Curtmola et al. [15]. Kamara et al. [21] define adaptive security for DSSE schemes. In adaptive secure schemes, as opposed to non-adaptive secure schemes, adversaries take into account the results of previous interactions with the scheme. Despite adhering to these security definitions, (D)SSE schemes are at risk of attacks. Leakage abuse attacks (LAAs) leverage the leakage of search and update operations of schemes to mount, recovering search query or document contents. Islam et al. describe the first query recovery attack [20]. They show how an adversary with full background knowledge regarding the stored content can determine the keyword hidden in search queries. The passive attack works on any SSE scheme that leaks the access pattern. By observing the search queries sent by the client and the subsequent document identifiers sent by the server, their model is able to infer the queried keyword with high accuracy. Cash et al. extend this work by describing several LAAs, both passive and active [9]. They improve the attack of Islam et al. by requiring only partial knowledge of the stored content. Additionally, they describe plaintext recovery attacks; attacks that aim to recover the content of the stored documents. For these attacks, the adversary requires knowledge of some documents or the ability to inject documents. Zhang et al. [32] describe efficient file-injection attacks aiming to recover keywords from search queries, assuming little knowledge of stored content. They provide an adaptive and non-adaptive version of the attack. The adaptive attack requires less injected files and achieves a higher query recovery rate compared to its non-adaptive counterpart. File injection can be simple depending on the environment of the scheme. For example, if a scheme is used to store email, files can be injected by simply sending an email to the client. To defend against query recovery attacks, the security notion *forward privacy* has been informally defined by Stefanov et al. [27] and is defined formally by Bost et al. [5]. Forward private schemes do not leak which keywords are considered during updates, making it impossible to link newly added data to earlier search queries. This serves as a countermeasure to file-injection attacks. Forward privacy does not fully protect the scheme against these attacks, however, as search queries can still be recovered if documents are injected prior to the query. Bost et al. introduce *backward privacy* as another security notion for DSSE schemes [6]. In backward private schemes, search queries cannot be executed over deleted entries, limiting the potential of (future) attacks. Full backward privacy requires hiding the update pattern, which consists of the timestamps of all updates. Currently, the only way to achieve this is by using ORAM, leading to schemes that do not scale well [23]. Therefore, Bost et al. introduce three weakened levels of backward privacy. The first level is the most secure and leaks the least amount of information. Subsequent levels increase allowed leakage, reducing security. Bost et al. describe $B(\Sigma)$ and $B'(\Sigma)$: constructions for building a two-round backward private scheme from a DSSE scheme Σ, both achieving the second degree of backward privacy. They additionally define Janus: a single-round backward private scheme achieving the lowest degree of backward privacy.

2.3 Query Expressiveness

To allow for more query flexibility, several extensions to the basic single keyword search have been proposed. *Conjunctive queries* allow the client to search documents for multiple keywords. Conjunctive queries can be considered boolean expressions of keywords connected by conjunction operators. *Boolean queries* extend conjunctive queries by allowing different kinds of boolean operators such as negations and disjunctions. Cash et al. describe a scheme featuring boolean queries [10]. *Comparison queries* and *range queries* allow one to search numerical data. Bethencourt et al. propose a scheme allowing for range queries [2]. Boneh and Waters introduce a scheme supporting both comparison and range queries [4]. *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 describe a substring supporting scheme [11]. *Fuzzy queries* allow for keywords to match with queries if they are within a specific edit distance. *Wildcard queries* allow the client to insert wildcard, or joker, characters in the search query. The type of wildcard differs per scheme. 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. consider Bloom filters in the multi-client setting, allowing for substring, fuzzy and wildcard queries [28]. Hu et al. introduce a scheme that is more efficient compared to Suga et al. and allows clients to update the database [18], [19]. The scheme by Bösch et al. operates in the dynamic single-user environment. Here, wildcard support is implemented naively by generating and inserting all wildcard variants of a keyword upon database insertion [8]. 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 describe a wildcard supporting scheme capable of supporting two types of wildcards by cleverly storing keyword characteristics in Bloom filters [33]. Saha and Koshiba [24] and Yasuda et al. [31] propose packing methods for secure pattern matching using Learning With Errors (LWE). Their methods can be combined with the single-user and multi-user schemes defined in Brakerski and Vaikuntanathan [7] to construct wildcard supporting schemes. Faber et al. [16] propose a matching algorithm that can operate in both a single and multi-user environment based on the conjunctive search scheme by Cash et al. [10]. Their scheme supports substring, phrase, range and wildcard queries, and allows any combination of these query types using boolean operators. *Phrase queries* are the sentence equivalent of wildcard queries. Rather than considering a word and allowing for joker characters, phrase queries consider a sequence of words and allow one to leave out one or multiple words, depending on the implementation. Other multi-user wildcard supporting schemes are proposed by Wang et al. [29], Yang et al. [30] and Sedghi et al. [25]. Wang et al. propose a scheme without an index based on bilinear pairings. Instead,

the scheme outputs searchable ciphertext. The scheme by Yang et al. supports user authorization and revocation. Their scheme features seven matching algorithms based on secure multi-party computation (MPC), allowing for a maximum of two wildcards in a query. The scheme by Sedghi et al. makes use of public-key hidden vector encryption (HVE). Chung et al. use common-conditioned-subsequence-preserving (CCSP) techniques to define the schemes FETCH and uFETCH: database-ready schemes with a sub-linear search complexity [13], [14]. Both papers lack security proofs for the proposed schemes, however. Kim et al. present the first scheme supporting three wildcard types [22]. The scheme makes use of fully homomorphic encryption (FHE). In their evaluation, however, they find the efficiency to be underwhelming for real-world applications. More recently, Chatterjee et al. constructed an SSE scheme also supporting three wildcard types [12]. Their scheme comes with a sub-linear search time in the three-party OSPIR setting.

3 PRELIMINARIES

3.1 SSE Schemes

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 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.

3.1.1 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. [8]. Results are hidden by altering the **Search** protocol, adding new algorithms DecSearch and FetchDocuments. 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, 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 **Search** protocol, document identifiers are first revealed to the client rather than the server. The sequence diagram of the extended **Search** protocol is depicted in Figure 1.

3.2 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,

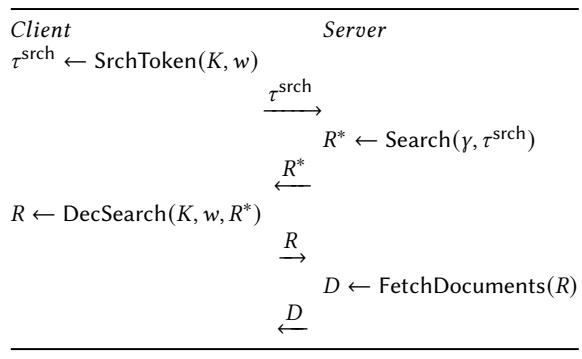


Figure 1: Sequence diagram of the Search protocol in a result-hiding SSE scheme

where u is the timestamp of the operation and w is the searched keyword. Update operations append $(u, \text{op}, (\text{ind}, w))$ 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.

$$\text{sp}(w) = \{u \mid (u, w) \in Q\},$$

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

The search pattern $\text{sp}(w)$ 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 $\text{ap}(w)$ leaks the document identifiers ind of documents that contain keyword w at the time of the search.

3.3 Security Model

The security model for SSE schemes often considered in the literature is called $\mathcal{L}$*-adaptive security* [15]. 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 2 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 section 5.3.

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

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

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

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

Figure 2: Adaptive Semantic Security Games for Result-Hiding DSSE Schemes

Definition 3.1 ($\mathcal{L}$-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:

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

3.4 Forward Privacy

Forward privacy has been introduced by Stefanov et al. [27] and is further explored by Bost et al. [5]. 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 3.2 (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 as:

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

where ind is the document identifier, w is the updated keyword and $\mathcal{L}'$, $\mathcal{L}''$ are stateless.

3.5 Backward Privacy

In addition to forward privacy, Bost et al. specify backward privacy [6]. Backward privacy 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 backward privacy with decreasing strength, depending on the amount of information that is leaked [6]. We consider the two strongest notions.

(1) **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.

(2) **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 to the data: $(\mathsf{add}, \mathsf{ind}_1, w_1)$, $(\mathsf{add}, \mathsf{ind}_1, w_2)$, $(\mathsf{add}, \mathsf{ind}_2, w_1)$, $(\mathsf{del}, \mathsf{ind}_1, w_1)$. Upon a search query for keyword w_1, the first notion reveals ind_2, that it was inserted at time slot 2 and that there were three updates to w_1. The second notion additionally reveals that updates regarding w_1 occurred at time slot 1, 2 and 3. 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.

$$\mathsf{UpHist}(w) = \{(u, \mathsf{op}, \mathsf{ind}) \mid (u, \mathsf{op}, (\mathsf{ind}, w)) \in Q\},$$

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

$$\mathsf{Updates}(w) = \{u \mid (u, \mathsf{op}, (\mathsf{ind}, w)) \in Q\}.$$

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 $\mathsf{Updates}(w)$.

$$\mathsf{ap}(w) = \{\mathsf{ind} \mid (u, \mathsf{ind}) \in \mathsf{TimeDB}(w)\},$$

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

$$\mathsf{Updates}(w) = \{u \mid (u, \mathsf{op}, \mathsf{ind}) \in \mathsf{UpHist}(w)\}.$$

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

Definition 3.3 (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:

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

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 as:

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

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

3.6 Bloom Filters

A Bloom filter is an efficient data structure in which items can be stored, but not retrieved [3]. It can only tell whether it contains an element and does so with a probabilistic nature; it returns either *possibly contains* or *definitively does not contain*. A Bloom filter is an array of bits, which are initially all 0. There are multiple unique hash functions that map an element to a position in the array, following a uniform random distribution. To add an element, it is fed into the hash functions. The resulting positions in the array are set to 1. To test whether an element is in the Bloom filter it is fed into

the hash functions. Then, if any of the resulting positions in the array are set to 0, the element is definitively not in the set. If all positions are 1, the element is either in the set, or the bits are set to 1 due to the insertion of other elements. This false positive rate of the Bloom filter can be controlled by changing the number of inserted elements, the number of hash functions and the length of the array.

4 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.

4.1 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.

$$\mathsf{sp}(q) = \{u \mid (u, q) \in Q\},$$

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

$$\mathsf{UpHist}(q) = \{(u, \mathsf{op}, \mathsf{ind}) \mid (u, \mathsf{op}, (\mathsf{ind}, w)) \in Q \wedge w \dot{\subseteq} q\},$$

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

$$\mathsf{Updates}(q) = \{u \mid (u, \mathsf{op}, (\mathsf{ind}, w)) \in Q \wedge w \dot{\subseteq} q\}.$$

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 backward privacy introduced earlier to the wildcard setting by using the leakage functions we defined.

Definition 4.1 (Insertion Pattern Revealing Backward Privacy For Wildcard Supporting SSE Schemes). 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:

$$\mathcal{L}^{\mathsf{Srch}}(q) = \mathcal{L}'(\mathsf{TimeDB}(q), a_q),$$
$$\mathcal{L}^{\mathsf{Add}}(\mathsf{ind}, w) = \perp,$$
$$\mathcal{L}^{\mathsf{Del}}(\mathsf{ind}, w) = \perp,$$

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

Definition 4.2 (Update Pattern Revealing Backward Privacy For Wildcard Supporting SSE Schemes). A wildcard supporting, $\mathcal{L}$-adaptively-secure SSE scheme is **update 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:

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

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

5 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 [6]. 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 backward privacy. 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.

5.1 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 as

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

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

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.

5.2 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.

5.2.1 Storage. The client stores one extra key K_{Lib} and maintains 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 that are currently in the database.

5.2.2 Operations. During the setup phase, the client generates an extra key K_{Lib}. For add and delete operations, the client performs an additional 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.

5.2.3 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.

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

5.3 Security

Theorem 5.1. *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*

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

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

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

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

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

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

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

$$\mathcal{L}^{\mathsf{Add}}_{\mathsf{Lib}_{fp}}(\mathsf{ind}, w) = \perp,$$
$$\mathcal{L}^{\mathsf{Del}}_{\mathsf{Lib}_{fp}}(\mathsf{ind}, w) = \perp,$$

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 outputs of $\mathrm{SSE}^{\mathsf{Lib}}_{\mathrm{Real}_{\mathcal{A}}}(\lambda, n)$ and $\mathrm{SSE}_{\mathrm{Ideal}_{\mathcal{A}, \mathcal{S}_{\mathsf{Lib}}, \mathcal{L}_{\mathsf{Lib}}}}(\lambda, n)$ 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 $\widetilde{\mathcal{L}}^{\mathsf{Srch}}_{\Sigma}(q) = \mathcal{L}(\widetilde{\mathsf{sp}}_{\Sigma}(q), \widetilde{\mathsf{UpHist}}_{\Sigma}(q))$ as follows:

  $$\widetilde{\mathsf{sp}}_{\Sigma}(q) = \mathsf{sp}_{\mathsf{Lib}}(q),$$
  $$\widetilde{\mathsf{UpHist}}_{\Sigma}(q) = \{(u, \mathsf{add}, E_{K_{\mathcal{S}_{\mathsf{Lib}}}}(\perp_c, \perp_{\mathsf{op}}, \perp_{\mathsf{ind}}, \perp_w)) \mid u \in \mathsf{Updates}_{\mathsf{Lib}}(q)\}.$$

  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 Libertas results in a search for query q in Σ, the search patterns for 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 $\widetilde{\mathcal{L}}^{\mathsf{Add}}_{\Sigma}(\mathsf{ind}, w) = \mathcal{L}(\widetilde{\mathsf{ind}}, \widetilde{w})$ as follows:

  $$\widetilde{\mathsf{ind}} = E_{K_{\mathcal{S}_{\mathsf{Lib}}}}(\perp_c, \perp_{\mathsf{op}}, \perp_{\mathsf{ind}}, \perp_w),$$
  $$\widetilde{w} = w.$$

  Then, rather than running $\Sigma.\mathsf{AddToken}(K_{\Sigma}, E_{K_{\mathsf{Lib}}}(c, \mathsf{add}, \mathsf{ind}, w))$, 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 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 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

  $$\mathcal{L}^{\mathsf{Add}}_{\mathsf{Lib}_{fp}}(\mathsf{ind}, w) = \perp .$$

  We construct $\widetilde{\mathcal{L}}^{\mathsf{Add}}_{\Sigma_{fp}}(\mathsf{ind}, w) = \mathcal{L}(\widetilde{\mathsf{ind}})$ as follows:

  $$\widetilde{\mathsf{ind}} = E_{K_{\mathcal{S}_{\mathsf{Lib}}}}(\perp_c, \perp_{\mathsf{op}}, \perp_{\mathsf{ind}}, \perp_w),$$

- (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 $\widetilde{R}^*$ as follows:

  $$\widetilde{R}^* = \{E_{K_{\mathcal{S}_{\mathsf{Lib}}}}(\perp_c, \perp_{\mathsf{op}}, \perp_{\mathsf{ind}}, \perp_w) \mid u \in \mathsf{Updates}_{\mathsf{Lib}}(q)\},$$

  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 $\widetilde{R}$ as follows:

  $$\widetilde{R} = \{\mathsf{ind} \mid (u, \mathsf{ind}) \in \mathsf{TimeDB}_{\mathsf{Lib}}(q)\}.$$

□

6 EVALUATION

In order to empirically evaluate the cost of backward privacy in our Libertas construction, we implemented Libertas and a wildcard supporting scheme. We picked the scheme proposed by Zhao and Nishide [33] as it is an exemplar wildcard scheme. It is forward private and allows for updates on a document-keyword pair level rather than considering complete documents, making integration with Libertas easy. Additionally, it supports two wildcard types, allowing for greater query flexibility.

6.1 Zhao and Nishide Recap

The scheme by Zhao and Nishide [33] makes use of Bloom filters [3] to store keyword and query characteristics. For every document-keyword pair, a Bloom filter is stored in the index. Queries are translated into a Bloom filter that is subsequently checked against stored Bloom filters to find matching documents. Rather than checking all bits, the search algorithm only requires that all bits set in the query Bloom filter are also set in the Bloom filter generated for the keyword. An overview of the scheme's algorithms, including the generation of the Bloom filters, can be found in Appendix A. For the rest of the paper, we will refer to the scheme as Z&N.

6.2 Setup

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

6.2.2 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.

6.2.3 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.

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

6.3 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 $\text{Libertas}_{\text{Z\&N}}$. We disregarded the SrchToken operation as it is identical for both schemes.

6.3.1 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.

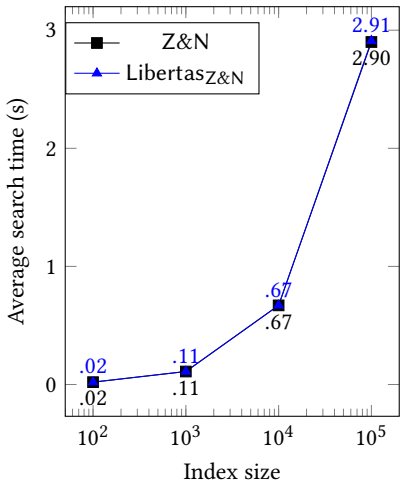


Figure 3: Average search time for exact keyword search per index size (x-axis in logarithmic scale).

6.3.2 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.

6.3.3 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.

6.3.4 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.

6.4 Results

6.4.1 Basic Search. We can see from Figure 3 that $\text{Libertas}_{\text{Z\&N}}$ experiences virtually no overhead compared to Z&N when considering exact keyword searches, regardless of the index size.

6.4.2 Wildcard Query Search. Figure 4 shows us that the overhead of $\text{Libertas}_{\text{Z\&N}}$ barely increases when considering queries containing wildcards such that they match multiple keywords. Note that, for the given data set, every additional wildcard increases the number of matching keywords ten-fold. $\text{Libertas}_{\text{Z\&N}}$ appears to be faster

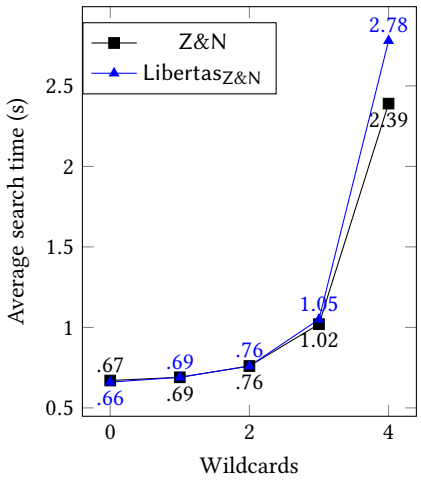


Figure 4: Average search time for wildcard query search per number of wildcards (index size 10^4).

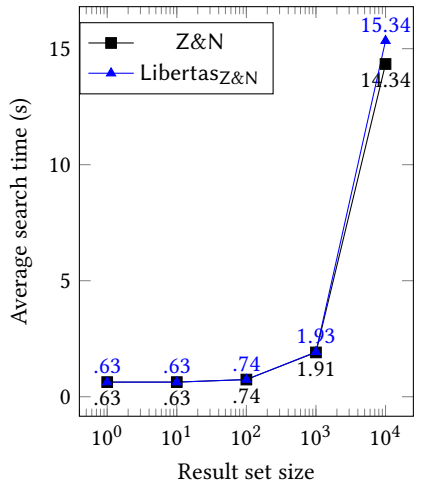


Figure 5: Average search time per result set size (x-axis in logarithmic scale, index size 10^4).

than Z&N when the query contains no wildcards. This is merely a result of measurement error.

6.4.3 Varying Result Set Size. Figure 5 indicates that Libertas$_{Z\&N}$ and Z&N have a comparable performance regardless of result set size.

6.4.4 Varying Number of Deletions. Figure 6 clearly 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's nature, however, its index increases, slowing down searches linearly with the number of deletions instead. Libertas$_{Z\&N}$ appears to be faster than Z&N when there are no deletions. This is merely a result of measurement error.

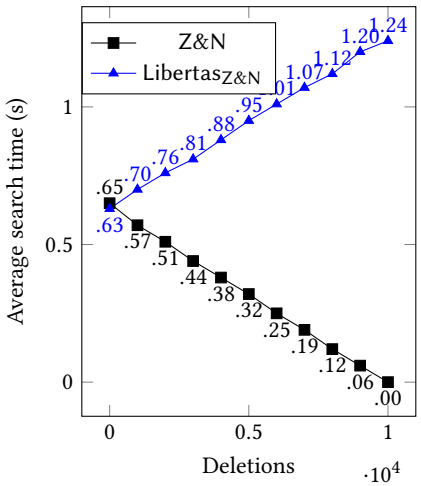


Figure 6: Average search time per number of deletions (index size 10^4).

7 DISCUSSION

7.1 Query similarity

The wildcard leakage functions we introduced in Section 4.1 allow for query similarity leakage. We consider Definition 4.1. 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.

7.2 Real World Application

Libertas$_{Z\&N}$ 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.

7.3 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. [6] 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. [21].

7.4 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 [23].

8 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.

REFERENCES

[1] Martın Abadi and Phillip Rogaway. 2000. Reconciling two views of cryptography. In *Proceedings of the IFIP International Conference on Theoretical Computer Science*. Springer, 3–22.

[2] John Bethencourt, H Chan, Adrian Perrig, Elaine Shi, and Dawn Song. 2006. Anonymous multi-attribute encryption with range query and conditional decryption. In *IEEE Symposium on Security & Privacy*.

[3] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[4] Dan Boneh and Brent Waters. 2007. Conjunctive, subset, and range queries on encrypted data. In *Theory of cryptography conference*. Springer, 535–554.

[5] Raphael Bost. 2016. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1143–1154.

[6] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. 2017. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1465–1482.

[7] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Annual cryptology conference*. Springer, 505–524.

[8] Christoph Bösch, Richard Brinkman, Pieter Hartel, and Willem Jonker. 2011. Conjunctive Wildcard Search over Encrypted Data. https://link.springer.com/chapter/10.1007/978-3-642-23556-6_8

[9] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 668–679.

[10] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. 2013. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Annual cryptology conference*. Springer, 353–373.

[11] Melissa Chase and Emily Shen. 2015. Substring-Searchable Symmetric Encryption. *Proc. Priv. Enhancing Technol.* 2015, 2 (2015), 263–281.

[12] Sanjit Chatterjee, Manish Kesarwani, Jayam Modi, Sayantan Mukherjee, Shravan Kumar Parshuram Puria, and Akash Shah. 2020. Secure and efficient wildcard search over encrypted data. *International Journal of Information Security* (2020), 1–46.

[13] Shen-Ming Chung, Ming-Der Shieh, and Tzi-Cker Chiueh. 2019. FETCH: A cloud-native searchable encryption scheme enabling efficient pattern search on encrypted data within cloud services. *International Journal of Communication Systems* (2019), e4141.

[14] Shen-Ming Chung, Ming-Der Shieh, Tzi-Cker Chiueh, Chia-Chia Liu, and Chia-Heng Tu. 2020. uFETCH: A Unified Searchable Encryption Scheme and Its Saas-Native to Make DBMS Privacy-Preserving. *IEEE Access* 8 (2020), 93894–93906.

[15] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. 2011. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security* 19, 5 (2011), 895–934.

[16] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. 2015. Rich queries on encrypted data: Beyond exact matches. In *European symposium on research in computer security*. Springer, 123–145.

[17] Eu-Jin Goh et al. 2003. Secure indexes. *IACR Cryptol. ePrint Arch.* 2003 (2003), 216.

[18] Changhui Hu and Lidong Han. 2016. Efficient wildcard search over encrypted data. *International Journal of Information Security* 15, 5 (2016), 539–547.

[19] Changhui Hu, Lidong Han, and Siu Ming Yiu. 2016. Efficient and secure multi-functional searchable symmetric encryption schemes. *Security and Communication Networks* 9, 1 (2016), 34–42.

[20] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: ramification, attack and mitigation.. In *Ndss*, Vol. 20. Citeseer, 12.

[21] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 965–976.

[22] Myungsun Kim, Hyung Tae Lee, San Ling, Benjamin Hong Meng Tan, and Huaxiong Wang. 2017. Private compound wildcard queries using fully homomorphic encryption. *IEEE Transactions on Dependable and Secure Computing* (2017).

[23] Muhammad Naveed. 2015. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. *IACR Cryptol. ePrint Arch.* 2015 (2015), 668.

[24] Tushar Kanti Saha and Takeshi Koshiba. 2016. An enhancement of privacy-preserving wildcards pattern matching. In *International Symposium on Foundations and Practice of Security*. Springer, 145–160.

[25] Saeed Sedghi, Peter Van Liesdonk, Svetla Nikova, Pieter Hartel, and Willem Jonker. 2010. Searching keywords with wildcards on encrypted data. In *International Conference on Security and Cryptography for Networks*. Springer, 138–153.

[26] Dawn Xiaoding Song, David Wagner, and Adrian Perrig. 2000. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*. IEEE, 44–55.

[27] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical Dynamic Searchable Encryption with Small Leakage.. In *NDSS*, Vol. 71. 72–75.

[28] Takanori Suga, Takashi Nishide, and Kouichi Sakurai. 2012. Secure keyword search using Bloom filter with specified character positions. In *International Conference on Provable Security*. Springer, 235–252.

[29] Zhaoli Wang, Jinli Han, Meijuan Wang, Yaqing Shi, and Hui Dong. 2018. Public Key Encryption with Wildcards Keyword Search. In *2018 Eighth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC)*. IEEE, 538–541.

[30] Yang Yang, Ximeng Liu, Robert H Deng, and Jian Weng. 2017. Flexible wildcard searchable encryption system. *IEEE Transactions on Services Computing* 13, 3 (2017), 464–477.

[31] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. 2014. Privacy-preserving wildcards pattern matching using

symmetric somewhat homomorphic encryption. In *Australasian Conference on Information Security and Privacy*. Springer, 338–353.

[32] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2016. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 707–720.

[33] Fangming Zhao and Takashi Nishide. 2016. Searchable symmetric encryption supporting queries with multiple-character wildcards. In *International Conference on Network and System Security*. Springer, 266–282.

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 [33] and uses Bloom filters [3] 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. ind $\|$ 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, $S_K^{(o)}$('diana') = {'1:d', '2:i', '3:a', '4:n', '5:a', '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'.

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

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.

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

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. $S_T^{(o)}$('di*a_a*\0') = {'1:d', '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 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 $S_K^{(p2)}(w)$.

Algorithm 2 Z&N

Setup(λ)

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

BuildIndex(λ)

$\gamma \leftarrow$ empty list

SrchToken(K, q)

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

AddToken(K, ind, w)

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

DelToken(K, ind, w)

$b_{\mathsf{id}} \leftarrow g(K_G, \mathsf{ind} \,\|\, w)$
$\tau^{\mathsf{del}} = b_{\mathsf{id}}$
Return τ^{del}

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

τ^{srch} consists of Bloom filter positions and hashes of these positions. We arrange these as $((p_1, g(K_G, p_1)), \ldots, (p_i, g(K_G, p_i)))$
For all $(\mathsf{ind}, \mathsf{BF}, b_{\mathsf{id}})$ in γ:
    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 $t \in [1, i]$.
Return R

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

Add τ^{add} to γ

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

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