Practical Non-Interactive Searchable Encryption with Forward and Backward Privacy

Shi-Feng Sun*, Ron Steinfeld*, Shangqi Lai*,
Xingliang Yuan*, Amin Sakzad*, Joseph K. Liu*, Surya Nepal† and Dawu Gu‡
* Monash University, Australia
† Data61, CSIRO, Australia
‡ Shanghai Jiao Tong University, China

***Abstract*—In Dynamic Symmetric Searchable Encryption (DSSE), forward privacy ensures that previous search queries cannot be associated with future updates, while backward privacy guarantees that subsequent search queries cannot be associated with deleted documents in the past. In this work, we propose a generic forward and backward-private DSSE scheme, which is, to the best of our knowledge, the *first practical and non-interactive Type-II backward-private* DSSE scheme not relying on trusted execution environments. To this end, we first introduce a new cryptographic primitive, named *Symmetric Revocable Encryption* (SRE), and propose a modular construction from some succinct cryptographic primitives. Then we present our DSSE scheme based on the proposed SRE, and instantiate it with lightweight symmetric primitives. At last, we implement our scheme and compare it with the most efficient** Type-II **backward-private scheme to date (Demertzis *et al.*, NDSS 2020). In a typical network environment, our result shows that the search in our scheme outperforms it by $2-11\times$ under the same security notion.**

I. Introduction

Symmetric Searchable Encryption (SSE) enables a client to encrypt a collection of data and outsource it to an untrusted server. In general, the data is encrypted in such a way that it can be efficiently searched while not sacrificing data and query privacy. In contrast to the general-purpose solutions to search over encrypted data (*e.g.*, ORAM [30] or fully-homomorphic encryption [28]), SSE achieves a better efficiency, at the cost of allowing for some information leakage captured by a well-specified leakage function [21], [13].

The early work on SSE [50], [21], [13], [26] focused on private search over static data. In order to support updates on encrypted data, some progress has been made on DSSE [43], [42], [12], [51], [35], [48]. In this setting, a client should be able to arbitrarily *add* documents into or *delete* them from the database, and the private search should be still supported, even after the updates. However, the update operations may reveal additional information, which for example has been leveraged by the file injection attack [57] to breach query privacy.

Most recent work on SSE [7], [9], [46], [15], [52], [4] concentrated on how to improve the security of DSSE. Specifically, there has been recently a lot of interest in the notions of *forward* and *backward privacy*. They were introduced by Stefanov *et al.* [51] and later formalized by Bost *et al.* [7], [9]. Informally, forward privacy ensures that future updates cannot be associated to previous search queries, which is useful for mitigating the powerful file injection attacks and has drawn extensive attention [7], [9], [46], [25], [3], [38] in recent years. In contrast, backward privacy guarantees that subsequent searches cannot be associated to the deleted documents in the past. Ideally, the leakage of backward-private schemes should only depend on the documents *currently* matching the query (*i.e.*, the matched documents excluding the deleted ones) in the database, but it is very difficult to achieve without using the complicated techniques, *e.g.*, ORAM, as it requires to hide both *the number and the pattern of updates*. As a consequence, Bost *et al.* [9] defined, in terms of how much information leaked from the additions and deletions, three flavors of backward privacy for single-keyword search:

Type-I backward privacy: allows schemes to leak the (identifers of) documents *currently* matching the queried keyword w, the timestamps of inserting them into the database, as well as the total number of updates on w.

Type-II backward privacy: additionally leaks the timestamps and operation types of all updates on the queried keyword w, apart from the information revealed in Type-I.

Type-III backward privacy: compared to Type-II, further leaks which deletion operation cancelled precisely which addition.

The above three types of privacy are ordered from the most to the least secure; the stronger privacy the schemes achieve, the less information the updates leak. As with the leakage of additions that has been successfully leveraged by file injection attacks to break query privacy [57], the information revealed from deletions might also be exploited to launch potential attacks. Specifically, Type-III backward privacy leaks which documents are deleted at what time. It has been demonstrated that time is crucial information and can be exploited to break the security of a wide range of systems, *e.g.*, timing analysis on network traffic [27] and side-channel attacks against hardware enclaves [18]. In our application, the timestamps of updates seem unrelated to the content of data, but knowing when a keyword and/or a document is deleted could create opportunities for attackers to correlate other information of later queries or launch statistical inference. Therefore, it is desired to design SSE schemes with as strong as possible privacy, *i.e.*, minimising information leakage of updates.

Recently, a few backward-private SSE schemes [9], [15],

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24162
www.ndss-symposium.org

[52], [4] have been proposed per flavor. However, most existing schemes achieve different tradeoffs between computation/communication cost and security guarantees. In brief, the Type-I schemes [9], [15] are based on ORAMs, and Type-II schemes [9], [15], [22] need high communication cost and multiple rounds of interactions to perform a search/update query. This cost incurred by interactions is usually non-negligible and will lead to a noticeable delay ($20 - 30\times$ slower than the non-interactive scheme as in our evaluation) in the real-world network environment. In contrast, the schemes [9], [52] need only one roundtrip for each search and update, but at the expense of providing only the *weakest* backward privacy. Very recently, the hardware-based Type-I scheme, Type-II scheme and Type-III scheme were proposed in [4], [55], which are all *non-interactive* but rely heavily on the power of Intel SGX [20]. Unfortunately, various security vulnerabilities in SGX have been revealed, such as [10], [54], [18], and it may still suffer from potential (side-channel) attacks in future.

Therefore, it is still challenging to design *practical*, *non-interactive* and *strong backward-private* DSSE schemes *without hardware* assumptions. In this paper, we make affirmative progress towards this problem and put forward the *first* generic DSSE scheme with all above desired features by leveraging a newly introduced cryptographic primitive. The main contributions are summarized below.

Our Contributions. We explore the new way of constructing (forward and) backward-private SSE schemes in this work. Our basic idea, similar to [9], [52], is to encrypt the document identifiers in such a way that the deleted ones cannot be decrypted, even if they can be retrieved by the server. To achieve a higher level of backward privacy, the crucial point is to make the update operations leak as little information as possible. Our starting point is to make the server oblivious of the deletions. To do so, we introduce a new cryptographic primitive, named *Symmetric Revokable Encryption* (SRE), which allows us to accomplish the deletions locally with a low storage request and captures the essential properties needed for designing backward-private SSE schemes following the above idea. In details, our main contributions include:

- We formalize the syntax and security of SRE, which can be seen as a symmetric predicate encryption and may be independent of interest. Then we propose a generic construction of SRE from a multiple puncturable PRF, a traditional symmetric encryption scheme and a Bloom filter. Due to the usage of Bloom filter, our construction features a compressed revocation procedure, which is crucial for our SSE application. Furthermore, we show it satisfies the proposed security in the standard model.
- Based on the proposed SRE scheme and a basic forward-private SSE scheme, we present a new *non-interactive* DSSE scheme, and argue that it is Type-II backward-private under the security notion of [9]. Also, it inherits the forward privacy of the underlying SSE scheme. To the best of our knowledge, the proposed scheme is the *first forward and Type-II backward-private* SSE scheme that supports both *non-interactive search and update* and depends *not* on the random oracles and hardware assumptions.
- We introduce an efficient instantiation, named Aura, of our DSSE scheme with the GGM tree-based PRF [29], and obtain the first practical, scalable and *non-interactive* DSSE scheme with both forward and Type-II backward privacy as well as support for *large* deletions. A comprehensive comparison of Aura with previous works is summarised in Table I, where we only consider the schemes without hardware assumptions. We implement Aura and perform a comprehensive evaluation. The results show that Aura outperforms both the state-of-the-art (non)-interactive forward and backward-private DSSE schemes (Janus++ [52] and SD_d [22]) in terms of search time, insertion time, deletion time, and communication cost.

Technical Overview. Following the basic idea of designing backward-private SSE schemes mentioned before, the main task is to develop an encryption scheme that can revoke the decryption capability of the master secret key; given a revoked secret key on a list of tags (associated with a set of document identifier/keyword pairs), the server can recover an encrypted identifier if the associated tag does not belong to the tag list, otherwise fails. This is achieved by relying heavily on incremental puncturable encryption in both Janus [9] and Janus++ [52], where a fresh key component is produced and outsourced to the server whenever a deletion happens. Thus too much information about deletions is revealed to the server.

To achieve stronger backward privacy, our essential idea is to make the deletions (*i.e.*, revocation) *oblivious* to the server. We observe that the schemes of [9], [52] could in fact be run in this way, exactly by locally recording all tags to be deleted and then revoking them *in one shot* rather than in an *incremental* manner. However, this usually requires to allocate some storage for recording these tags before generating the revoked secret key for them. In general, the cost of storage requested grows linearly with the size of all tags to be deleted. This is undesirable in SSE scenarios.

What we need in essence is an encryption mechanism that enables us to revoke the decryption capability of the initial secret key over a set of tags *in one shot* and with only a *low request for memory*. Informally, it is like a predicate encryption [44], [45] for predicate $\mathsf{P}(R, t) = 1$ *iff* $t \notin R$, where R is a set of tags associated with a revoked secret key, t is a tag attached to a ciphertext, and the decryption succeeds only if $\mathsf{P}(R, t) = 1$. In this sense, it can be seen as the dual of identity-based revocation system [47]. As far as we know, unfortunately, there have been no such schemes proposed so far.

In order to design such a practical encryption scheme that can revoke a list of tags *in one shot* with only *a low memory request*, our main idea is to first *compress* all tags one-by-one to a short-size data structure, and then conduct the revocation based on this data structure. Specifically, in our construction we employ the Bloom filter [6], which is *a well-known data structure for compact set presentation*, and perform the revocation by leveraging the multi-puncturable PRF [36]. For revocation, we generate a punctured secret key on all indices of the entries (of the Bloom filter) with value '1', which are corresponding to the tags to be revoked. With this key, we can compute the value of the pseudorandom function on at least one index if the associated tag is not revoked, as there exists at least one entry with value '0' for an unrevoked tag (due to the property of the Bloom filter). Otherwise, no PRF value can be computed on any of the indices derived

TABLE I: Comparison with previous works. N, D, and W denote the total number of keyword/document pairs, total number of documents, and total number of distinct keywords in the database, respectively. For a keyword w, a_w is the total number of inserted entries matching w, d_w denotes the number of deleted entries matching w and $d = \max_w d_w$. n_w is the size of search result matching w and $n_w = a_w - d_w$. $\tilde{O}$ notation hides polylogarithmic factors.

Schemes	Communication			Computation		Client Storage	Backward Privacy
	#Rounds‡	Search	Update	Search	Update		
Moneta [9]	3	$\tilde{O}(a_w \log N + \log^3 N)$	$\tilde{O}(\log^3 N)$	$\tilde{O}(a_w \log N + \log^3 N)$	$\tilde{O}(\log^2 N)$	$O(1)$	Type-I
Orion [15]	$O(\log N)$	$O(n_w \log^2 N)$	$O(\log^2 N)$	$O(n_w \log^2 N)$	$O(\log^2 N)$	$O(1)$	Type-I
Fides [9]	2	$O(a_w + d_w)$	$O(1)$	$O(a_w + d_w)$	$O(1)$	$O(W \log D)$	Type-II
Mitra [15]	2	$O(a_w + d_w)$	$O(1)$	$O(a_w + d_w)$	$O(1)$	$O(W \log D)$	Type-II
SD_a [22]	2	$O(a_w + \log N)$	$O(\log N)$	$O(a_w + \log N)$	$O(\log N)$	$O(1)$	Type-II
SD_d [22]	2	$O(a_w + \log N)$	$O(\log^3 N)$	$O(a_w + \log N)$	$O(\log^3 N)$	$O(1)$	Type-II
Aura [Sec. IV]	1	$O(n_w)$	$O(1)$	$O(n_w)$	$O(1)$	$O(Wd)$†	Type-II
$\mathsf{Diana}_{\mathsf{del}}$ [9]	2	$O(n_w + d_w \log a_w)$	$O(1)$	$O(a_w)$	$O(\log a_w)$	$O(W \log D)$	Type-III
Janus [9]	1	$O(n_w)$	$O(1)$	$O(n_w d_w)$	$O(1)$	$O(W \log D)$	Type-III
Horus [15]	$O(\log d_w)$	$O(n_w \log d_w \log N)$	$O(\log^2 N)$	$O(n_w \log d_w \log N)$	$O(\log^2 N)$	$O(W \log D)$	Type-III
Janus++ [52]	1	$O(n_w)$	$O(1)$	$O(n_w d)$	$O(d)$	$O(W \log D)$	Type-III

†: Our storage cost is asymptotically larger than others, but it can be seen from Table II that the overhead is acceptable in practice.
‡: The #Rounds here indicates the rounds of communication required for servers to receive the document ind. This setting is consistent with the existing SSE schemes [21], [43], [12], [9], [15].

from this tag. To encrypt a message along with a tag, we only need to encrypt it with the PRF values evaluated on the entry indices (of the Bloom filter) associated with the tag, which is similar to the encryption method of Bloom filter encryption [24], and the decryption can be realized by using the punctured secret key. Thus we get a kind of revocable encryption with *compressed* revocation and call it *Compressed* SRE.

A. Related Work

Song *et al.* [50] introduced the notion of SSE. After that, extensive effort has been made to improve its security [16], [21], [51], [7], [9], [41], functionality [17], [43], [13], [26], [35], [39] or performance [13], [12], [14], [48], [23]. Almost all SSE schemes allow for the leakage termed search pattern and access pattern. To understand the realistic impacts of leakage, the community has recently started to study how to exploit it to break the security of SSE [11], [57], [34], [33], [5]. Most recently, a line of work for defence [8], [41], [40] has been presented. Concurrently, another line of important work is to design schemes with forward and backward privacy, which guarantees a higher level security for DSSE.

Forward and backward privacy was initialized by Stefanov *et al.* [51] and later formalized by Bost *et al.* [7], [9]. Since then, forward privacy has been studied extensively and many efficient schemes [9], [15], [46], [38], [25], [3], [4] have been proposed following the seminal work of Bost [7]. Backward privacy, however, has been investigated far less. Bost *et al.* [9] first proposed several constructions from constrained cryptographic primitives, including Moneta that can achieve the strongest (*i.e.*, Type-I) backward privacy, Type-II scheme Fides, and Type-III schemes $\mathsf{Diana}_{\mathsf{del}}$ and Janus. Subsequently, Chamani *et al.* [15] proposed three improved constructions, including Type-I scheme Orion, Type-II scheme Mitra and Type-III scheme Horus. At the same time, Sun *et al.* [52] proposed a practical Type-III scheme Janus++ by making use of their symmetric puncturable encryption. Very recently, Demertzis *et al.* [22] proposed three new schemes focusing on small client storage, namely Type-III scheme QOS, Type-II schemes SD_a and SD_d. All these schemes achieve different tradeoffs between security and efficiency. In particular, the existing Type-I and II schemes rely on either ORAMs or multi-round of interactions. The exceptions are Janus [9] and Janus++ [52] that are practical and completely non-interactive but achieving only Type-III backward privacy. In addition, Amjad *et al.* [4] proposed several schemes with all types of backward privacy by leveraging the power of Intel SGX [20]. They are all non-interactive but depending heavily on the security and reliability of trusted execution environments.

In this work, we will explore new approaches of designing practical, non-interactive, and forward- and backward-private DSSE schemes without hardware assumptions.

II. Background

In this part, we recollect the syntax and security of the cryptographic primitives used throughout this work.

A. Symmetric Encryption

A Symmetric Encryption (SE) scheme with message space $\mathcal{M}$, key space $\mathcal{K}$ and ciphertext space $\mathcal{C}$ consists of three polynomial-time algorithms $\mathsf{SE} = (\mathsf{SE.Gen}, \mathsf{SE.Enc}, \mathsf{SE.Dec})$: On input a security parameter λ, $\mathsf{SE.Gen}(1^\lambda)$ outputs a secret key $k \in \mathcal{K}$; $\mathsf{SE.Enc}(k, m)$ takes as input a secret key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$, and outputs a ciphertext $ct \in \mathcal{C}$; $\mathsf{SE.Dec}(k, ct)$ takes a secret key $k \in \mathcal{K}$ and a ciphertext ct, and outputs m or $\perp$ that indicates failure.

An SE scheme is *perfectly correct* if for all $m \in \mathcal{M}$, $k \leftarrow \mathsf{SE.Gen}(1^\lambda)$ and $ct \leftarrow \mathsf{SE.Enc}(k, m)$, it holds that $\Pr[\mathsf{SE.Dec}(k, ct) = m] = 1$.

SECURITY. The IND-CPA security of SE is defined by the following experiment $\mathbf{Exp}^{\text{IND-CPA}}_{\mathsf{SE},\mathcal{A}}(\lambda)$ between a challenger and an adversary $\mathcal{A}$.

Setup: Challenger runs $k \leftarrow \mathsf{SE.Gen}(1^\lambda)$ and chooses a random bit $\gamma \in \{0, 1\}$.

Phase 1: $\mathcal{A}$ adaptively issues a polynomial number of encryption queries. For each query on $m \in \mathcal{M}$, the challenger returns $ct \leftarrow \mathsf{SE.Enc}(k, m)$.

Challenge: $\mathcal{A}$ issues messages $m_0, m_1 \in \mathcal{M}$ with equal length, and receives ciphertext $ct^* \leftarrow \mathsf{SE.Enc}(k, m_\gamma)$.

Phase 2: This is identical to Phase 1.

Guess: $\mathcal{A}$ outputs γ'. The experiment outputs 1 if $\gamma' = \gamma$.

Definition 1 (IND-CPA Security)**.** *An SE scheme* $\mathsf{SE} = (\mathsf{SE.Gen}, \mathsf{SE.Enc}, \mathsf{SE.Dec})$ *is IND-CPA secure if for all* $\lambda \in \mathbb{N}$ *and probabilistic polynomial time (PPT) adversaries* $\mathcal{A}$*, the advantage of* $\mathcal{A}$ *winning in the experiment*

$$\mathsf{Adv}^{\textit{IND-CPA}}_{\mathsf{SE},\mathcal{A}}(\lambda) = |\Pr[\mathbf{Exp}^{\text{IND-CPA}}_{\mathsf{SE},\mathcal{A}}(\lambda) = 1] - 1/2| \leq \nu(\lambda)$$

where the probability is taken over the randomness of the experment and $\nu(\lambda)$ *is negligible in* λ*.*

B. Bloom Filter

A Bloom Filter (BF) [6] is a probabilistic data structure. It can be used to rapidly and space-efficiently perform set membership test, at the cost of allowing for false positives. For many applications, the *space saving* outweighs this drawback when the false-positive probability is small enough. In this work, we focus on the standard BF given in [6] that is sufficient for our applications. Next we recall its formal definition following the syntax of [24]. Particularly, the Bloom filter in [6] consists of three polynomial-time algorithms $\mathsf{BF} = (\mathsf{BF.Gen}, \mathsf{BF.Upd}, \mathsf{BF.Check})$:

$\mathsf{BF.Gen}(b, h)$: It takes as input two integers $b, h \in \mathbb{N}$, and samples a collection of universal hash functions $H = \{H_j\}_{j\in[h]}$, where $H_j : \mathcal{X} \to [b]$ is from a universe $\mathcal{X}$ to a finite set $[b]$. Finally, it outputs H and an initial b-bit array $B = 0^b$ with each bit $B[i]$ for $i \in [b]$ set to 0.

$\mathsf{BF.Upd}(H, B, x)$: It takes $H = \{H_j\}_{j\in[h]}$, $B \in \{0,1\}^b$ and an element $x \in \mathcal{X}$, updates the current array B by setting $B[H_j(x)] \leftarrow 1$ for all $j \in [h]$, and finally outputs the updated B. For simplicity, we use $B_R \leftarrow \mathsf{BF.Upd}(H, B, R)$ to denote the final array after inserting all elements in R one-by-one.

$\mathsf{BF.Check}(H, B, x)$: It takes $H = \{H_j\}_{j\in[h]}$, $B \in \{0,1\}^b$ and an element $x \in \mathcal{X}$, and checks if $B[H_j(x)] = 1$ for all $j \in [h]$. If ture, it outputs 1, otherwise returns 0.

A Bloom filter BF is *perfectly complete* if for all integers $b, h \in \mathbb{N}$, any set R of elements in $\mathcal{X}$, and $(H, B) \leftarrow \mathsf{BF.Gen}(b, h)$ as well as $B_R \leftarrow \mathsf{BF.Upd}(H, B, R)$, it holds

$$\Pr\left[\mathsf{BF.Check}(H, B_R, x) = 1\right] = 1$$

for all $x \in R$. This means a BF with perfect completeness can always recognize the added elements.

Next, we briefly introduce the definition of *false-positive probability*. Informally, it is the probability that an element not yet added to BF is mistaken for being contained in it. Given an upper-bound on the size of R, the probability can be made sufficiently low by adjusting the parameters b, h adequately. Formally, for a set R of n elements in $\mathcal{X}$, we let $(H, B) \leftarrow \mathsf{BF.Gen}(b, h)$ and $B_R \leftarrow \mathsf{BF.Upd}(H, B, R)$. Then for any $x \in \mathcal{X}$, we have that

$$\Pr[\mathsf{BF.Check}(H, B_R, x) = 1 \wedge x \notin R] \approx (1 - e^{-nh/b})^h,$$

where the probability is taken over the randomness of $\mathsf{BF.Gen}(b, h)$.

C. Puncturable Pseudorandom Function

First, we introduce the syntax and security of (a variant of) t-puncturable pseudorandom function (t-Punc-PRF) [36]. Informally, a t-Punc-PRF allows a PRF key to be punctured at any set of inputs S s.t. $|S| \leq t$, where $t(\cdot)$ is a fixed polynomial. Formally, a function $F_t : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is a t-Punc-PRF with key space $\mathcal{K}$ if there is an additional key space $\mathcal{K}_p$ and three polynomial-time algorithms ($\mathsf{F}_t.\mathsf{Setup}$, $\mathsf{F}_t.\mathsf{Punc}$, $\mathsf{F}_t.\mathsf{Eval}$) with the following syntax:

$\mathsf{F}_t.\mathsf{Setup}(1^\lambda)$: It takes a security parameter λ and outputs a description of a PRF key $k \in \mathcal{K}$.

$\mathsf{F}_t.\mathsf{Punc}(k, S)$: It takes a key $k \in \mathcal{K}$ and a set of elements $S \subset \mathcal{X}$ s.t. $|S| \leq t(\lambda)$, and outputs a punctured key $k_S \in \mathcal{K}_p$.

$\mathsf{F}_t.\mathsf{Eval}(k_S, x)$: It takes as input a punctured key $k_S \in \mathcal{K}_p$ and $x \in \mathcal{X}$, and outputs $y \in \mathcal{Y}$ or a symbol $\perp$.

A t-Punc-PRF is *correct* if for all $S \subset \mathcal{X}$ s.t. $|S| \leq t(\lambda)$, $x \in \mathcal{X} \setminus S$, $k \leftarrow \mathsf{F}_t.\mathsf{Setup}(1^\lambda)$, and $k_S \leftarrow \mathsf{F}_t.\mathsf{Punc}(k, S)$, it holds that $\Pr[\mathsf{F}_t.\mathsf{Eval}(k_S, x) \neq F_t(k, x)] \leq \nu(\lambda)$, where the probability is taken over $k \in \mathcal{K}$ and $\nu(\lambda)$ is negligible.

SECURITY. In contrast to [36], a weaker security of t-Punc-PRF F_t is sufficient for our applications, where an adversary $\mathcal{A}$ is only permitted to ask for a *single* punctured secret key query. Precisely, the security is defined by the experiment $\mathbf{Exp}^{t\text{-Punc-PRF}}_{F_t,\mathcal{A}}(\lambda)$ below.

Setup: Challenger chooses a random bit $\gamma \in \{0,1\}$ and runs $k \leftarrow \mathsf{F}_t.\mathsf{Setup}(1^\lambda)$.

Challenge: On input $S^* = \{x^*_1, x^*_2, \ldots, x^*_\ell\} \subset \mathcal{X}$ from $\mathcal{A}$, the challenger computes $k_{S^*} \leftarrow \mathsf{F}_t.\mathsf{Punc}(k, S^*), y^*_i = F(k, x^*_i)$ and selects $u_i \xleftarrow{\$} \mathcal{Y}$ for all $i \in [\ell]$. If $\gamma = 0$, it returns $(k_{S^*}, \{y^*_i\}^\ell_{i=1})$, otherwise returns $(k_{S^*}, \{u_i\}^\ell_{i=1})$.

Guess: Adversary $\mathcal{A}$ outputs a guess γ' and the experiment outputs 1 if $\gamma' = \gamma$.

Definition 2 (Weak Security)**.** *A function* $F_t : \mathcal{K} \times \mathcal{X} \leftarrow \mathcal{Y}$ *is a weakly secure* t*-Punc-PRF if for all* λ *and PPT adversaries* $\mathcal{A}$*, its advantage defined below,* $\mathsf{Adv}^{t\text{-Punc-PRF}}_{F_t,\mathcal{A}}(\lambda) = |\Pr[\mathbf{Exp}^{t\text{-Punc-PRF}}_{F_t,\mathcal{A}}(\lambda) = 1] - 1/2| \leq \nu(\lambda)$ *where the probability is taken over the randomness of the experiment and* $\nu(\lambda)$ *is negligible.*

For our applications, we further require that the PRF key can be punctured at S one-by-one and the resulted punctured secret key for S be independent of the order of punctures. To be more precise, it is desired that the punctured secret key k_S for $S = \{x_1, x_2, \ldots, x_\ell\}$ s.t. $\ell \leq t(\lambda)$ can be computed in an alternative way:

$\mathsf{F}_t.\mathsf{Punc}(k_{i-1}, x'_i)$: On input a punctured key k_{i-1} for $S'_{i-1} = \{x'_1, x'_2, \ldots, x'_{i-1}\} \subset S$, where $k_0 = k$ is a randomly chosen PRF key, and a new element $x'_i \in S$, it generates a punctured key $k_i \in \mathcal{K}_p$ for $S'_i = S'_{i-1} \cup \{x'_i\}$. Finally, it outputs k_ℓ that is equal to $k_S \leftarrow \mathsf{F}_t.\mathsf{Punc}(k, S)$. Henceforth, all *multi-puncturable PRFs* we use in this work refer to the t-Punc-PRF with above property, unless stated otherwise.

D. Symmetric Searchable Encryption

A DSSE scheme Σ consists of one algorithm Setup and two protocols Search and Update: On input a security parameter λ and an initial database DB, Setup outputs a secret

key K, the state σ of the client, and an encrypted database EDB that will be sent to the server; Search takes a query q, the secret key K and the state σ from the client, as well as the database EDB from the server, and outputs the search result R matching q; Update takes the secret key K, the state σ, an input in and the associated operation op from the client, as well as the database EDB from the server, where op is either addition add or deletion del and in consists of a document identifier ind and a set of keywords w. Then it inserts in to or removes in from EDB, depending on op. Notice that in this work, we only consider search queries containing a single keyword w (i.e., $\mathsf{q} = w$).

A DSSE scheme $\Sigma = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ is *correct* if the Search protocol returns correct results for every query. For a formal definition, we refer the readers to [12].

SECURITY. The security of DSSE is parameterized by a stateful leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Updt})$ that captures the revealed information to an adversary $\mathcal{A}$ during the execution of the real scheme. In particular, $\mathcal{L}^{Stp}, \mathcal{L}^{Srch}$ and $\mathcal{L}^{Updt}$ correspond to the information leaked during Setup, Search and Update, respectively. Informally, the security ensures that $\mathcal{A}$ cannot learn more information beyond what can be referred from $\mathcal{L}$.

A formal security as defined in [12], [9] is captured by a real experiment $\text{REAL}^{\Sigma}_{\mathcal{A}}(\lambda)$ and an ideal experiment $\text{IDEAL}^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$:

$\text{REAL}^{\Sigma}_{\mathcal{A}}(\lambda)$: $\mathcal{A}$ chooses a database DB. The experiment runs $\mathsf{Setup}(1^{\lambda}, \text{DB})$ and returns EDB. Then $\mathcal{A}$ adaptively queries search (*resp.* update) on q (*resp.*, $(\mathsf{op}, \mathsf{in})$). In response, the experiment runs $\mathsf{Search}(\mathsf{q})$ (*resp.*, $\mathsf{Update}(\mathsf{op}, \mathsf{in})$) and returns the transcript of each operation. At last, $\mathcal{A}$ outputs a bit b.

$\text{IDEAL}^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$: $\mathcal{A}$ chooses a DB. The experiment returns EDB simulated by $\mathcal{S}(\mathcal{L}^{Stp}(\mathsf{DB}))$. Then $\mathcal{A}$ adaptively queries search (*resp.*, update) on q (*resp.*, $(\mathsf{op}, \mathsf{in})$), and receives the transcript simulated by $\mathcal{S}(\mathcal{L}^{Srch}(\mathsf{q}))$ (*resp.*, $\mathcal{S}(\mathcal{L}^{Updt}(\mathsf{op}, \mathsf{in}))$). Finally, $\mathcal{A}$ outputs a bit b.

Now we continue to recall *forward and backward privacy* of DSSE, initially formalized in [7], [9]. Informally, *forward privacy* ensures that each update leaks no information about the keyword contained in the keyword/document pair to be updated, while *backward privacy* means that when a keyword/document pair (w, ind) is added to and then deleted from the database, subsequent search on w does not reveal ind. As noted in [9], ind must be revealed if a search on w is performed after inserting (w, ind) but before removing it, so we only consider the case that no search happens between the addition and the deletion of the same keyword/document pairs. Ideally, a backward-private SSE should leak nothing about the deletions, and at least not reveal (the identifiers of) the deleted documents [51]. In terms of different levels of leakage, three types of backward privacy from Type-I to Type-III were introduced in [9]. They are defined through the above experiments by further imposing certain constraints on $\mathcal{L}$. Before proceeding, we first revisit the relevant functions needed for the formal definition of backward privacy. We follow the notation of [9] with minor modifications.

The leakage function $\mathcal{L}$ records a list of queries issued so far, *i.e.*, $Q = \{(u, w) \text{ or } (u, \mathsf{op}, \mathsf{in})\}$. Among the queries, (u, w) is a search query performed on timestamp u, which starts at 0 and increases with the coming query, and $(u, \mathsf{op}, \mathsf{in})$ is an update query where $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ and in is in the form of (w, ind) for single-keyword search. With query list Q, the relevant functions are defined as follows.

$\mathsf{sp}(w)$ is the search pattern over keyword w and consists of timestamps of all search queries on w. It describes which search queries are on the same keyword, formally defined as

$$\mathsf{sp}(w) = \{u : (u, w) \in Q\}.$$

$\mathsf{TimeDB}(w)$ is the extended access pattern [1], which consists of both the non-deleted documents matching w and the timestamps of inserting them to the database. Formally, it is

$$\mathsf{TimeDB}(w) = \big\{(u, ind) : \big(u, \mathsf{add}, (w, ind)\big) \in Q \text{ and } \forall u', \big(u', \mathsf{del}, (w, ind)\big) \notin Q\big\}.$$

$\mathsf{UpHist}(w)$ is the history of updates on keyword w and consists of all update queries on w. Formally, it is defined as

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

$\mathsf{Updates}(w)$ is the list of timestamp and operation pairs of updates on keyword w. Formally, it is defined as

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

We remark that the "$\mathsf{Updates}(w)$" here is slightly different from [9]. Here, it additionally contains the operation type op rather than only the corresponding timestamps. Actually, the "$\mathsf{UpTime}(w)$" defined below is the "$\mathsf{Updates}(w)$" in [9].

$\mathsf{UpTime}(w)$ is the update pattern over keyword w and consists of the timestamps of updates on w. Formally, it is

$$\mathsf{UpTime}(w) = \{u : (u, \mathsf{add}, (w, ind)) \text{ or } (u, \mathsf{del}, (w, ind)) \in Q\}.$$

$\mathsf{DelTime}(w)$ is the list of timestamps of the inserted documents (matching w) that were deleted later. Formally,

$$\mathsf{DelTime}(w) = \big\{u : \exists u', ind \text{ s.t. } \big(u', \mathsf{del}, (w, ind)\big) \in Q \text{ and } \big(u, \mathsf{add}, (w, ind)\big) \in Q\big\}.$$

Note that this is a new function we introduce, which is part of the "$\mathsf{Updates}(w)$" defined above and will be used for the security analysis of our scheme.

$\mathsf{DelHist}(w)$ is the deletion history of w and consists of both the timestamp of each deletion operation and the timestamp of the inserted entry it removes. Formally, it is defined as

$$\mathsf{DelHist}(w) = \big\{(u^{\mathsf{add}}, u^{\mathsf{del}}) : \exists\ ind \text{ s.t. } \big(u^{\mathsf{del}}, \mathsf{del}, (w, ind)\big) \in Q \text{ and } \big(u^{\mathsf{add}}, \mathsf{add}, (w, ind)\big) \in Q\big\}.$$

In contrast to $\mathsf{DelTime}(w)$, $\mathsf{DelHist}(w)$ additionally leaks which deletion removes which addition.

With these functions, now we are ready to formally define the notions of forward and backward privacy. We follow the definitions of [9], [15], [52], except that the search pattern $\mathsf{sp}(w)$ (a common leakage in most existing SSE schemes, *e.g.*, [43], [13], [9], [52]) is also included in the leakage function

[1]The regular access/result pattern [13] $\mathsf{DB}(w)$ can be computed as $\mathsf{DB}(w) = \{ind : \exists u \text{ s.t. } (u, ind) \in \mathsf{TimeDB}(w)\}$.

$\mathcal{L}^{Srch}$ in our definitions. Notice that, although $\mathsf{sp}(w)$ is not considered in the security notions of [9], [15], [52], it is in fact necessary for showing the security of their schemes.

Definition 3 (Forward Privacy [7], [9])**.** *A DSSE scheme $\Sigma = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ is $\mathcal{L}$-adaptively forward-private if for leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Updt})$ and all efficient $\mathcal{A}$ making at most $q(\lambda)$ queries, there exists a PPT algorithm $\mathcal{S}$ such that*

$$\left| \Pr[\textsc{Real}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] - \Pr[\textsc{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda) = 1] \right| \leq \nu(\lambda),$$

and the update leakage function $\mathcal{L}^{Updt}$ can be written as

$$\mathcal{L}^{Updt}(\mathsf{op}, (w, ind)) = \mathcal{L}'(\mathsf{op}, ind),$$

where $\nu(\lambda)$ is negligible in λ and $\mathcal{L}'$ is stateless.

Definition 4 (Backward Privacy [9])**.** *A DSSE scheme $\Sigma = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ is $\mathcal{L}$-adaptively Type-I/II/III -backward-private if for leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Updt})$ and all PPT adversary $\mathcal{A}$ making at most $q(\lambda)$ queries, there exists a PPT algorithm $\mathcal{S}$ such that*

$$\left| \Pr[\textsc{Real}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] - \Pr[\textsc{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda) = 1] \right| \leq \nu(\lambda),$$

and the search and update leakage functions $\mathcal{L}^{Srch}$ and $\mathcal{L}^{Updt}$ can be written as the following types, respectively:

Type-I : $\mathcal{L}^{Updt}(\mathsf{op}, (w, ind)) = \mathcal{L}'(\mathsf{op})$ *and* $\mathcal{L}^{Srch}(w) = \mathcal{L}''(\mathsf{sp}(w), \mathsf{TimeDB}(w), u_w)$,

Type-II : $\mathcal{L}^{Updt}(\mathsf{op}, (w, ind)) = \mathcal{L}'(\mathsf{op}, w)$ *and* $\mathcal{L}^{Srch}(w) = \mathcal{L}''(\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{UpTime}(w))$,

Type-III : $\mathcal{L}^{Updt}(\mathsf{op}, (w, ind)) = \mathcal{L}'(\mathsf{op}, w)$ *and* $\mathcal{L}^{Srch}(w) = \mathcal{L}''(\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{DelHist}(w))$,

where u_w is the total number of updates on w, $\nu(\lambda)$ is negligible and $\mathcal{L}'$, $\mathcal{L}''$ are stateless.

It can be seen that the three types of backward privacy from Type-I to Type-III are ordered from the most to the least secure. In this work, we are concerned with Type-II-backward privacy. According to the definition, a Type-II-backward-private SSE scheme may leak $\mathsf{sp}(w), \mathsf{TimeDB}(w)$, and $\mathsf{Updates}(w)$ that can be derived from op and $\mathsf{UpTime}(w)$. Actually, our Type-II-backward-private SSE scheme only leaks $\mathsf{sp}(w), \mathsf{TimeDB}(w)$, and $\mathsf{DelTime}(w)$ which is part of $\mathsf{Updates}(w)$. In contrast, the Type-III-backward-private SSE scheme additionally leaks $\mathsf{DelHist}(w)$ that captures which deletion cancels which addition.

III. Symmetric Revocable Encryption

In this section, we introduce a new cryptographic primitive, named Symmetric Revocable Encryption (SRE). Roughly, it resembles (a variant of) Symmetric Puncturable Encryption (SPE) [52], but in fact they are distinct in several aspects. More details about their similarities and differences will be discussed later. Next we will first formalize the syntax and security of SRE, and then propose a generic construction with a desirable feature for our application.

A. Syntax of SRE

An SRE scheme $\mathsf{SRE} = (\mathsf{SRE.KGen}, \mathsf{SRE.Enc}, \mathsf{SRE.KRev}, \mathsf{SRE.Dec})$ with key space $\mathcal{MSK}$, message space $\mathcal{M}$ and tag space $\mathcal{T}$ includes four polynomial-time algorithms:

$\mathsf{SRE.KGen}(1^\lambda)$: It takes a security parameter λ as input and outputs a system secret key $msk \in \mathcal{MSK}$.

$\mathsf{SRE.Enc}(msk, m, T)$: It takes as input a system secret key msk and a message $m \in \mathcal{M}$ with a list of tags $T \subseteq \mathcal{T}$, and outputs a ciphertext ct for m under tags T.

$\mathsf{SRE.KRev}(msk, R)$: It takes as input a system secret key msk and a revocation list $R \subseteq \mathcal{T}$, and outputs a revoked secret key sk_R, which can be used to decrypt only the ciphertext that has no tag belonging to R.

$\mathsf{SRE.Dec}(sk_R, ct, T)$: It takes as input a revoked secret key sk_R and a ciphertext ct encrypted under tags T, and outputs the message m or a failure symbol $\perp$.

Definition 5 (Correctness)**.** *For all security parameter $\lambda \in \mathbb{N}$, message $m \in \mathcal{M}$, tag list $T \subseteq \mathcal{T}$ and revocation list $R \subseteq \mathcal{T}$ s.t. $R \cap T = \emptyset$, the probability*

$$\Pr\left[\mathsf{SRE.Dec}(sk_R, ct, T) = m : \begin{array}{l} msk \leftarrow \mathsf{SRE.KGen}(1^\lambda) \\ ct \leftarrow \mathsf{SRE.Enc}(msk, m, T) \\ sk_R \leftarrow \mathsf{SRE.KRev}(msk, R) \end{array} \right]$$

is at least $1 - \nu(\lambda)$. That is, the correctness error is upper-bounded by a possibly non-negligible function $\nu(\cdot)$.

Notice that, the regular definition of correctness requires that a ciphertext under tag list T can be decrypted (with an overwhelming probability) by a revoked secret key on revocation list R only if $R \cap T = \emptyset$. Here, we define it in a relaxed manner, allowing for a non-negligible correctness error, which is sufficient for our applications.

B. Security of SRE

The semantic security of SRE is defined via an IND-REV-CPA experiment, denoted by $\mathbf{Exp}^{\text{IND-REV-CPA}}_{\mathsf{SRE},\mathcal{A}}(\lambda)$. In the experiment, the adversary $\mathcal{A}$ is given access to an $\mathsf{Encryption}$ oracle, by which $\mathcal{A}$ can get the ciphertext of any message under a list of tags, and a $\mathsf{Key\ Revocation}$ oracle, by which $\mathcal{A}$ can obtain a revoked secret key for any revocation list chosen by herself. In particular, the experiment is described as follows:

Setup: On input a parameter λ, the experiment runs $msk \leftarrow \mathsf{SRE.KGen}(1^\lambda)$ and initializes an empty set $\mathcal{Q}$.

Phase 1: $\mathcal{A}$ can adaptively issue the following queries

- **Encryption**(m, T): On input a message m and a list T of attached tags, the experiment runs $ct \leftarrow \mathsf{SRE.Enc}(msk, m, T)$ and returns the ciphertext ct.
- **Key Revocation**(R): On input a revocation list R, the experiment runs $sk_R \leftarrow \mathsf{SRE.KRev}(msk, R)$. It then returns sk_R and adds R to $\mathcal{Q}$.

Challenge: On input messages $m_0, m_1 \in \mathcal{M}$ along with tag list $T^* \subseteq \mathcal{T}$, the experiment rejects the query if there exists $R \in \mathcal{Q}$ such that $T^* \cap R = \emptyset$. Otherwise, it chooses $\gamma \xleftarrow{\$} \{0, 1\}$ and returns $ct^* \leftarrow \mathsf{SRE.Enc}(msk, m_\gamma, T^*)$.

Phase 2: This is identical to Phase 1 except $\mathcal{A}$ is disallowed to ask for any key revocation query R *s.t.* $R \cap T^* = \emptyset$.

Guess: $\mathcal{A}$ outputs γ' and the experiment returns 1 if $\gamma' = \gamma$.

Definition 6 (Adaptive Security)**.** *An SRE scheme is IND-REV-CPA secure if for all $\lambda \in \mathbb{N}$ and PPT adversary $\mathcal{A}$, the advantage of $\mathcal{A}$ winning in the experiment*

$$\mathsf{Adv}^{\text{IND-REV-CPA}}_{\mathsf{SRE},\mathcal{A}}(\lambda) = |\Pr[\mathbf{Exp}^{\text{IND-REV-CPA}}_{\mathsf{SRE},\mathcal{A}}(\lambda) = 1] - 1/2|$$

is at most $\nu(\lambda)$, where the probability is taken over the coins of the experiment and $\nu(\lambda)$ is negligible.

Similarly, we can define *selective* security of SRE by an experiment $\mathbf{Exp}^{\text{IND-sREV-CPA}}_{\mathsf{SRE},\mathcal{A}}(\lambda)$, which is identical to above experiment except T^* is submitted before the Setup phase.

Definition 7 (Selective Security)**.** *An SRE scheme is IND-sREV-CPA secure if for all $\lambda \in \mathbb{N}$ and PPT adversary $\mathcal{A}$, the advantage of $\mathcal{A}$ winning in the experiment*

$$\mathsf{Adv}^{\text{IND-sREV-CPA}}_{\mathsf{SRE},\mathcal{A}}(\lambda) = |\Pr[\mathbf{Exp}^{\text{IND-sREV-CPA}}_{\mathsf{SRE},\mathcal{A}}(\lambda) = 1] - 1/2|$$

is at most $\nu(\lambda)$, where the probability is taken over the coins of the experiment and $\nu(\lambda)$ is negligible.

Now we complete the description of the syntax and security of SRE. Before going ahead, we first clarify the similarities and differences between SRE and SPE [52]. In general, SPE focuses on achieving forward security by updating the secret key gradually, while SRE emphasizes the functionality of revoking the decryption capability of the master secret key. Although both of them can be used to revoke a list of tags, SRE is more relaxed than SPE and provides a different type of security. In particular, a set of tags are revoked *separately* in SPE, by repeatedly running the puncture algorithm of SPE (*i.e.*, $SK_i \leftarrow \mathsf{Pun}(SK_{i-1}, t_i)$ [2] where $SK_0 = msk$, following the notation of [52]), while they may be revoked in an *arbitrary* manner in SRE. Therefore, SPE can be thought of as a specific case of SRE from the perspective of functionality.

At the first glance, it seems that an SRE scheme can be constructed in a trivial and black-box way from any SPE scheme by repeatedly invoking the puncture algorithm of SPE, as indicated above. That is, the algorithm $\mathsf{SRE.KRev}(msk, R)$ generates a revoked secret key sk_R as: Given a master secret key msk and a list of tags, say $R = \{t_1, t_2, \ldots, t_\tau\}$, it recursively calls $SK_i \leftarrow \mathsf{Pun}(SK_{i-1}, t_i)$ from $i = 1$ to τ, and finally sets $sk_R = SK_\tau$. As demonstrated below, however, this does not hold as the security of SPE cannot imply that of SRE. In particular, we present a counter-example that is derived from a secure SPE but cannot satisfy the security of SRE.

Let $\mathsf{SPE} = (\mathsf{SPE.Setup}, \mathsf{SPE.Enc}, \mathsf{SPE.Pun}, \mathsf{SPE.Dec})$ be a semantically secure SPE scheme supporting one tag per message, as shown in [52]. Then we construct a new SPE scheme $\mathsf{SPE}' = (\mathsf{SPE}'.\mathsf{Setup}, \mathsf{SPE}'.\mathsf{Enc}, \mathsf{SPE}'.\mathsf{Pun}, \mathsf{SPE}'.\mathsf{Dec})$ that supports two tags per message, and show that SPE' is secure but the SRE scheme derived from SPE' by following the above way is not. Specifically, the SPE' is constructed as follows:

$\mathsf{SPE}'.\mathsf{Setup}(1^\lambda)$: It is the same as the $\mathsf{SPE.Setup}$ algorithm of SPE, which takes as input a security parameter λ and outputs an initial master secret key msk.

$\mathsf{SPE}'.\mathsf{Enc}(msk, m, T)$: On input a msk and a message m attached with tags $T = \{t_1, t_2\}$, it chooses randomly $k_1, k_2 \in \mathcal{M}$ and outputs the ciphertext $ct = (ct_0, ct_1, ct_2)$, such that $ct_0 = k_1 \oplus k_2 \oplus m,\ ct_1 \leftarrow \mathsf{SPE.Enc}(msk, k_1, t_1),\ ct_2 \leftarrow \mathsf{SPE.Enc}(msk, k_2, t_2)$.

$\mathsf{SPE}'.\mathsf{Pun}(SK_{i-1}, t'_i)$: It is identical to the $\mathsf{SPE.Pun}$ algorithm of SPE that further punctures SK_{i-1} on tag t'_i and outputs a new secret key SK_i.

$\mathsf{SPE}'.\mathsf{Dec}(SK_i, ct, T)$: On input a secret key SK_i associated with tags $R = \{t'_1, t'_2, \ldots, t'_i\}$ and a ciphertext $ct = (ct_0, ct_1, ct_2)$ under tags $T = \{t_1, t_2\}$, it outputs $\perp$ if $R \cap T \neq \emptyset$. Otherwise,

1) Compute $k'_j = \mathsf{SPE.Dec}(SK_i, ct_j, t_j)$ for $j \in \{1, 2\}$.
2) Return $m' = ct_0 \oplus k'_1 \oplus k'_2$.

It can be seen that SPE' is correct, as k'_j can be correctly recovered (*i.e.*, $k'_j = k_j$) if $t_j \notin R$, following the correctness of SPE [52]. On the other hand, SPE' is secure under the security of the underlying SPE. Informally, it is required that at least one challenge tag, say $t^*_1 \in T^*$, be punctured by a valid adversary during the query phase, then k^*_1 is completely hidden in ct^*_1 due to the semantical security of SPE, and so the challenge message m^*_γ is perfectly concealed. However, the SRE scheme derived from SPE' is not secure. Recall that, in the IND-REV-CPA game of SRE the adversary is allowed to ask for a number of key revocation queries on R with the only restriction that $R \cap T^* \neq \emptyset$. In this case, the security of the SRE scheme can be broken by an efficient adversary $\mathcal{A}$ as follows: $\mathcal{A}$ issues two key revocation queries on R_1 and R_2, where for the challenge tags $T^* = \{t^*_1, t^*_2\}$ the queries satisfy the conditions that $t^*_1 \in R_1,\ t^*_2 \notin R_1$ and $t^*_2 \in R_2,\ t^*_1 \notin R_2$. After receiving the revoked secret keys sk_{R_1} and sk_{R_2}, $\mathcal{A}$ uses sk_{R_1} and sk_{R_2} to compute $k^*_2 \leftarrow \mathsf{SPE.Dec}(sk_{R_1}, ct^*_2, t^*_2)$ and $k^*_1 \leftarrow \mathsf{SPE.Dec}(sk_{R_2}, ct^*_1, t^*_1)$, respectively. Finally, $\mathcal{A}$ recovers $m^*_\gamma = ct^*_0 \oplus k^*_1 \oplus k^*_2$.

In other words, the above attack indicates that distinct revoked secret keys could be combined together to decrypt some ciphertext that cannot be decrypted by each one separately. This kind of attack is always termed as collusion attack, which is not captured by SPE schemes. In contrast, the security of SRE implies collusion-resistance, meaning that the combination of revoked secret keys sk_R and $sk_{R'}$ cannot decrypt the ciphertext under tags T if $R \cap T \neq \emptyset$ and $R' \cap T \neq \emptyset$. This property plays an important role in multi-client settings, *e.g.*, multi-client SSE [37].

By this time, we have shown that the above intuition does not work for building SRE schemes that support at least two tags per message. However, we are not sure whether or not it works for the case of SRE scheme supporting one tag per message, since it is unclear how to reduce the collusion-resistance of SRE to the security of SPE. Notice that, even if it could work for the later case, we believe that SRE scheme can be constructed in more efficient ways, as it is more relaxed and general than SPE.

Alternatively, SRE can be seen as a symmetric predicate encryption for the predicate that $\mathsf{P}(R, T) = 1$ *if* $R \cap T = \emptyset$ and 0 otherwise, which can provide more fine-grained access control over encrypted data as exemplified in [44]. Since the

[2] It takes as input SK_{i-1} and tag t_i, and outputs a new secret key SK_i, which can decrypt the ciphertexts that SK_{i-1} can other than those encrypted under t_i. For more details of SPE, please refer to the Section 3 of [52].

predicate $\mathsf{P}(R,t)=1$ *iff* $t \notin R$ for $t \in \mathcal{T}$ and $R \subseteq \mathcal{T}$ suffices our application, we are mainly interested in constructing efficient SRE schemes with a single tag per message in the following, and leaving more general constructions in future.

C. Construction of SRE

In this section, we propose an SRE scheme based on a multi-puncturable PRF, a standard symmetric encryption scheme and a Bloom filter. At the first glance, the way of encrypting a message is similar to that of Bloom Filter Encryption (BFE) [24] [3]; A message is encrypted under h indices (of the Bloom filter) derived from the tag of the message. However, the way of revoking the decryption capability of the secret key between BFE and our SRE is essentially different. In BFE, the key idea is to associate the key pair of the scheme to a Bloom filter, where the initial secret key consists of b parts with each part corresponding to an entry index of the Bloom filter (that contains b entries), and to realize puncturing by simply deleting the corresponding parts of the secret key. In contrast, the initial secret key of our scheme is independent of the Bloom filter, and we revoke it on a set of tags in one shot by leveraging the multi-puncturable PRF [36]. The Bloom filter here is used *merely* for recording and compressing all tags to be revoked. This results in a low storage request before launching the revocation. We call this feature *Compressed Revocation*, which is crucial for our application and will be formalized and explained with more details after the construction.

Next we present the detailed construction. Let $\mathsf{SE} = (\mathsf{SE.Gen}, \mathsf{SE.Enc}, \mathsf{SE.Dec})$ be a standard SE scheme with key space $\mathcal{Y}$, $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ a multi-puncturable PRF with algorithms ($\mathsf{MF.Setup}$, $\mathsf{MF.Punc}$, $\mathsf{MF.Eval}$), and $\mathsf{BF} = (\mathsf{BF.Gen}, \mathsf{BF.Upd}, \mathsf{BF.Check})$ a (b,h,n)-Bloom filter, where n is the maximum number of elements to be inserted and b, h are the numbers of Bloom filter entries and hash functions, respectively. Then our SRE scheme $\mathsf{SRE} = (\mathsf{SRE.KGen}, \mathsf{SRE.Enc}, \mathsf{SRE.KRev}, \mathsf{SRE.Dec})$ is described as follows:

$\mathsf{SRE.KGen}(1^\lambda, b, h)$: It takes a security parameter λ and integers $b, h \in \mathbb{N}$, and generates the system secret key msk by

1) Runs $(H, B) \leftarrow \mathsf{BF.Gen}(b,h)$, such that $H = \{H_j\}_{j\in[h]}$ and $B = 0^b$.
2) Generates $sk \leftarrow \mathsf{MF.Setup}(1^\lambda)$, and outputs $msk = (sk, H, B)$. Note H can be set as public parameters.

$\mathsf{SRE.Enc}(msk, m, t)$: It takes $msk = (sk, H, B)$ and a message $m \in \mathcal{M}$ with tag $t \in \mathcal{T}$, and outputs ciphertext ct as

1) Computes $i_j = H_j(t) \in [b]$ and $sk_{i_j} = F(sk, i_j)$ for all $j \in [h]$.
2) Generates $ct_j = \mathsf{SE.Enc}(sk_{i_j}, m)$ for all $j \in [h]$, and returns the ciphertext $ct = (ct_1, ct_2, \ldots, ct_h)$ and t.

$\mathsf{SRE.KRev}(msk, R)$: It takes $msk = (sk, H, B)$ and a list $R = \{t_1, t_2, \ldots, t_\tau\}$ of tags to be revoked *s.t.* $\tau \le n$, then generates the revoked secret key sk_R for R:

1) Computes $B_R \leftarrow \mathsf{BF.Upd}(H, B, R)$, by which the entries of B indexed by $\{H_j(t_i)\}_{i\in[\tau], j\in[h]}$ are set to 1 (*i.e.*, $B[H_j(t_i)] \leftarrow 1$) for all $t_i \in R$.
2) Finds the index set $I = \{i' \in [b] : B_R[i'] = 1\}$ from B_R, then computes $sk_I \leftarrow \mathsf{MF.Punc}(sk, I)$ and sets $sk_R = (sk_I, H, B_R)$.

[3] It is a puncturable encryption focusing on highly efficient puncturing.

$\mathsf{SRE.Dec}(sk_R, ct, t)$: It takes a ciphertext $ct = (ct_1, ct_2, \ldots, ct_h)$ encrypted under tag t and a revoked secret key $sk_R = (sk_I, H, B_R)$, then recovers the message as:

1) Checks if $\mathsf{BF.Check}(H, B_R, t) = 1$. If true, the decryption fails. Otherwise,
2) Finds an index $i^* \in [b]$ *s.t.* $B_R[i^*] = 0$ (*i.e.*, $i^* \notin I$ derived from R), and then computes $sk_{i^*} = \mathsf{MF.Eval}(sk_I, i^*)$.
3) Finally computes $m = \mathsf{SE.Dec}(sk_{i^*}, ct_{i^*})$.

CORRECTNESS. According to Definition 5, the choices of R and $T = \{t\}$ are independent of the randomness (*i.e.*, H) used to construct the Bloom filter in our construction, so it is sufficient to consider the correctness of Bloom filter in the traditional setting than the adversarial environment [19]. It can be seen that the revoked secret key is generated in a compressed manner (exactly based on the Bloom filter), so our scheme introduces a non-negligible correctness error. Next we show it is up-bounded by the false-positive probability of the standard Bloom filter. In particular, we suppose that a revoked secret key $sk_R = (sk_I, H, B_R)$ is associated with revocation list $R = \{t_1, t_2, \ldots, t_\tau\}$ and a ciphertext $ct = (ct_1, ct_2, \ldots, ct_h)$ is generated under tag t such that $t \notin R$, then we can use sk_R to decrypt ct if $\mathsf{BF.Check}(H, B_R, t) = 0$, because this ensures that there exists $i^* \in [h]$ *s.t.* $B_R[i^*] = 0$ (*i.e.*, $i^* \notin I$ derived from R) and we can compute $sk_{i^*} = \mathsf{MF.Eval}(sk_I, i^*) = F(sk, i^*)$ and recover $m = \mathsf{SE.Dec}(sk_{i^*}, ct_{i^*})$. The correctness follows from that of the multi-puncturable PRF F and SE. Otherwise (*i.e.*, $\mathsf{BF.Check}(H, B_R, t) = 1$), the decryption fails, because this means $B_R[H_j(t)] = 1$ for all $j \in [h]$ (*i.e.*, $\{H_j(t)\} \subseteq I$) and we cannot get any $sk_{H_j(t)}$ to decrypt any part of ct. This indicates that the correctness error is exactly the false-positive probability of the Bloom filter, thus we have $\Pr[\mathsf{SRE.Dec}(sk_R, ct, t) = \perp] = \Pr[t \notin R \wedge \mathsf{BF.Check}(H, B_R, t) = 1] \approx 2^{-h}$, where h is the parameter of Bloom filter.

Note that, the revoked secret key of our SRE scheme is always computed from the initial (or master) secret key, so the scheme does not provide *forward security* as guaranteed by puncturable encryption [32], [24]. Nevertheless, it is sufficient for our SSE application, as what we are concerned is the ability of revoking the master secret key with a low storage request. In addition, the size of the revoked secret key in our scheme mainly depends on that of the punctured secret key of pseudorandom PRF F, which in our instantiation is about $O(\log b)$ on average instead of $O(b)$ as in BFE [24].

COMPRESSED REVOCATION. In above scheme, the secret key is revoked on all tags R at a time. Intuitively, to revoke a number of tags *in one shot*, certain storage should be allocated for them before launching the revocation. Usually, the storage cost is dominated by the total size of all the tags to be revoked, which is undesirable in SSE application. To avoid this issue, we first *compress* all tags one-by-one by leveraging a *compact* data structure, thus getting a compressed revocation list at last, and then generate the revoked secret key based on the compact data structure. By this way, only a low storage cost is introduced on the client side. Specifically, the key-revocation procedure $\mathsf{SRE.KRev}(\cdot,\cdot)$ of our construction can be split into two abstract sub-algorithms, formalized as below:

$\mathsf{SRE.KRev}(sk, D, R)$: On input a secret key sk, the description D of a short-sized data structure initialized at the

setup of the system, and a list $R = \{t_1, t_2, \ldots, t_\tau\}$ of tags to be revoked, it generates the revoked secret key sk_R as below:

1) $D \leftarrow \mathsf{SRE.Comp}(D, t_i)$: Inserts each $t_i \in R$ individually to the (initially-empty) data structure D and finally gets a compressed revocation list of R.
2) $sk_R \leftarrow \mathsf{SRE.cKRev}(sk, D)$: Computes the revoked secret key sk_R based on the compressed revocation list D that is obtained after adding all tags in R to the data structure.

Hereafter, we refer to an SRE scheme enjoying this feature as a *Compressed* SRE (CSRE for short).

The data structure D employed in our concrete construction is exactly a $(b, h, n)-$ Bloom-filter, where b, h and n are the numbers of Bloom filter entries, different hash functions, and the elements to be inserted, respectively. Assuming the false-positive probability tolerated is p and the optimal number of hash functions used is h, then the required size of Bloom filter in our construction is $b = -n \ln p/(\ln 2)^2$ bits. For example, when $p = 10^{-5}$ and $n = 2^{20}$, the size b of Bloom filter we need is about 3 MB. With such kind of compressed revocation, the client needs only a short and constant-size storage for launching the revocation, which exactly consists of $|sk|+b$ bits in our scheme, and the revoked secret key will be generated and sent immediately to the server when performing each search.

SECURITY. Our compressed SRE scheme is IND-sREV-CPA secure under the security of the underlying primitives, which is formally stated in Theorem 1.

Theorem 1. *The Compressed SRE scheme is IND-sREV-CPA secure, if BF is a (b, h, n)-Bloom-filter, $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is a secure b-Punc-PRF and SE is IND-CPA secure. Particularly, for all PPT algorithms $\mathcal{A}$ in the IND-sREV-CPA game, there exist PPT algorithms $\mathcal{B}$ and $\mathcal{B}'$ such that*

$$\mathsf{Adv}^{\text{IND-sREV-CPA}}_{\mathsf{SRE},\mathcal{A}}(\lambda) \leq 2\mathsf{Adv}^{b\text{-Punc-PRF}}_{F,\mathcal{B}}(\lambda) + 2h \cdot \mathsf{Adv}^{\text{IND-CPA}}_{\mathsf{SE},\mathcal{B}'}(\lambda).$$

Proof of Theorem 1: The proof proceeds with a sequence of games. It starts with the real game and ends with a game where the adversary's advantage is 0. In each game, we call Win_i the event that the adversary $\mathcal{A}$ wins in game G_i.

Game G_0: This is the real game for selective security of SRE. Namely, after receiving tag t^* that $\mathcal{A}$ wishes to be challenged upon, the challenger produces $msk = (sk, H, B)$ by running $msk \leftarrow \mathsf{SRE.KGen}(1^\lambda, b, h)$, and then answers all $\mathcal{A}$'s queries with msk. For an encryption query (m, t), it computes $i_j = H_j(t)$ and $sk_{i_j} = F(sk, i_j)$ for all $j \in [h]$, and outputs $ct = (ct_1, ct_2, \ldots, ct_h)$ *s.t.* $ct_j \leftarrow \mathsf{SE.Enc}(sk_{i_j}, m)$ for $j \in [h]$. For a key-revocation query $R \subseteq \mathcal{T}$, the challenger computes $B_R \leftarrow \mathsf{BF.Upd}(H, B, R)$ and finds the index set $I = \{i' \in [b] : B_R[i'] = 1\}$, then it computes $sk_I \leftarrow \mathsf{MF.Punc}(sk, I)$ and returns $sk_R = (sk_I, H, B_R)$. In the challenge phase, $\mathcal{A}$ comes up with two distinct messages $m_0, m_1 \in \mathcal{M}$, and receives a challenge ciphertext $ct^* \leftarrow \mathsf{SRE.Enc}(msk, m_\gamma, t^*)$ for $\gamma \xleftarrow{\$} \{0, 1\}$. Finally, $\mathcal{A}$ outputs her guess $\gamma' \in \{0, 1\}$ when she halts. We note that for each key-revocation query R, it must hold that $t^* \in R$ if $\mathcal{A}$ is a legitimate adversary. From the Definition 7, we know that $\mathsf{Adv}^{\text{IND-sREV-CPA}}_{\mathsf{SRE},\mathcal{A}}(\lambda) = |\Pr[\mathsf{Win}_0] - 1/2|$.

Game G_1: In this game, we modify the way of generating ciphertexts. Namely, we produce in advance all PRF values $sk_i = F(sk, i)$ for all $i \in [b]$, and use them straightforwardly to compute the challenge ciphertext ct^* and those for encryption queries. The other queries are responded in the same way as before. Obviously, it holds that $\Pr[\mathsf{Win}_1] = \Pr[\mathsf{Win}_0]$.

Game G_2: This game is identical to G_1, apart from the generation of the revoked secret key. Namely, the revoked secret key sk_R for each key-revocation query R is generated in the following way:

1) Computes $B_R \leftarrow \mathsf{BF.Upd}(H, B, R)$ and finds out the index set $I = \{i' \in [b] : B_R[i'] = 1\}$.
2) Sets $I_{t^*} = \{H_j(t^*)\}_{j\in[h]}$, which is a subset of I due to $t^* \in R$, and computes $sk_{t^*} \leftarrow \mathsf{MF.Punc}(sk, I_{t^*})$.
3) Computes $sk_I \leftarrow \mathsf{MF.Punc}(sk_{t^*}, I \setminus I_{t^*})$ and returns $sk_R = (sk_I, H, B_R)$.

In other words, the punctured secret key sk_I in this scheme is computed in the alternative way, as mentioned in Section II-C, which ensures that the sk_I in both games are identically distributed. Thus, we have that $\Pr[\mathsf{Win}_2] = \Pr[\mathsf{Win}_1]$.

Game G_3: We further modify the way of producing PRF values $\{sk_i\}_{i\in[b]}$ in G_1. Namely, we first compute the index set $I_{t^*} = \{H_j(t^*)\}_{j\in[h]}$ corresponding to t^*, select $u_i \xleftarrow{\$} \mathcal{Y}$ for all $i \in I_{t^*}$, and then set the PRF values $\{sk_i\}_{i\in[b]}$ as:

$$sk_i = \begin{cases} F(sk, i), & i \in [b] \setminus I_{t^*} \\ u_i, & i \in I_{t^*} \end{cases}.$$

In other words, all PRF values associated with t^* (*i.e.*, $\{sk_i\}_{i\in I_{t^*}}$) are replaced by random ones. In consequence, the challenge chiphertext ct^* and the ciphertexts corresponding to the encryption queries on t^* will be simulated directly with these random values. As for key-revocation queries, they are processed similarly as before.

Under the security of multi-puncturable PRF, we argue that the above modification changes the distribution of $\mathcal{A}$'s view only negligibly, as stated in Lemma 1.

Lemma 1. *Suppose that $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is a secure b-Punc-PRF, then games G_3 and G_2 are computationally indistinguishable for all PPT adversary, which means that*

$$|\Pr[\mathsf{Win}_3] - \Pr[\mathsf{Win}_2]| \leq 2\mathsf{Adv}^{b\text{-Punc-PRF}}_{F,\mathcal{B}}(\lambda).$$

Proof of Lemma 1: We assume by contradiction that there exists an adversary $\mathcal{A}$ that can distinguish G_3 and G_2 with non-negligible advantage. Then we can construct an efficient reduction $\mathcal{B}$ that can break the security of the multi-puncturable PRF F. The algorithm $\mathcal{B}$ proceeds as follows.

$\mathcal{B}(1^\lambda)$ first obtains $t^* \in \mathcal{T}$ that $\mathcal{A}$ wishes to be challenged upon, and generates $(H, B) \leftarrow \mathsf{BF.Gen}(b, h)$, where $H = \{H_j\}_{j\in[h]}$ and $B = 0^b$. Then it computes $I_{t^*} = \{i^*_j : i^*_j = H_j(t^*)\}_{j\in[h]}$ and replays I_{t^*} to the challenger of the multi-puncturable PRF F. The challenger responds to $\mathcal{B}$ by running $sk \leftarrow \mathsf{MF.Setup}(1^\lambda)$ and generating $(sk_{t^*}, \{y^*_j\}_{j\in[h]})$:

- Computes $sk_{t^*} \leftarrow \mathsf{MF.Punc}(sk, I_{t^*})$ and $y_{j,0} = F(sk, i^*_j)$ for all $i^*_j \in I_{t^*}$.
- Chooses $y_{j,1} \xleftarrow{\$} \mathcal{Y}$, $\delta \xleftarrow{\$} \{0, 1\}$ for all $j \in [h]$, and sets $y^*_j = y_{j,0}$ if $\delta = 0$ and $y^*_j = y_{j,1}$ otherwise.

$\mathcal{B}$ then sets all PRF values $\{sk_i\}_{i\in[b]}$ as:

$$sk_i = \begin{cases} \mathsf{MF.Eval}(sk_{t^*}, i), & i \in [b] \setminus I_{t^*} \\ y^*_j, & i \in I_{t^*} \end{cases}$$

where for each $i \notin I_{t^*}$ we have $sk_i = \mathsf{MF.Eval}(sk_{t^*}, i) = F(sk, i)$ and for $i \in I_{t^*}$ (*i.e.*, $\exists j$ *s.t.* $i = i_j^*$) $sk_i = sk_{i_j^*} = y_j^*$. After that, $\mathcal{B}$ uses them together with sk_{t^*} to simulate all following queries.

For an encryption query (m, t), $\mathcal{B}$ computes $i_j = H_j(t)$ and $ct_j \leftarrow \mathsf{SE.Enc}(sk_{i_j}, M)$ for all $j \in [h]$, where sk_{i_j} is set as above. Then it returns $ct = (ct_1, ct_2, \ldots, ct_h)$ to $\mathcal{A}$.

For a key-revocation query R, $\mathcal{B}$ generates the revoked secret key sk_R in the following way:

1) Computes $B_R \leftarrow \mathsf{BF.Upd}(H, B, R)$ and finds out the index set $I = \{i' \in [b] : B_R[i'] = 1\}$, which includes $I_{t^*} = \{H_j(t^*)\}_{j \in [h]}$ due to $t^* \in R$.
2) Computes $sk_I \leftarrow \mathsf{MF.Punc}(sk_{t^*}, I \setminus I_{t^*})$ and returns $sk_R = (sk_I, H, B_R)$.

In the challenge phase, $\mathcal{B}$ receives two distinct messages m_0, m_1 from $\mathcal{A}$ and picks $\gamma \xleftarrow{\$} \{0, 1\}$. Then it computes $ct_j^* \leftarrow \mathsf{SE.Enc}(sk_{i_j^*}, m_\gamma)$ for all $j \in [h]$, where $sk_{i_j^*} = y_j^*$, and returns $ct^* = (ct_1^*, ct_2^*, \ldots, ct_h^*)$. Eventually, $\mathcal{A}$ outputs her guess $\gamma' \in \{0, 1\}$, and $\mathcal{B}$ returns $\delta' = 1$ if $\gamma' = \gamma$.

It can be seen that $\mathcal{B}$ perfectly simulates game G_2 when $\delta = 0$ (*i.e.*, $y_j^* = F(sk, i_j^*)$) and G_3 when $\delta = 1$ (*i.e.*, $y_j^* \xleftarrow{\$} \mathcal{Y}$), so we have $\Pr[\delta' = 1 | \delta = 0] = \Pr[\mathsf{Win}_2]$ and $\Pr[\delta' = 1 | \delta = 1] = \Pr[\mathsf{Win}_3]$, and the advantage of $\mathcal{B}$ is

$$\begin{aligned} \mathsf{Adv}_{F,\mathcal{B}}^{m\text{-Punc-PRF}}(\lambda) &= \tfrac{1}{2} \left| \Pr[\delta' = 1 | \delta = 1] - \Pr[\delta' = 1 | \delta = 0] \right| \\ &= \tfrac{1}{2} \left| \Pr[\mathsf{Win}_3] - \Pr[\mathsf{Win}_2] \right|. \end{aligned}$$

This completes the proof of Lemma 1. ∎

Game G_4: This differs from G_3 only in the generation of ct^*. Namely, $ct^* = (ct_1^*, ct_2^*, \ldots, ct_h^*)$ in this game is computed as the encryption of constant string 0's, instead of m_γ. More precisely, each ct_j^* for $j \in [h]$ is generated as $ct_j^* \leftarrow \mathsf{SE.Enc}(sk_{i_j^*}, 0^{|m|})$, where $sk_{i_j^*} \xleftarrow{\$} \mathcal{Y}$ and $|m|$ denotes the bit-length of the message $m \in \mathcal{M}$. We observe that the random bit $\gamma \in \{0, 1\}$ is independent of this game, so we have that $\Pr[\mathsf{Win}_4] = 1/2$.

Next, we proceed to show that the views of $\mathcal{A}$ in both G_4 and G_3 are computationally indistinguishable under the IND-CPA security of SE, as stated formally in Lemma 2.

Lemma 2. *Suppose that* $\mathsf{SE} = (\mathsf{SE.Gen}, \mathsf{SE.Enc}, \mathsf{SE.Dec})$ *is an IND-CPA secure symmetric encryption scheme, then* G_3 *and* G_4 *are computationally indistinguishable for all PPT adversaries* $\mathcal{A}$*. That is,*

$$\left| \Pr[\mathsf{Win}_4] - \Pr[\mathsf{Win}_3] \right| \leq 2h \cdot \mathsf{Adv}_{\mathsf{SE},\mathcal{B}'}^{\text{IND-CPA}}(\lambda),$$

where $\mathcal{B}'$ *is a PPT adversary against the IND-CPA security.*

This lemma can be proven by the standard hybrid argument. For more details, please refer to Appendix A-A.

Given all above lemmas, we get the advantage of any PPT adversary $\mathcal{A}$ attacking our scheme:

$$\begin{aligned} \mathsf{Adv}_{\mathsf{SRE},\mathcal{A}}^{\text{IND-sREV-CPA}}(\lambda) &= \left| \Pr[\mathsf{Win}_0] - \Pr[\mathsf{Win}_4] \right| \\ &\leq 2\mathsf{Adv}_{F,\mathcal{B}}^{b\text{-Punc-PRF}}(\lambda) \\ &+ 2h \cdot \mathsf{Adv}_{\mathsf{SE},\mathcal{B}'}^{\text{IND-CPA}}(\lambda). \end{aligned}$$

This concludes the proof of Theorem 1. ∎

IV. Backward-Private SSE from CSRE

Next we propose a generic construction of forward and backward private SSE from CSRE, and show it can achieve Type-II backward privacy within a single roundtrip.

A. Generic Construction

Our construction follows the essential idea of Janus [9] and Janus++ [52]. That is, the document identifiers are encrypted in a way that the deleted documents cannot be decrypted, even if their ciphertexts can be retrieved by the server. To make clear the differences between them, we first give a brief introduction to Janus++. In general, it is based on two forward-private SSE instances Σ_{add} and Σ_{del}, one for addition and the other for deletion. More specifically, each time a new document identifier/keyword pair (ind, w) is inserted, the identifier is encrypted with SPE and stored to the server by employing Σ_{add}. Similarly, whenever a pair (ind, w) is deleted, an associated punctured key element is generated and outsourced to the server by leveraging Σ_{del}. To search on keyword w, the client runs the search protocol of both Σ_{add} and Σ_{del}. Consequently, the server retrieves all encrypted identifies containing w and all remaining key elements w.r.t. w from the instance Σ_{add} and Σ_{del}, respectively. Then the server uses the secret key to decrypt all documents that are not deleted.

Compared to Janus++, our construction uses only one forward-private SSE instance Σ_{add}. Particularly, it is employed to store the newly inserted document identifiers encrypted with a CSRE scheme, which realizes the addition of our SSE scheme and leaks no more information than Σ_{add} allows. For the deletion, it is conducted locally by the client, rather than by resorting to the server. Thus it is *oblivious* and leaks *nothing* to the server, and the client need not to interact frequently with the server, at the cost of only a small storage due to the *compressed* revocation property of our SRE scheme. Whenever the client performs search on w, it generates and sends to the server a revoked secret key for the deleted documents containing w. Then the server is able to decrypt all non-deleted documents with this key. Under this framework, our SSE scheme not only achieves Type-II backward privacy but also scales for large deletions.

We note that our scheme, same as previous works [9], [52], also requires the client to refresh the encryption key after each search, as the server can use the previous revoked secret key to decrypt the non-deleted document indices inserted in future. Nevertheless, we do not need to re-encrypt the search result with the refreshed key, which can be stored in a cache as processed in [9], [52]. As a result, we can obtain a large storage saving on server side by physically removing the retrieved ciphertexts from the server.

Next we briefly describe our scheme, the details of which are shown in Algorithm 1. Let $\Sigma_{\mathsf{add}} = (\Sigma_{\mathsf{add}}.\mathsf{Setup}, \Sigma_{\mathsf{add}}.\mathsf{Search}, \Sigma_{\mathsf{add}}.\mathsf{Update})$ be a forward private SSE scheme, and $\mathsf{SRE} = (\mathsf{SRE.KGen}, \mathsf{SRE.Enc}, \mathsf{SRE.KRev}, \mathsf{SRE.Dec})$ a CSRE scheme in which $\mathsf{SRE.KRev}$ consists of two sub-algorithms $\mathsf{SRE.Comp}$ and $\mathsf{SRE.cKRev}$ (cf. Compressed Revocation in Section III-C). Then our generic SSE scheme Σ based on Σ_{add} and SRE is comprised of $(\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$.

Algorithm 1 Type-II Backward-Private DSSE Σ from Compressed SRE

$\mathsf{Setup}(1^\lambda)$

1: $(\mathrm{EDB}_{\mathsf{add}}, K_{\mathsf{add}}, \sigma_{\mathsf{add}}) \leftarrow \Sigma_{\mathsf{add}}.\mathsf{Setup}(1^\lambda)$
2: $K_s, K_t \xleftarrow{\$} \{0,1\}^\lambda$, $\mathrm{EDB}_{cache} \leftarrow \emptyset$, **MSK**, **C**, **D** $\leftarrow \perp$
3: **return** $\big((K_{\mathsf{add}}, K_s, K_t), (\sigma_{\mathsf{add}}, \textbf{MSK}, \textbf{C}, \textbf{D}), (\mathrm{EDB}_{\mathsf{add}}, \mathrm{EDB}_{cache})\big)$

$\mathsf{Search}(K, w, \sigma; \mathrm{EDB})$

Client:

1: $i \leftarrow \textbf{C}[w]$, $(sk, D) \leftarrow \textbf{MSK}[w]$, $D \leftarrow \textbf{D}[w]$
2: **if** $i = \perp$ **then**
3: **return** $\emptyset$
4: **end if**
5: Compute $sk_R \leftarrow \mathsf{SRE.cKRev}(sk, D)$ ▷ D is from **D**[w]
6: Send (sk_R, D) and $tkn = F(K_s, w)$ to server
7: $msk = (sk, D) \leftarrow \mathsf{SRE.KGen}(1^\lambda)$ ▷ Update msk for w after search
8: $\textbf{MSK}[w] \leftarrow msk$, $\textbf{D}[w] \leftarrow D$, $\textbf{C}[w] \leftarrow i+1$

Client & Server:

9: Run $\Sigma_{\mathsf{add}}.\mathsf{Search}(K_{\mathsf{add}}, w||i, \sigma_{\mathsf{add}}; \mathrm{EDB}_{\mathsf{add}})$, and server gets a list $((ct_1, t_1), (ct_2, t_2), \ldots, (ct_\ell, t_\ell))$ of ciphertext and tag pairs

Server:

1: Server uses (sk_R, D) to decrypt all ciphertexts $\{(ct_i, t_i)\}$ as follows
2: **for** $i \in [1, \ell]$ **do**
3: $ind_i = \mathsf{SRE.Dec}((sk_R, D), ct_i, t_i)$
4: **if** $ind_i \neq \perp$ **then**
5: NewInd$\leftarrow$ NewInd $\cup \{(ind_i, t_i)\}$
6: **else**
7: DelInd$\leftarrow$ DelInd $\cup \{t_i\}$
8: **end if**
9: **end for**
10: OldInd$\leftarrow \mathrm{EDB}_{cache}[tkn]$
11: OldInd$\leftarrow$ OldInd$\backslash\{(ind, t) : \exists\ t_i \in$ DelInd *s.t.* $t = t_i\}$
12: Res$\leftarrow$ NewInd $\cup$ OldInd, $\mathrm{EDB}_{cache}[tkn] \leftarrow$ Res
13: **return** Res

$\mathsf{Update}(K, \mathsf{op}, (w, ind), \sigma; \mathrm{EDB})$

Client:

1: $msk \leftarrow \textbf{MSK}[w]$, $D \leftarrow \textbf{D}[w]$, $i \leftarrow \textbf{C}[w]$
2: **if** $msk = \perp$ **then**
3: $msk \leftarrow \mathsf{SRE.KGen}(1^\lambda)$, where $msk = (sk, D)$
4: $\textbf{MSK}[w] \leftarrow msk$, $\textbf{D}[w] \leftarrow D$
5: $i \leftarrow 0$, $\textbf{C}[w] \leftarrow i$
6: **end if**
7: Compute $t \leftarrow F_{K_t}(w, ind)$
8: **if** $\mathsf{op} = \mathsf{add}$ **then**
9: $ct \leftarrow \mathsf{SRE.Enc}(msk, ind, t)$
10: Run $\Sigma_{\mathsf{add}}.\mathsf{Update}(K_{\mathsf{add}}, \mathsf{add}, w||i, (ct, t), \sigma_{\mathsf{add}}; \mathrm{EDB}_{\mathsf{add}})$
11: **else** (*i.e.*, $\mathsf{op} = \mathsf{del}$)
12: $D \leftarrow \mathsf{SRE.Comp}(D, t)$, $\textbf{D}[w] \leftarrow D$
13: **end if**

$\mathsf{Setup}(1^\lambda)$: The client generates $(\mathrm{EDB}_{\mathsf{add}}, K_{\mathsf{add}}, \sigma_{\mathsf{add}}) \leftarrow \Sigma_{\mathsf{add}}.\mathsf{Setup}(1^\lambda)$, picks $K_s, K_t \xleftarrow{\$} \{0,1\}^\lambda$, and initializes lists **MSK**, **C**, **D**, and EDB_{cache} for storing encryption keys associated with each keyword, the search times of each keyword, the compressed deletion list *w.r.t.* each keyword, and the search results of previous queries, respectively. At the outset, **MSK**, **C** and **D** are filled with symbol $\perp$ and EDB_{cache} is set as empty. The algorithm then outputs $K = (K_{\mathsf{add}}, K_s, K_t)$, $\sigma = (\sigma_{\mathsf{add}}, \textbf{MSK}, \textbf{C}, \textbf{D})$ and $\mathrm{EDB} = (\mathrm{EDB}_{\mathsf{add}}, \mathrm{EDB}_{cache})$.

$\mathsf{Search}(K, w, \sigma; \mathrm{EDB})$: When performing search on w, the client obtains the search times i, the current secret key sk and the compressed deletion list D associated with w, by looking up $\textbf{C}[w]$, $\textbf{MSK}[w]$ and $\textbf{D}[w]$, respectively. Then it checks if $i = \perp$ (*i.e.,* $msk = \perp$), if so the client gets nothing. Otherwise, the client computes the revoked secret key $sk_R \leftarrow \mathsf{SRE.cKRev}(sk, D)$ and token $tkn = F(K_s, w)$ and sends $\big((sk_R, D), tkn\big)$ to the server. After this, the client refreshes msk[4], and runs $\Sigma_{\mathsf{add}}.\mathsf{Search}$ together with the server to retrieve the encrypted indices matching w. Then the server decrypts the non-deleted indices with sk_R, and returns them along with the non-deleted ones in cache as the search result.

$\mathsf{Update}(K, (\mathsf{op}, (w, ind)), \sigma; \mathrm{EDB})$: When adding a new entry (w, ind) to the database, the client obtains the most recent encryption key msk from $\textbf{MSK}[w]$ and inserts the encryption, $ct \leftarrow \mathsf{SRE.Enc}(msk, ind, t)$, of ind under tag $t = F_{K_t}(w, ind)$ to $\mathrm{EDB}_{\mathsf{add}}$. To delete the entry (w, ind), the client inserts the corresponding tag $t = F_{K_t}(w, ind)$ to the compressed deletion list D (*i.e.*, $D \leftarrow \mathsf{SRE.Comp}(D, t)$).

We remark that, for simplicity, it is implicitly assumed that at least one deletion happens for the queried keyword w. In fact, this assumption can be removed by slightly adapting the Search algorithm. Particularly, if no deletion occurs on w, we only need to perform a deletion on a *dummy* identity ind^* by running $\mathsf{Update}(K, \mathsf{del}, (w, ind^*), \sigma; \mathrm{EDB})$ prior to computing sk_R. Moreover, we note that our DSSE is constructed modularly from Σ_{add} and CSRE, so it would benefit immediately from any improvement on the building blocks.

[4]The encryption key msk must be updated after each search and will be used to encrypt future entries matching w, as noted by [9], [52].

B. Security Proof

Our scheme Σ can achieve Type-II backward privacy (cf. Definition 4), which in contrast to Type-III backward privacy does not leak which deletion operations remove which addition operations. As previous works [9], [15], [52], it also achieves *forward privacy*, which follows easily from that of Σ_{add}. In the following, we concentrate on Type-II backward privacy, formalized as Theorem 2.

Theorem 2. *The proposed scheme Σ is $\mathcal{L}_{BS}$-adaptively Type-II-backward-private, if Σ_{add} is an $\mathcal{L}_{FS}$-adaptively forward-private SSE scheme, CSRE is IND-sREV-CPA secure and F is a secure PRF, where $\mathcal{L}_{FS}$ is the leakage of Σ_{add} as defined in [7] and $\mathcal{L}_{BS} = (\mathcal{L}_{BS}^{Srch}, \mathcal{L}_{BS}^{Updt})$ is defined as $\mathcal{L}_{BS}^{Updt}(\mathsf{op}, w, ind) = \mathsf{op}$ and $\mathcal{L}_{BS}^{Srch} = (\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{DelTime}(w))$.*

Proof sketch. The proof is conducted in a similar way as Janus [9] and Janus++ [52], except that the security of Σ relies on that of the proposed SRE and the transcript is simulated with less leakage. A simplified proof is given in Appendix A-B. More details are shown in the full version.

V. Instantiation

In this section, we propose an instantiation of our generic

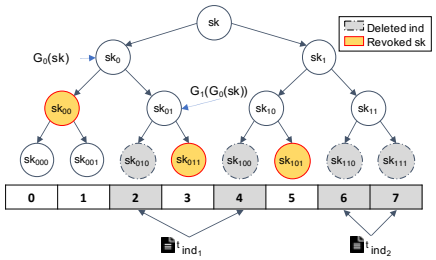


Fig. 1: Example of Compressed SRE

SSE scheme, termed as Aura. Notice that we can instantiate it by integrating a concrete CSRE scheme with any forward-private SSE scheme *e.g.,* in [7], [9]. Next we focus on how to instantiate the CSRE scheme by leveraging the GGM tree-based PRF and the standard Bloom filter.

In particular, we employ the GGM tree-based PRF to realize the multi-puncturable PRF and a standard Bloom filter to generate a compact revocation list. In our SSE scheme, each keyword w is associated with a GGM PRF key sk_w and an (initially empty) Bloom filter $B_w = 0^b$ for deletion on this keyword w between two consecutive search queries. As introduced in Section III-C, a Bloom filter is used to compress the deleted document tags and the GGM PRF key will be revoked based on the compact revocation list, so the number of leaf nodes in the GGM tree is set as the bit length of Bloom filter. In our GGM PRF, $G : \{0,1\}^\lambda \leftarrow \{0,1\}^{2\lambda}$ is a length-doubling pseudorandom generator, and the output of $G(sk)$ is divided into two halves $G_0(sk)$ and $G_1(sk)$. The value of GGM PRF F on ℓ-bit strings (*i.e.*, sk_i) is computed as $F_{sk}(i) = G_{i_{\ell-1}}(\cdots G_{i_1}(G_{i_0}(sk)))$, where the binary representation of the BF entry index i is $i_{\ell-1}\cdots i_1 i_0$. Here, $\ell = \lceil \log(b) \rceil$ and b is the number of BF entries.

When inserting a (w, ind) pair, we follow the encryption function of SRE. Specifically, the tag t of ind is firstly mapped to h entry indices, $\{i_j = H_j(t)\}_{j\in[h]}$, of the BF B_w. For each index $i_j \in [0, b-1]$, its corresponding leaf node $F_{sk_w}(i_j)$ is calculated from the master secret key sk_w of the GGM tree. Then the ind is encrypted with each leaf node as an encryption key, and the h encrypted copies will be uploaded to the server. After that, if the client starts to delete a (w, ind) pair, the tag t of ind is inserted to the BF B_w for later batch revocation. Namely, the h entries of B_w corresponding to t will be set to '1'. If more ind's on this w are deleted, the associated tags t's are continuously inserted to B_w. For a search query over w, the client generates the revoked secret key for the server, by puncturing the associated GGM PRF key sk_w on the indices of B_w entries with value '1', *i.e.*, $I_w = \{i' \in [b] : B_w[i'] = 1\}$. Specifically, for each revoked index in I_w, we find a path from the corresponding revoked leaf node to the root, and the revoked secret key sk_R consists of the siblings of the nodes on all the paths, but excluding those siblings that sit also on some other paths. As a result, sk_R is generated on the fly before search and the complexity of the communication cost is $\Theta(\log(b))$. Then sk_R and B_w are sent to the server with the search token together. To conduct the search, the server first uses the search token to fetch the matched encrypted entries. Meanwhile, it expands sk_R to get all leaf nodes of the GGM tree except the revoked ones. After that, the server checks the tag t of each fetched ciphertext with B_w to see if all h entries of B_w with indices $\{H_j(t)\}$ are marked as '1'. If so, it skips this entry, as it is deleted and cannot be decrypted. Otherwise, it follows the index of one entry with value '0' to find the corresponding GGM leaf node for decryption.

As an example in Figure 1, two tags for ind_1 and ind_2 are revoked, then sk_R consists of sk_{00}, sk_{011} and sk_{101}, which will be sent to the server for search. When receiving sk_R, the server can derive sk_{000} and sk_{001} from sk_{00} via $G_0(sk_{00})$ and $G_1(sk_{00})$, respectively. Then it can use sk_{000}, sk_{001}, sk_{011} and sk_{101} to decrypt the ind that is not revoked.

VI. Experimental Evaluation

Our evaluation reports the time cost of insertion, search, and deletion, as well as the communication cost, and compares the above metrics with the state-of-the-art interactive and non-interactive backward-private SSE schemes.

A. Implementation and Settings

We implement Aura in C++ and use OpenSSL to implement cryptographic primitives, *i.e.*, AES and SHA256. Pseudorandom generator in GGM PRF is implemented via AES. In order to evaluate the performance of Aura under the network environments, we leverage Thrift [49] to enable the network communication between Aura client and server. Our source code is publicly available in [1].

We conduct our evaluations under LAN and WAN environments. We run Aura client and server on a Ubuntu Server 18.04 LTS workstation (Intel Core i7-8850H 2.6GHz CPU with 6 cores and 32GB RAM) for LAN evaluations. The delay between the client and server is less than 0.1 ms, and the bandwidth in between is $1,000$Mbps. For evaluations on the WAN, we hire three e2-standard-8 instances (8 vCore, 32 GB RAM) with Ubuntu Server 18.04 LTS from Google Cloud. These instances are placed in Singapore, Sydney and South Carolina, respectively. We use the server in Sydney as the SSE server, and the one in Singapore (average round-trip delay to Sydney: 103 ms, bandwidth to Sydney: $9,700$Mbps) and South Carolina (average round-trip delay to Sydney: 197 ms, bandwidth to Sydney: $9,300$Mbps) as the SSE client. All the latency/bandwidth information is reported by Google [31].

We compare Aura with the interactive and non-interactive SSE schemes. Regarding the interactive ones, there exist several schemes [9], [15], [22] achieving Type-II backward privacy. In our evaluation, we compare with the most efficient scheme SD_d [22], that features small client storage at the expense of interactive update and search. Since SD_d is not implemented under the network environment, we adapt its original implementation to make it compatible with Thrift. Our implementation is available in [2]. For the non-interactive schemes, we choose Janus++ [52]. Note that it can only achieve Type-III backward privacy, while Aura achieves Type-II backward privacy. In the following evaluation, we evaluate the search cost required for the server to obtain the document ind's used for retrieving the real documents. We note that the server in Aura and Janus++ can get the ind's without extra communication costs after getting search tokens from the client, while the SD_d server requires an extra round to send the encrypted document ind's back to the client and then receive the decrypted ones from it.

For Janus++, we set the size of its tag space as 2^{16}. Namely, each tag is 16-bit long and the height of each

TABLE II: Bloom Filter Storage Cost

d	10		100		1,000		10,000	
	h = 5	h = 13	h = 5	h = 13	h = 5	h = 13	h = 5	h = 13
Storage (KiB)	0.036	0.024	0.35	0.23	3.54	2.34	35.37	23.4

GGM tree is 16. Since SD_d requires to specify the total number of updates during the setup phase, we set the number as '1,000,010', '1,000,100', '1,001,000' and '1,010,000' to instantiate SD_d in the evaluation.

To evaluate the scalability, we set d as '10', '100', '1,000', and '10,000', where d is the number of maximum deletions between two searches of a certain keyword, and set the number of deletions $d_w = d$. Then we derive two sets of BF parameters based on the tradeoff between client storage saving and time efficiency. For both parameter settings, we set the false positive rate $p = 10^{-4}$. We argue that this rate is acceptable in practice. If the number of matched documents in a query is not large, *e.g.*, $< 10,000$, all results can be returned in high confidence. For very large datasets, a small amount of false negatives are tolerable. For example, given a database with web pages or domain-specific documents , it may contain near-duplicate pages or documents. They may include almost the same contents but with different identifiers [53], [56]. Missing few results will not affect quality of service.

In the first parameter setting, the size of BF is derived from $b = -d \ln p/(\ln 2)^2$, and the optimal number of hash functions is equal to $h = \lceil b/d \ln 2 \rceil$, which is 13. Accordingly, the BF can bring 42.5% storage saving at the client compared to the naive approach where the client uses a 32-bit string to denote a deleted ind. This setting is suitable for clients with limited storage, *e.g.*, mobile devices. In the second setting, we envision that a relatively powerful client can allocate more storage for time efficiency, since the encryption time and search time scale with h. To still gain storage saving for the client, we obtain the minimum h, which is set as 5 and the saving is 12.5%. Detailed space consumption of BF is given in Table II.

B. Evaluation and Comparison with SD_d

We firstly review the SD_d protocol. The SD_d server requires to know the number of updates (U) during the setup phase, and it will initialise $\lceil \log U \rceil$ EDBs, where the i-th EDB keeps at most $2^i, i = 0, 1, ..., \lceil \log U \rceil - 1$ encrypted entries. The insertion and deletion of SD_d follows the same algorithm, which adds an encrypted entry with an operator indicating insert (INS) or delete (DEL) into the above EDBs. A new encrypted entry will be inserted in the first EDB (EDB_0) and gradually moved to the following EDBs ($\text{EDB}_{1,2,...,\lceil \log U \rceil - 1}$). During the search phase, the client queries all EDBs and retrieves all encrypted entries matching the query. Then, the client gets the final search result by decrypting the entries and removing all deleted ind's according to the associated operators. To retrieve the real documents, the client should send the non-deleted ind's back to the server.

Insertion time. To evaluate and compare the insertion time of Aura and SD_d, we update the EDB with '1,000,000' insertions and '10', '100', '1,000' and '10,000' deletions, respectively. Table III presents the unit time for insertion under both LAN and WAN environments. It shows that SD_d has a constant cost, i.e., 26ms under LAN. The cost consists of the time to generate an encrypted entry, insert a new entry and move the existing EDB entries to the next level EDBs if there exist full EDBs. The moving cost dominates the above insertion cost because it requires the client to retrieve the full EDB back and re-encrypt the EDB. To ensure forward privacy, the re-encrypted EDB is put into an oblivious map for the server to store it in the next level EDB. On the other hand, Aura only sends the new encrypted entries to the server during the insertion phase. It does not incur any costs of re-encryption and oblivious operations. Overall, Aura outperforms SD_d by $288\times$ ($h = 5$) and $123\times$ ($h = 13$) on the LAN environment.

TABLE III: Insertion Time (ms/ind) Comparison of Aura and SD_d when $d =$ '10', '100', '1,000' and '10,000'

Scheme	d = 10	d = 100	d = 1,000	d = 10,000
$\mathsf{Aura}_{h=5}$ (LAN)	0.05	0.06	0.07	0.09
$\mathsf{Aura}_{h=13}$ (LAN)	0.09	0.13	0.18	0.21
SD_d (LAN)	26.27	26.21	26.22	26.49
$\mathsf{Aura}_{h=5}$ (103ms)	50.97	50.12	50.39	50.78
$\mathsf{Aura}_{h=13}$ (103ms)	50.82	50.34	50.21	50.47
SD_d (103ms)	3317.22	3321.18	3336.67	3366.25
$\mathsf{Aura}_{h=5}$ (197ms)	99.53	99.42	99.47	99.89
$\mathsf{Aura}_{h=13}$ (197ms)	98.97	99.36	99.23	99.55
SD_d (197ms)	5721.82	5715.77	5731.74	5733.36

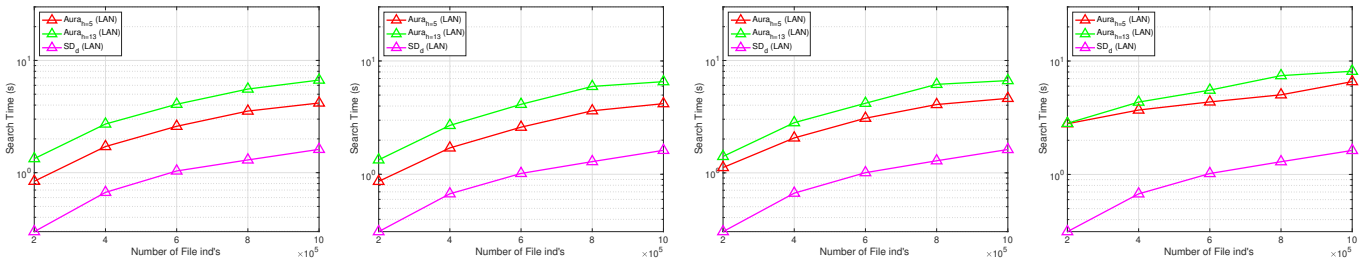
TABLE IV: Deletion Time Cost

Scheme	Aura		Janus++	SD_d (LAN)	SD_d (103ms)	SD_d (197ms)
	h=5	h=13				
Per deletion (μs)	0.0009	0.002	570	26,000	3,324,120	5,715,097

On the WAN environment, the network delay affects the insertion time significantly. In particular, SD_d can take more than $3,300$ms (103ms delay) or 5700ms to insert one ind. The reason is that SD_d has to move more entries among different EDBs to vacate the space for the new entry if the smaller EDBs have been filled. Thus, the moving operation performs more operations that need to communicate with the server, e.g., EDB status checking, ORAM access, re-encryption. When inserting $1,000,000$ ind's, the above process should be repeated 18 times at most before adding the new ind into the EDB. Due to the network delay, the cost per ind insertion of Aura also increases to 50ms (103ms delay) or 99ms (197ms delay), as it requires a one-way communication to send a fixed number of encrypted entries (the number of entries is fixed after setting h). Nonetheless, it is still $66\times$ to $219\times$ faster than SD_d.

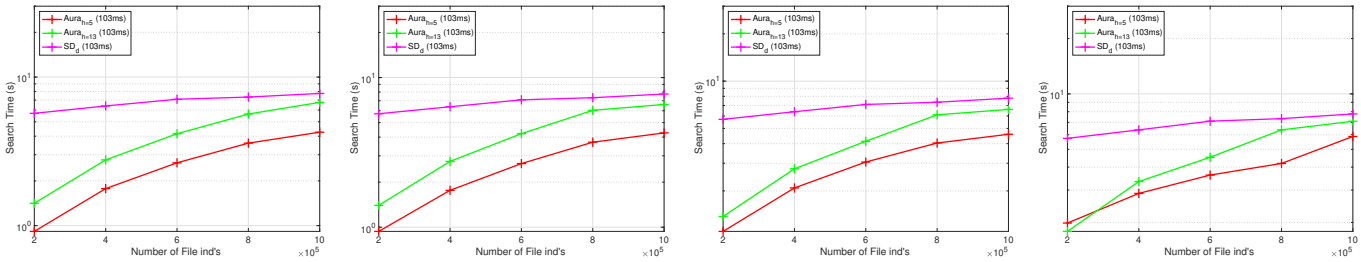
Deletion time. We run deletion operations 100 times for each scheme, and the average deletion time of each scheme is reported in Table IV. Note that the deletion in SD_d runs the same process as for insertion. Thus, the deletion time is similar to the insertion time, and it involves a non-negligible cost (26ms for LAN / $3,324$ms for 103ms-delay WAN / $5,715$ms for 197ms-delay WAN) to delete one entry. This is 7 to 9 orders of magnitude slower than Aura ($0.002 - 0.0009\mu s$), since Aura only needs to insert an associated tag of ind into a local Bloom filter when deleting an entry.

Search time. To demonstrate the benefit of using non-interactive schemes, we compare the search time of SD_d and Aura on LAN (search time with a negligible network delay) and WAN (with 103ms and 197ms-network delays) settings. Figure 2 illustrates that SD_d is $3\times$ to $4\times$ faster than Aura in terms of the computational cost under the LAN environment. However, Figures 3 and 4 show that SD_d



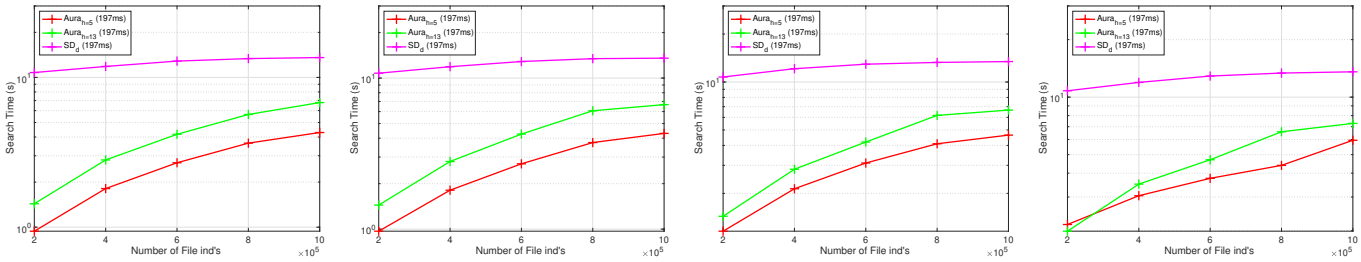
(a) Search time for $d = 10$ (b) Search time for $d = 100$ (c) Search time for $d = 1,000$ (d) Search time for $d = 10,000$

Fig. 2: Search Time Comparison of Aura and SD_d in LAN when $d =$'10', '100', '1,000' and '10,000'



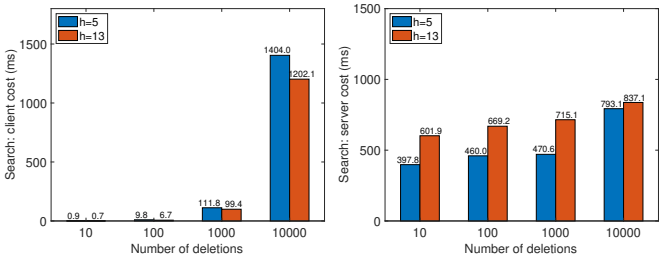
(a) Search time for $d = 10$ (b) Search time for $d = 100$ (c) Search time for $d = 1,000$ (d) Search time for $d = 10,000$

Fig. 3: Search Time Comparison of Aura and SD_d in WAN (103ms delay) when $d =$'10', '100', '1,000' and '10,000'



(a) Search time for $d = 10$ (b) Search time for $d = 100$ (c) Search time for $d = 1,000$ (d) Search time for $d = 10,000$

Fig. 4: Search Time Comparison of Aura and SD_d in WAN (197ms delay) when $d =$'10', '100', '1,000' and '10,000'



(a) Aura client search time cost (b) Aura server search time cost

Fig. 5: Search Time Comparison of Aura on Server and Client with different h ($h = 5$ and $h = 13$) when $d =$'10', '100', '1,000' and '10,000'

experiences a significant slowdown for the search on WAN: It becomes $1.03\times$ to $5\times$ slower than Aura when there is a 103ms-delay. Furthermore, Aura is $2\times$ to $11\times$ faster if the network delay increases to 197ms. This is because SD_d requires the client to query all EDBs (OLDEST_i, OLDER_i, OLD_i, $i = 0, 1, ..., \lceil \log U \rceil$-1) sequentially, thus resulting in multiple rounds of communication for each search, which makes it susceptible to the network performance. In our setting, we have $i = \lceil \log 200,010 \rceil - 1$ (17) to $\lceil \log 1,010,000 \rceil - 1$ (19), which indicates that there are 54 to 60 roundtrips for the query. On the other hand, Aura only incurs a one-way communication for the search, and the network delay only introduces 50-600ms extra time cost to transmit the revoked keys, Bloom filter and query token.

TABLE V: Communication Cost for Search

Scheme	Aura		Janus++	SD_d
	h=5	h=13		
d = 10	3KB	2KB	32B	8.58MB
d = 100	37KB	24KB	32B	8.58MB
d = 1,000	205KB	163KB	32B	8.58MB
d = 10,000	362KB	135KB	32B	8.58MB

Search communication cost. Table V shows that Aura only introduces a KB-level communication cost (2-362KB) even when $d = 10,000$. SD_d incurs an MB-level communication cost (8.58MB) during the search phase, which is $65\times$ ($h = 5$) and $24\times$ ($h = 13$) larger than Aura when $d = 10,000$. The reason is that SD_d must retrieve all encrypted entries matching the query keyword and remove deleted ones at the client. Further, the remaining ind's need to be sent back to the server for retrieving the real documents. The above process implies a roundtrip communication with a large amount of document ind's, e.g., in our setting, client receives '1,000,010',

TABLE VI: Communication Cost for Deletion

Scheme	Aura	Janus++	SD_d
d = 10	0	2KB	570B
d = 100	0	24KB	4KB
d = 1,000	0	240KB	42KB
d = 10,000	0	2400KB	429KB

TABLE VII: Insertion Time (ms/ind) of Janus++ when $d =$ '10', '100', '1,000' and '10,000'

Scheme	d = 10	d = 100	d = 1,000	d = 10,000
Janus++ (ms)	0.47	4.94	49.8	489.7

'1,000,100', '1,001,000' and '1,010,000' encrypted entries and sends '999,990', '999,900', '999,000' and '990,000' ind's back[5]. In contrast, Aura only sends the revoked secret key, Bloom filter and query token to the server (several KB), and the server can get the corresponding document ind's without additional communications.

Deletion communication cost. SD_d also incurs a communication cost during deletion as the client needs to retrieve and re-encrypt some entries. It also sends these re-encrypted and new entries to the server. As shown in Table VI, the cost is linear in the number of deletions. In particular, when deleting 10,000 entries, SD_d client sends 429KB to the server. In contrast, Aura does not incur any communication cost during deletion as it is a local process run by the client.

C. Evaluation and Comparison with Janus++

As mentioned in Section IV-A, Janus++ is constructed from SPE, where the encryption key of each ind is generated from d puncturable PRFs. Particularly, each puncturable PRF is realised by a GGM tree, so each encryption or decryption requires an evaluation of d GGM trees. Thus, it is not scalable for large number of deletions, as demonstrated later.

Insertion time. Table VII depicts the insertion time of Janus++, which scales linearly in the number d of deletions. When d is large, the performance of Janus++ degrades dramatically. In Aura, it takes constant time for insertion, *i.e.*, $0.06 - 0.09ms$ per ind for $h = 5$ and $0.13 - 0.21ms$ per ind for $h = 13$, as shown in Table III. The reason is that each encryption key derivation needs one single traversal from the root to a certain leaf node in the GGM tree, and our scheme involves only one tree. Note that the cost of $h = 13$ is larger than $h = 5$, because the number of ciphertext copies per ind is equal to h which contributes to h underlying key derivation and encryption operations.

Deletion time. We report the deletion time of each ind in Table IV. For each deletion, Aura only needs to insert the tag of the deleted ind to BF, and the cost is negligible to that of Janus++. In Janus++, each deletion revokes a certain leaf node in a GGM tree, which incurs a path traversal to generate the revoked secret key elements.

Search time. In Table VIII, we report the search time of Janus++. The results illustrate that Aura significantly outperforms Janus++ during the search phase. As mentioned, the main reason is that Aura only involves one GGM tree evaluation for decryption, while Janus++ needs to evaluate d GGM trees. Besides, Aura is implemented via C++ and OpenSSL, while Janus++ is implemented via Python and PyCrypto, which further contributes to the performance downgrade.

[5]The communication of search in Table V is a constant because the protocol transmits 2,000,000 ind's in total.

TABLE VIII: Search Time (s) of Janus++

# of ind's	$200,000$	$400,000$	$600,000$	$800,000$	$1,000,000$
d = 10	75.43	158.9	219.39	295.21	368.5
d = 100	736.85	1,538.57	2,269.03	2,987.85	3,731.71
d = 1,000	8,073.17	16,542.07	24,406.5	33,691.52	41,766.21
d = 1,0000	84,594.42	172,439.55	256,762.04	339,898.41	435,146.61

The search time of Aura consists of the client cost and server cost. Figure 5a and 5b shows the client and server cost of a search query with $1,000,000$ matched ind's under different d's. We observe that the client cost is affected by d; a larger d results in a larger GGM tree, and thus it requires more time to generate the revoked key. Besides, the client cost is affected by h; a larger h will reduce the size of BF and GGM tree (see Table II), which in turn decreases the cost of generating the revoked key. At the server side, the search time is dominated by recovering the ind's that are not deleted. This is correlated with h, as it requires to evaluate h hash functions of BF to decide whether or not each retrieved entry has been deleted.

In particular, Aura takes $0.004ms$ and $0.006ms$ to recover an ind for $h = 5$ and $h = 13$, respectively.

Search communication cost. The communication cost of Janus++ and Aura is presented in Table V. Note that the search cost of Aura relates to the size of the revoked key, which is linear in the number of GGM nodes that form the minimum cover of the unrevoked leaf nodes (*w.r.t.* the BF entries with value '0'). The results are derived from '10', '100', '1,000', '10,000' random deletions, respectively. In the worst case, half $b/2$ of BF entries will be revoked and $b/2$ GGM nodes will be sent to the server. As shown, Janus++ outperforms Aura as it only sends a token to the server during the search phase.

Deletion communication cost. Compared to Janus++, Aura saves the communication cost introduced by deletions significantly as shown in Table VI. We present the cost of the revocation key consisting of a set of GGM tree nodes in both schemes, because it dominates the communication cost during deletion. In Janus++, once a deletion happens, a part of the revoked key (a set of GGM tree nodes) will be sent to the server. And the communication cost increases linearly in the number of deletions. In Aura, the client does not communicate with the server during the deletion, and it just inserts the deleted ind into a Bloom filter at local.

Storage cost. The storage cost per ind in Aura is linear in h. Fortunately, this will not be the bottleneck under the framework of backward-private SSE [9], [52]. Like existing non-interactive backward-private SSE schemes Janus and Janus++, only the newly inserted ind's are encrypted via the dedicated schemes. After search, the results on this keyword can be cached at the server, and all the ciphertext copies can be physically deleted. Such treatment does not compromise the security of SSE, because the cached results are already known from the access pattern of previous searches.

VII. Conclusions

We first introduce a new cryptographic primitive, named Symmetric Revocable Encryption, and propose a generic con-

struction from the Bloom filter and multi-puncturable pseudorandom function. Then we present the first non-interactive Type-II backward-private SSE scheme (without hardware assumptions) from the newly introduced primitive. We also implement it in real networks and the evaluation demonstrates its practicality and scalability. Still, it is challenging to design efficient Type-I backward-private SSE within a single roundtrip, and very interesting to employ our revocation encryption technique to construct backward-private SSE in the multi-client setting. In addition, most of existing backward and forward-private SSE schemes support only restricted types of queries, so constructing practical SSE schemes with support for strong security as well as rich queries is still left open.

Acknowledgment. We would like to thank the anonymous reviewers for their constructive comments. This work is supported in part by the Natural Science Foundation of China (No. 61802255), the Australian Research Council (ARC) Discovery Projects (No. DP180102199, DP200103308), and the Key (Keygrant) Project of Chinese Ministry of Education (No. 2020KJ010201).

REFERENCES

[1] https://github.com/MonashCybersecurityLab/Aura.

[2] https://github.com/MonashCybersecurityLab/SDd.

[3] G. Amjad, S. Kamara, and T. Moataz, "Breach-resistant structured encryption," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 1, pp. 245–265, 2019.

[4] ——, "Forward and backward private searchable encryption with SGX," in *Proceedings of the 12th European Workshop on Systems Security, EuroSec@EuroSys 2019, Dresden, Germany, March 25, 2019*, 2019, pp. 4:1–4:6.

[5] L. Blackstone, S. Kamara, and T. Moataz, "Revisiting leakage abuse attacks," in *NDSS*. The Internet Society, 2020.

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[7] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *ACM CCS 2016, Vienna, Austria, October 24-28, 2016*, pp. 1143–1154.

[8] R. Bost and P. Fouque, "Thwarting leakage abuse attacks against searchable encryption - A formal approach and applications to database padding," *IACR Cryptology ePrint Archive*, p. 1060.

[9] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *ACM CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pp. 1465–1482.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure:{SGX} cache attacks are practical," in *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[11] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in *ACM CCS 2015, Denver, CO, USA, October 12-16, 2015*, 2015, pp. 668–679. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813700

[12] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[13] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *CRYPTO 2013, Santa Barbara, CA, USA, August 18-22, 2013*, 2013, pp. 353–373.

[14] D. Cash and S. Tessaro, "The locality of searchable symmetric encryption," in *EUROCRYPT 2014, Copenhagen, Denmark, May 11-15, 2014*, 2014, pp. 351–368.

[15] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *ACM CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pp. 1038–1055.

[16] Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *ACNS 2005, New York, NY, USA, June 7-10, 2005*, 2005, pp. 442–455.

[17] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *ASIACRYPT 2010, Singapore, December 5-9, 2010*, 2010, pp. 577–594.

[18] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution," *arXiv preprint arXiv:1802.09085*, 2018.

[19] D. Clayton, C. Patton, and T. Shrimpton, "Probabilistic data structures in adversarial environments," in *ACM CCS 2019, London, UK, November 11-15, 2019*, 2019, pp. 1317–1334.

[20] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[21] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *ACM CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pp. 79–88.

[22] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage," in *NDSS*, 2020.

[23] I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency," in *CRYPTO 2018, Santa Barbara, CA, USA, August 19-23, 2018*, 2018, pp. 371–406.

[24] D. Derler, T. Jager, D. Slamanig, and C. Striecks, "Bloom filter encryption and applications to efficient forward-secret 0-rtt key exchange," in *EUROCRYPT 2018, Tel Aviv, Israel, April 29 - May 3, 2018*, 2018, pp. 425–455.

[25] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans, "Efficient dynamic searchable encryption with forward privacy," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 1, pp. 5–20, 2018.

[26] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, "Rich queries on encrypted data: Beyond exact matches," in *ESORICS 2015, Vienna, Austria, September 21-25, 2015*, 2015, pp. 123–145.

[27] S. Feghhi and D. J. Leith, "A web traffic analysis attack using only timing information," *IEEE Trans. Information Forensics and Security*, no. 8, pp. 1747–1759.

[28] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *ACM STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, 2009, pp. 169–178.

[29] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions (extended abstract)," in *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, 1984, pp. 464–479. [Online]. Available: https://doi.org/10.1109/SFCS.1984.715949

[30] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[31] Google, "Google Cloud Inter-Region Latency and Throughput," https://datastudio.google.com/u/0/reporting/fc733b10-9744-4a72-a502-92290f608571/page/70YCB [online], 2020.

[32] M. D. Green and I. Miers, "Forward secure asynchronous messaging from puncturable encryption," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 305–320. [Online]. Available: https://doi.org/10.1109/SP.2015.26

[33] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 315–331.

[34] ——, "Learning to reconstruct: Statistical learning theory and encrypted database attacks," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 1067–1083.

[35] F. Hahn and F. Kerschbaum, "Searchable encryption with secure and efficient updates," in *ACM CCS 2014, Scottsdale, AZ, USA, November 3-7, 2014*, pp. 310–320.

[36] S. Hohenberger, V. Koppula, and B. Waters, "Adaptively secure puncturable pseudorandom functions in the standard model," in *ASIACRYPT 2015, Auckland, New Zealand, November 29 - December 3, 2015*, 2015, pp. 79–102.

[37] S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Outsourced symmetric private information retrieval," in *ACM CCS 2013, Berlin, Germany, November 4-8, 2013*, 2013, pp. 875–888.

[38] S. Kamara and T. Moataz, "Boolean searchable symmetric encryption with worst-case sub-linear complexity," in *EUROCRYPT 2017, Paris, France, April 30 - May 4, 2017*, 2017, pp. 94–124.

[39] ——, "SQL on structurally-encrypted databases," in *ASIACRYPT 2018, Brisbane, QLD, Australia, December 2-6, 2018*, 2018, pp. 149–180.

[40] ——, "Computationally volume-hiding structured encryption," in *EUROCRYPT 2019, Darmstadt, Germany, May 19-23, 2019*, 2019, pp. 183–213.

[41] S. Kamara, T. Moataz, and O. Ohrimenko, "Structured encryption and leakage suppression," in *CRYPTO 2018, Santa Barbara, CA, USA, August 19-23, 2018*, 2018, pp. 339–370.

[42] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Financial Cryptography and Data Security FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, 2013, pp. 258–274.

[43] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *ACM CCS 2012, Raleigh, NC, USA, October 16-18, 2012*, pp. 965–976.

[44] J. Katz, A. Sahai, and B. Waters, "Predicate encryption supporting disjunctions, polynomial equations, and inner products," in *EUROCRYPT 2008, Istanbul, Turkey, April 13-17, 2008*, 2008, pp. 146–162.

[45] ——, "Predicate encryption supporting disjunctions, polynomial equations, and inner products," *J. Cryptology*, vol. 26, no. 2, pp. 191–224, 2013.

[46] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *ACM CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pp. 1449–1463.

[47] A. B. Lewko, A. Sahai, and B. Waters, "Revocation systems with very small private keys," in *IEEE S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010, pp. 273–285.

[48] I. Miers and P. Mohassel, "IO-DSSE: scaling dynamic searchable encryption to millions of indexes by improving locality," in *NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[49] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable Cross-Language Services Implementation," *Facebook White Paper*, vol. 5, no. 8, 2007.

[50] D. X. Song, D. A. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pp. 44–55.

[51] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[52] S. Sun, X. Yuan, J. K. Liu, R. Steinfeld, A. Sakzad, V. Vo, and S. Nepal, "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *ACM CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pp. 763–780.

[53] M. Theobald, J. Siddharth, and A. Paepcke, "Spotsigs: robust and efficient near duplicate detection in large web collections," in *ACM SIGIR 2008, Singapore, July 20-24, 2008*, 2008, pp. 563–570.

[54] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *USENIX Security 2018*, 2018, pp. 991–1008.

[55] V. Vo, S. Lai, X. Yuan, S. Sun, S. Nepal, and J. K. Liu, "Accelerating forward and backward private searchable encryption using trusted execution," in *ACNS (2)*, ser. Lecture Notes in Computer Science, vol. 12147. Springer, 2020, pp. 83–103.

[56] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang, "Efficient similarity joins for near-duplicate detection," *ACM Trans. Database Syst.*, vol. 36, no. 3, pp. 15:1–15:41, 2011.

[57] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *USENIX Security 2016, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 707–720.

APPENDIX A
PROOFS

A. Proof of Lemma 2

Lemma 3. *Suppose* $\mathsf{SE} = (\mathsf{SE.Gen}, \mathsf{SE.Enc}, \mathsf{SE.Dec})$ *is an IND-CPA secure symmetric encryption scheme, then* G_3 *and* G_4 *are computationally indistinguishable for all PPT adversary* $\mathcal{A}$*. That is,* $|\Pr[\mathsf{Win}_4] - \Pr[\mathsf{Win}_3]| \leq 2h \cdot \mathsf{Adv}^{\text{IND-CPA}}_{\mathsf{SE},\mathcal{B}'}(\lambda)$*, where* $\mathcal{B}'$ *is a PPT adversary against the IND-CPA security.*

Proof of Lemma 2: We prove this lemma by the standard hybrid argument. To the end, we first define a sequence of games $\mathsf{G}_{3.0}, \mathsf{G}_{3.1}, \ldots, \mathsf{G}_{3.h}$. Particularly, $\mathsf{G}_{3.\ell}$ for $\ell \in [0, h]$ is identical to G_3 except that the first ℓ components of ct^* (*i.e.*, $(ct^*_1, ct^*_2, \ldots, ct^*_\ell)$) are the encryption of 0's and the others are the encryption of m_γ. Clearly, it holds that $\mathsf{G}_{3.0} = \mathsf{G}_3$ and $\mathsf{G}_{3.h} = \mathsf{G}_4$. Next we argue that each two successive games $\mathsf{G}_{3.\ell-1}$ and $\mathsf{G}_{3.\ell}$ are computational indistinguishable for $\ell \in [h]$. Note that the only difference between them lies in the ℓ-th component ct^*_ℓ of ct^*, which is the encryption of m_γ in $\mathsf{G}_{3.\ell-1}$ while the encryption of 0's in $\mathsf{G}_{3.\ell}$.

We assume for the sake of contradiction that there exists an adversary $\mathcal{A}$ that can distinguish $\mathsf{G}_{3.\ell-1}$ and $\mathsf{G}_{3.\ell}$ with non-negligible advantage. Then we construct an efficient algorithm $\mathcal{B}'$ to break the IND-CPA security of SE, as follows.

$\mathcal{B}'(1^\lambda)$ runs $\mathcal{A}$ to obtain t^*, and generates $msk = (sk, H, B) \leftarrow \mathsf{SRE.KGen}(1^\lambda, b, h)$, where $sk \leftarrow \mathsf{MF.Setup}(1^\lambda)$, $H = \{H_j\}_{j\in[h]}$ and $B = 0^b$. Then it computes the set of indices $I_{t^*} = \{i^*_j : i^*_j = H_j(t^*)\}_{j\in[h]}$ corresponding to t^*, indicating that for $i \in I_{t^*}$ there exists $j \in [h]$ *s.t.* $i = i^*_j$. $\mathcal{B}'$ then randomly chooses $u_1, \ldots, u_{\ell-1}, u_{\ell+1}, \ldots, u_k$ from $\mathcal{Y}$ and sets the PRF values $\{sk_i\}_{i\in[b]\setminus\{i^*_\ell\}}$ as follows:

$$sk_i = \begin{cases} F(sk, i), & i \in [b] \setminus I_{t^*} \\ u_j, & i \in I_{t^*} \setminus \{i^*_\ell\} \end{cases}$$

where $sk_{i^*_j} = u_j$ for $j \in [h] \setminus \{\ell\}$. Next, $\mathcal{B}'$ uses them along with sk to simulate all the following queries.

For an encryption query (m, t), $\mathcal{B}'$ computes $i_j = H_j(t)$ for each $j \in [h]$, then checks if $i_j = i^*_\ell$ and simulates the j-th component of ct according to the following cases:

- $i_j \neq i^*_\ell$: Computes $ct_j \leftarrow \mathsf{SE.Enc}(sk_{i_j}, m)$ with the value sk_{i_j}, which is set as above and known to $\mathcal{B}'$.
- $i_j = i^*_\ell$: Asks the SE challenger to create a corresponding ciphertext $ct_j \leftarrow \mathsf{SE.Enc}(sk_{i^*_\ell}, m)$, where $sk_{i^*_\ell}$ is chosen randomly by the challenger and unknown to $\mathcal{B}'$.

Finally, $\mathcal{B}'$ returns $ct = (ct_1, ct_2, \ldots, ct_h)$ to $\mathcal{A}$.

For a key-revocation query R, $\mathcal{B}'$ simulates the response sk_R in the same way as G_3.

In the challenge phase, $\mathcal{B}'$ receives two distinct messages m_0, m_1 from $\mathcal{A}$ and picks $\gamma \xleftarrow{\$} \{0, 1\}$. Then it sets $m'_0 = m_\gamma$ and $m'_1 = 0^{|m|}$, and generates the challenge ciphertext as:

- $j \leq \ell - 1$: Computes $ct_j^* \leftarrow \mathsf{SE.Enc}(sk_{i_j^*}, 0^{|m|})$. Recall that $sk_{i_j^*} = u_j$ for $j \neq \ell$.
- $j = \ell$: Submits m_0', m_1' to the SE challenger and asks for a challenge ciphertext $ct_j^* \leftarrow \mathsf{SE.Enc}(sk_{i_\ell^*}, m_\delta')$, where $sk_{i_\ell^*}$ and $\delta \in \{0,1\}$ are randomly chosen by the challenger.
- $j \geq \ell + 1$: Computes $ct_j^* \leftarrow \mathsf{SE.Enc}(sk_{i_j^*}, m_\gamma)$.

Finally $\mathcal{B}'$ outputs $ct^* = (ct_1^*, ct_2^*, \ldots, ct_h^*)$.

At last, $\mathcal{A}$ outputs her guess $\gamma' \in \{0,1\}$ when she halts, and $\mathcal{B}'$ outputs $\delta' = 1$ if $\gamma' = \gamma$.

We can see from the above that $\mathcal{B}'$ perfectly simulates $\mathsf{G}_{3.\ell-1}$ when $\delta = 0$ (*i.e.*, $m_0' = m_\gamma$) and $\mathsf{G}_{3.\ell}$ when $\delta = 1$ (*i.e.*, $m_1' = 0^{|m|}$). Then we have $\Pr[\delta' = 1 | \delta = 0] = \Pr[\mathsf{Win}_{3.\ell-1}]$ and $\Pr[\delta' = 1 | \delta = 1] = \Pr[\mathsf{Win}_{3.\ell}]$, so the advantage of $\mathcal{B}'$ is

$$\begin{aligned} \mathsf{Adv}^{\text{IND-CPA}}_{\mathsf{SE},\mathcal{B}'}(\lambda) &= \tfrac{1}{2} \left| \Pr[\delta' = 1 | \delta = 1] - \Pr[\delta' = 1 | \delta = 0] \right| \\ &= \tfrac{1}{2} \left| \Pr[\mathsf{Win}_{3.\ell}] - \Pr[\mathsf{Win}_{3.\ell-1}] \right|. \end{aligned}$$

Further, we get that

$$\begin{aligned} \left| \Pr[\mathsf{Win}_4] - \Pr[\mathsf{Win}_3] \right| &\leq \sum_{\ell=1}^{h} \left| \Pr[\mathsf{Win}_{3.\ell}] - \Pr[\mathsf{Win}_{3.\ell-1}] \right| \\ &\leq 2h \cdot \mathsf{Adv}^{\text{IND-CPA}}_{\mathsf{SE},\mathcal{B}'}(\lambda). \end{aligned}$$

This completes the proof of Lemma 2. ∎

B. Proof of Theorem 2

Theorem 3. *The proposed SSE scheme Σ is $\mathcal{L}_{BS}$-adaptively Type-II-backward-private, if Σ_{add} is an $\mathcal{L}_{FS}$-adaptively forward-private SSE scheme, SRE is an IND-sREV-CPA secure compressed SRE scheme and F is a secure PRF, where $\mathcal{L}_{FS}$ is the leakage of Σ_{add} as defined in [7] and $\mathcal{L}_{BS} = (\mathcal{L}_{BS}^{Srch}, \mathcal{L}_{BS}^{Updt})$ is defined as $\mathcal{L}_{BS}^{Updt}(\mathsf{op}, w, ind) = \mathsf{op}$ and $\mathcal{L}_{BS}^{Srch} = (\mathsf{sp}(w), \mathsf{TimeDB}(w), \mathsf{DelTime}(w))$.*

Proof of Theorem 2: The proof proceeds with a sequence of games. It starts with the real game and ends with a game that can be efficiently simulated with the leakage $\mathcal{L}_{BS}$.

Game G_0: This is the real SSE security game $\textsc{Real}^{\Sigma}_{\mathcal{A}}(\lambda)$, so we have that $\Pr[\textsc{Real}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] = \Pr[\mathsf{G}_0 = 1]$.

Game G_1: In this game, we modify the way of evaluating PRF F. Namely, each time a previously unseen keyword w (*resp.*, document/keyword pair (w, ind)) is used, we selects a random output from the range space of F, instead of computing $F(K_s, w)$ (*resp.*, $F_{K_t}(w, ind)$), and records it in table `Tokens` (*resp.*, `Tags`). Whenever F is recalled on the same input w (*resp.*, (w, ind)), the associated PRF value is retrieved straightforwardly from `Tokens` (*resp.*, `Tags`). We claim that this changes the distribution of the adversary's view only negligibly. This is because the replacement of $F(K_s, \cdot)$ (*resp.*, $F(K_t, \cdot)$) induces a distinguishing advantage equal to that of PRF against an adversary making at most N calls to F. Therefore, there exists an efficient reduction algorithm $\mathcal{B}_1$ such that $|\Pr[\mathsf{G}_1 = 1] - \Pr[\mathsf{G}_0 = 1]| \leq 2\mathsf{Adv}^{\text{PRF}}_{F,\mathcal{B}_1}(\lambda)$.

Game G_2: We modify this game by replacing real calls to the SSE scheme Σ_{add} by calls to the associated simulator $\mathcal{S}_{\mathsf{add}}$. To do so, we use some bookkeeping to keep track of all the Update queries as they come, rather than rely on the server to store them, and postpone all addition and deletion operations to the subsequent Search query. This can be done only because the updates leak nothing about their contents, which is guaranteed by the forward privacy of Σ_{add} and the obliviousness of deletions to server.

Moreover, a list $\mathtt{L}_{\mathsf{add}}$ is initialized and used in this game. In particular, the list $\mathtt{L}_{\mathsf{add}}$ contains the encryption of the inserted indices for the subsequent search on w, their associated tags and the insertion timestamps, which in fact corresponds to the update history on w for the scheme Σ_{add} and will be taken as the input of the simulator $\mathcal{S}_{\mathsf{add}}$. It can be seen that the distinguishing advantage between G_2 and G_1 can be reduced to the $\mathcal{L}_{FS}$-adaptive forward privacy of Σ_{add}. Thus, there exists a PPT adversary $\mathcal{B}_2$ such that $|\Pr[\mathsf{G}_2 = 1] - \Pr[\mathsf{G}_1 = 1]| \leq \mathsf{Adv}^{\mathcal{L}_{FS}}_{\Sigma_{\mathsf{add}},\mathcal{S}_{\mathsf{add}},\mathcal{B}_2}(\lambda)$.

Game G_3: This game is identical to G_2 except for generating the ciphertexts of indices of the deleted documents. Namely, when encrypting the document indices that were inserted previously and deleted later with the SRE scheme, we replace the indices by constant 0.

Since the modification above works only on the ciphertexts with revoked tags, we can reduce the distinguishing advantage between G_3 and G_2 to the IND-sREV-CPA security of the SRE scheme. Notice that, the selective security is sufficient for the application here, as the reduction algorithm can obtain from $\mathsf{UpHist}(w)$ the revoked tags before simulating the encryption of non-deleted indices and the revoked secret key. Thus, there exists a reduction algorithm $\mathcal{B}_3$ such that $|\Pr[\mathsf{G}_3 = 1] - \Pr[\mathsf{G}_2 = 1]| \leq \mathsf{Adv}^{\text{IND-sREV-CPA}}_{\mathsf{SRE},\mathcal{B}_3}(\lambda)$.

Game G_4: In this game, we modify the way of constructing list $\mathtt{L}_{\mathsf{add}}$ and the way of updating the compressed data structure D. Namely, we first compute the leakage information `TimeDB` and `DelTime` from the table `UpHist`, and then base this information to construct $\mathtt{L}_{\mathsf{add}}$ and update D. This has no influence to the distribution of G_3, so we have that $\Pr[\mathsf{G}_4 = 1] = \Pr[\mathsf{G}_3 = 1]$.

Game G_5: The tags in this game is generated in a different way. Namely, we generate the tags on the fly, instead of computing them from document/keyword pairs and storing them in the table `Tags`. We can do it like this because it is supposed that each document index was added/deleted at most once during the updates. In this way, tags will not repeat and we need not to store them for keeping consistence. Therefore, it holds that $\Pr[\mathsf{G}_5 = 1] = \Pr[\mathsf{G}_4 = 1]$.

Simulator. To build a simulator from G_5, what remains to do is to avoid explicitly using the keyword w to generate `Tokens`$[w]$. This can be done easily through replacing w by min $\mathsf{sp}(w)$. Moreover, the construction of $\mathtt{L}_{\mathsf{add}}$ and update of D can be properly simulated by taking the leakage $\mathsf{TimeDB}(w)$ and $\mathsf{DelTime}(w)$ as the input of Search, and there is no need for the simulator to keep track of the updates any more. At this point, we can see that G_5 can be efficiently simulated by the simulator with the leakage function $\mathcal{L}_{BS}$, so we get that $\Pr[\mathsf{G}_5 = 1] = \Pr[\textsc{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}_{BS}}(\lambda) = 1]$.

Finally, by combining all above we conclude that the advantage of any PPT adversary attacking our scheme Σ is

$$\begin{aligned} & \left| \Pr[\textsc{Real}^{\Sigma}_{\mathcal{A}}(\lambda) = 1] - \Pr[\textsc{Ideal}^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}_{BS}}(\lambda) = 1] \right| \\ \leq\ & 2\mathsf{Adv}^{\text{PRF}}_{\mathcal{B}_1,F}(\lambda) + \mathsf{Adv}^{\mathcal{L}_{FS}}_{\Sigma_{\mathsf{add}},\mathcal{S}_{\mathsf{add}},\mathcal{B}_2}(\lambda) + \mathsf{Adv}^{\text{IND-sREV-CPA}}_{\mathsf{SRE},\mathcal{B}_3}(\lambda). \end{aligned}$$

∎