Bestie: Very Practical Searchable Encryption with Forward and Backward Security

Tianyang Chen[1], Peng Xu[1(✉)], Wei Wang[2], Yubo Zheng[1], Willy Susilo[3], and Hai Jin[1]

[1] National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China
{chentianyang,xupeng,zhengyubo,hjin}@mail.hust.edu.cn

[2] Cyber-Physical-Social Systems Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
viviawangwei@hust.edu.cn

[3] Institute of Cybersecurity and Cryptology, School of Computing and Information Technology, University of Wollongong, Wollongong, Australia
wsusilo@uow.edu.au

Abstract. *Dynamic searchable symmetric-key encryption* (DSSE) is a promising crypto-tool that enables secure keyword searching over dynamically added or deleted ciphertexts. Currently, many works on DSSE devote their efforts to obtaining *forward and backward security* and practical performance. However, it is still challenging to design a single DSSE scheme that simultaneously achieves this security, high performance, and real deletion. Note that real deletion is a critical feature to guarantee *the right of the user to be forgotten* stipulated by GDPR. Due to this fact, we propose a new *forward-and-backward secure* DSSE scheme named `Bestie`. To achieve high search performance, `Bestie` takes the traditional hash and pseudorandom functions and symmetric-key encryption as building blocks and supports parallel keyword search. `Bestie` also achieves non-interactive real deletion for avoiding the client to do a *clean-up* process. This feature not only guarantees the above GDPR rule but also makes `Bestie` more suitable for managing large-scale data. `Bestie` also saves the client's computation and communication costs. Finally, we experimentally compare `Bestie` with five previous well-known works and show that `Bestie` is much better in most respects. For example, `Bestie` requires approximately 3.66 microseconds to find a matching ciphertext. In contrast, `Bestie` has search performance at least 2 times faster than both `Mitra`* (CCS'18) and `Diana`$_{del}$ (CCS'17), 1,032× faster than `Fides` (CCS'17), and 38,332× faster than `Janus++` (CCS'18), respectively. Compared with `Mitra` (CCS'18), `Bestie` saves at least 80% client time cost during a search.

E. Bertino et al. (Eds.): ESORICS 2021, LNCS 12973, pp. 3–23, 2021.
https://doi.org/10.1007/978-3-030-88428-4_1

Keywords: Dynamic searchable symmetric-key encryption · Forward and backward security · High performance · Real deletion

1 Introduction

Dynamic searchable symmetric-key encryption (DSSE) [18] originates from searchable symmetric-key encryption [25] (SSE). DSSE allows a client to delegate keyword searches over ciphertexts to an honest-but-curious server while preserving the keyword privacy as SSE does. Additionally, DSSE uniquely enables the client to dynamically update his ciphertexts on the server, such as adding a new ciphertext or deleting an existing ciphertext.

A DSSE scheme usually consists of two parties, namely, a client and an honest-but-curious server, and three protocols between these two parties, such as Setup, Update, and Search. In the beginning, the client runs the Setup protocol to generate his/her symmetric keys, and the server initializes an empty database. The Update protocol allows the client to add a new ciphertext to or delete an existing ciphertext from the server. Each ciphertext contains a keyword-and-file-identifier entry. The Search protocol enables the client to delegate a secure keyword search to the server. The server then returns all found ciphertexts (namely, the ciphertexts containing the expected keyword) to the client.

In terms of the DSSE security, the research community continues to attempt to reduce the information leakage caused by Update and Search protocols as much as possible. The possible information leakage includes keyword privacy and access and search patterns. Recently, researchers have devoted their efforts to developing DSSE schemes with *forward and backward security* [1,2]. Forward security can resist file-injection attack [31] by hiding the relationship between a newly issued Update query and any previously issued Search queries. Backward security can hide the historically deleted ciphertexts from the server when running a Search operation. Hence, *forward and backward security* is widely accepted and strong security in practice.

In terms of the DSSE performance, we say that a DSSE scheme is practical if it achieves high search performance, low client overhead, and non-interactive real deletion. The importance of the first two features is very evident. High search performance means that a DSSE scheme can search over large-scale encrypted data. Low client overhead alleviates the minimum limits on the capability of the client's device. The client, even with a mobile device, can also apply a DSSE scheme. Non-interactive real deletion allows a DSSE scheme to erase the deleted or invalid ciphertexts from the server without the client's help. This not only saves the storage cost of the server but also benefits a DSSE scheme to manage large-scale ciphertexts. Moreover, (non-interactive) real deletion satisfies the GDPR stipulation that *a data subject should have the right to have his or her personal data erased and no longer processed where the personal data are no longer necessary in relation to the purposes for which they are collected or otherwise processed* [23].

However, it is still challenging to design a single DSSE scheme with *forward and backward security* and practical performance. We extensively investigate

Table 1. SSE Scheme Comparison. Note that column "Search Cost" presents the average time cost to find one matching ciphertext if no Delete query was issued before. Symbol n_w denotes the number of ciphertexts matching a keyword search. Column "Encryption Cost" is the average time cost for the client to generate one ciphertext. Column "Real Deletion" describes the deletion type a DSSE scheme can achieve.

Scheme	BP	Search cost (μs)			Round trips	Encryption cost (μs)	Real deletion
		Client	Server	Total			
`Fides` [2]	II	4,142.63	24.18	4,166.81	2	4,145.22	Interactive
`Mitra` [4]	II	5.91	0.53	6.44	2	5.81	Failed
`Mitra`* [4]	II	12.06	3.92	15.98	2	7.38	Interactive
Diana$_{del}$ [2]	III	82.73/n_w	11.83	>11.83	2	13.21	Failed
`Janus++` [29]	III	480.58/n_w	154,569	>154,569	1	154,692	Non-interactive
`Bestie` (ours)	III	0.89	2.77	3.66	2	7.83	Non-interactive

previous well-known DSSE schemes with *forward and backward security*. Table 1 lists some of these schemes and shows that they are still not effective in terms of performance. Concretely, both `Fides` [2] and `Janus++` [29] suffer from low search performance and high client overhead during encryption. `Fides` also introduces high client time cost into a search process. `Fides` and Diana$_{del}$ both from [2] and `Mitra`*[1] [4] consume more bandwidth to transfer re-encrypted ciphertexts or to determine which ciphertexts should be deleted. Both Diana$_{del}$ and `Mitra` [4] fail to achieve real deletion. Both `Fides` and `Mitra`* achieve interactive real deletion. It means that both `Fides` and `Mitra`* require the client to re-encrypt and re-upload all matching and still-valid ciphertexts to the server. Clearly, achieving interactive real deletion is not practical for handling large-scale data. The above observations motivate us to develop a practical *forward-and-backward-secure* DSSE scheme that achieves high search performance, low client overhead, and non-interactive real deletion, simultaneously.

In this paper, we present `Bestie`, a very practical and *forward-and-Type-III-backward-secure* DSSE scheme. As shown in Table 1, `Bestie` achieves a Search protocol with the highest search performance and the non-interactive real deletion. Hence, `Bestie` is very suitable for managing large-scale data. In the Search protocol, `Bestie` only offers a slightly higher client time cost during a search than both Diana$_{del}$ and `Janus++`, since the `Bestie` client requires some time cost to decrypt the returned file identifiers from the server. However, the decryption cost of `Bestie` does not influence the practicality because modern CPUs are equipped with hardware instructions to accelerate the decryption process, such as AES-NI of Intel CPUs. In addition, although `Bestie` slightly increases the client's time cost, the total time cost of `Bestie` during a search is still the best one compared with all the above-mentioned schemes.

Some may worry that *forward and Type-III-backward security* is not strong enough. However, it is a fact that *forward and Type-III-backward security* is enough to mitigate the widely-concerned file-injection attack. Meanwhile, it

[1] `Mitra`* is a variant of `Mitra` that achieves interactive real deletion by the client to re-encrypt and re-upload the still-valid searchable ciphertexts.

seems impossible to achieve Type-I or Type-II backward security and practical performance, simultaneously. To achieve Type-I or Type-II backward security, we have to adopt the costly cryptographic primitive oblivious random access machine (ORAM) (or similar techniques, e.g., `Horus` [4] and `FB-DSSE` [32]), occupy client storage resources to stash deletion queries (e.g., `Aura` [28]), or simply omit (*non-interactive*) real deletion (e.g., `Mitra` and `Mitra`* both from [4], $\texttt{SD}_a$ and $\texttt{SD}_d$ both from [8], and `CLOSE-FB` [13]). Adopting ORAM (or similar techniques) means that a DSSE scheme must transfer largely redundant data or consume a considerable computation cost.

In summary, we outline our main contributions as below:

- We propose a new DSSE scheme, named `Bestie`, and prove that it is *forward-and-Type-III-backward secure*;
- We comprehensively compare `Bestie` with previous DSSE schemes and show that `Bestie` offers much better performance;
- We introduce the parallelization method to implement the Search protocol of `Bestie` and experimentally evaluate its efficiency.

We organize the remaining sections of this paper as follows. Section 2 reviews the background knowledge of DSSE. Section 3 presents our `Bestie`. Section 4 experimentally compares `Bestie` with previous works. We introduce the related works in Sect. 5. Section 6 finally concludes this paper.

2 Background

Notations. In the remaining sections, we use $\lambda \in \mathbb{N}$ to denote the security parameter. We use symbol $e \xleftarrow{\$} \mathcal{X}$ to denote randomly choosing an element e from space or set $\mathcal{X}$. Symbol $\{0,1\}^k$ indicates all strings that are of binary length $k \in \mathbb{N}$, and symbol $\{0,1\}^*$ represents all strings of arbitrary binary length. Let 0^λ be the string of binary length λ whose bits are all zero. We assume that no file has the identifier 0^λ. Let symbol $\text{poly}(\lambda)$ denote a polynomial with input parameter λ. Let symbol $\perp$ denote the abort operation.

Definition 1 (DSSE). *A DSSE scheme Σ consists of three protocols between the client and the server, such as Σ.Setup, Σ.Update, and Σ.Search. Their definitions are given below:*

- *Protocol Σ.Setup(λ): In this protocol, the client initializes his secret key K_Σ and an empty state-set σ for the given security parameter λ and sends an empty encrypted database **EDB** to the server. The client keeps both his key K_Σ and state-set σ private.*
- *Protocol Σ.Update($K_\Sigma, \sigma, op, (w, id)$; **EDB**): In this protocol, the client either adds a new keyword-and-file-identifier entry (w, id) to or deletes an existing entry from the server according to parameter $op \in \{add, del\}$. Given key K_Σ and state-set σ, the client sends a new ciphertext of entry (w, id) to the server if $op = add$; otherwise (namely, $op = del$), he sends a delete token of entry (w, id) to the server. When receiving the above message, the server updates its database **EDB** correspondingly.*

– *Protocol* $\Sigma.\mathsf{Search}(K_\Sigma, \sigma, w; \boldsymbol{EDB})$*: In this protocol, given key* K_Σ *and state-set* σ*, the client sends a search trapdoor of the expected keyword* w *to the server. Then, the server searches the keyword over the database* $\boldsymbol{EDB}$ *and returns all valid file identifiers to the client.*

A DSSE scheme must be correct in the sense that all valid file identifiers can always be found. The formal definition can be found in [18].

A popular method to define the adaptive security of DSSE is to define the indistinguishability between a real game and an ideal game of DSSE. In both games, the adversary can adaptively issue Update and Search queries. In the real game, all keyword-and-file-identifier entries and secret keys are real, and both protocols Σ.Update and Σ.Search are correctly implemented. In contrast, in the ideal game, a simulator only uses leakage functions to simulate the responses to all queries of the adversary. We say that DSSE is adaptively secure if a simulator can simulate an ideal game that is indistinguishable from the real game. The fewer leakage functions the simulator uses, the more secure DSSE is.

Definition 2 (Adaptive Security of DSSE). *Given leakage functions* $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Updt}, \mathcal{L}^{Srch})$*, a DSSE scheme* Σ *is said to be* $\mathcal{L}$*-adaptively secure if for any sufficiently large security parameter* $\lambda \in \mathbb{N}$ *and adversary* $\mathcal{A}$*, there exists an efficient simulator* $\mathcal{S} = (\mathcal{S}.\mathsf{Setup}, \mathcal{S}.\mathsf{Update}, \mathcal{S}.\mathsf{Search})$ *for which* $|Pr[Real^{\Sigma}_{\mathcal{A}}(\lambda) = 1] - Pr[Ideal^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda) = 1]|$ *is negligible in* λ*, where games* $Real^{\Sigma}_{\mathcal{A}}(\lambda)$ *and* $Ideal^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$ *are defined as below:*

– $Real^{\Sigma}_{\mathcal{A}}(\lambda)$*: The real game exactly implements all DSSE protocols. Adversary* $\mathcal{A}$ *can adaptively issue* Update *and* Search *queries with input* $(op, (w, id))$ *and* w*, respectively, and observe the real transcripts generated by the DSSE protocols. At the end, adversary* $\mathcal{A}$ *outputs a bit.*
– $Ideal^{\Sigma}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda)$*: All transcripts are simulated by the simulator* $\mathcal{S}$*. Adversary* $\mathcal{A}$ *can issue the same queries as in the real game. The simulator* $\mathcal{S}$ *simulates the corresponding transcripts by taking leakage functions* $\mathcal{L}$ *as input. At the end, adversary* $\mathcal{A}$ *outputs a bit.*

Let list Q be a set of all Update and Search queries, where each entry in list Q has the form of either $(u, op, (w, id))$ or (u, w) for Update and Search queries, respectively, where parameter u denotes the timestamp of issuing a query. Given a keyword w, let function $\mathrm{sp}(w)$ return all timestamps of the Search queries about keyword w, function $\mathrm{TimeDB}(w)$ return all undeleted file identifiers of keyword w and the history timestamps for adding these files, and function $\mathrm{DelHist}(w)$ return the history timestamps of all paired Add and Delete operations about keyword w. The formal definitions of the above three functions are given below.

$$\mathrm{sp}(w) = \{u | (u, , w) \in Q\}$$

$$\mathrm{TimeDB}(w) = \{(u, id) | (u, add, (w, id)) \in Q \text{ and } \forall u', (u', del, (w, id)) \notin Q\}$$

$$\mathrm{DelHist}(w) = \{(u^{add}, u^{del}) | \exists id, (u^{add}, add, (w, id)) \in Q \text{ and } (u^{del}, del, (w, id)) \in Q\}$$

With the above functions, *forward and Type-III-backward security* is defined as follows.

Definition 3 (Forward and Type-III-Backward Security [1,2]). *An $\mathcal{L}$-adaptively secure DSSE scheme Σ is* forward-and-Type-III-backward secure *iff the* Update *and* Search *leakage functions $\mathcal{L}^{Updt}$ and $\mathcal{L}^{Srch}$ can be written as*

$$\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'(op) \text{ and } \mathcal{L}^{Srch}(w) = \mathcal{L}''(sp(w), TimeDB(w), DelHist(w))$$

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

Besides the Type-III backward security, there exist two other levels of backward security, namely Type-I backward security and Type-II backward security. Type-I backward security requires that a `Search` query leaks only TimeDB(w) and the total number of updating w. In contrast, Type-II backward security additionally allows a `Search` query to leak the timestamps of updating w. This paper focuses on the Type-III backward security. The formal definitions of Type-I and Type-II backward securities can be found in [2].

3 Construction of `Bestie`

As shown in Table 1, our proposed `Bestie` is practical in almost all aspects. To achieve this, the construction of `Bestie` adopts the following three core ideas:

- To achieve high search performance, `Bestie` generates searchable ciphertexts in a counter-based design and collects all found ciphertexts into a group for each Search query. This method avoids redundant computation in the next-time search on these ciphertexts. The counter-based design enables the server to find all matching ciphertexts by traversing all valid counter values, and the resulting search complexity is sub-linear with respect to the total number of ciphertexts. Additionally, `Bestie` avoids any expensive cryptographic operation and adopts some hash computations. Hence, `Bestie` achieves faster search performance than the previous works, such as `Fides` and `Janus++`.
- To realize non-interactive real deletion while ensuring minimal information leakage, we apply the idea of combining logical deletion and real deletion introduced by Xu *et al.* [30]. `Bestie` additionally achieves higher security (less information leakage) than the scheme in [30]. In short, when issuing an Update query to delete a historical ciphertext, `Bestie` uploads a new ciphertext to the server. This ciphertext contains the index of a historical ciphertext that the client expects to delete. When a Search query finds this new ciphertext, the server can decrypt the contained index, and this index can guide the server to really delete the corresponding historical ciphertext.
- To achieve forward security, `Bestie` chooses a new key for a keyword to generate the keyword ciphertexts after each search for the keyword. In other words, after each search for a keyword, `Bestie` chooses a new key to generate the following ciphertexts of the keyword. To avoid storing many keys for a keyword on the client-side, `Bestie` allows the client to maintain a Search

counter for each keyword and dynamically generate any historical key of a keyword. Moreover, the use of a group on the server-side enables Bestie to send a constant-size search trapdoor for each Search query, instead of delegating many search trapdoors generated by the different historical keys of a keyword to the server. To achieve backward security, Bestie encrypts all file identifiers such that the server learns nothing about the deleted file identifier.

3.1 Our Construction

Let λ be the security parameter. Bestie utilizes a PRF function $\mathbf{F} : \mathcal{K}_{\mathbf{F}} \times \mathcal{X}_{\mathbf{F}} \rightarrow \mathcal{Y}_{\mathbf{F}}$, where $\mathcal{K}_{\mathrm{F}} = \{0,1\}^{\lambda}$ is the secret key space, $\mathcal{X}_{\mathbf{F}} = \{0,1\}^{*}$ is the domain and $\mathcal{Y}_{\mathbf{F}} = \{0,1\}^{\lambda}$ is the range. The security of PRF ensures that the output of $\mathbf{F}$ is indistinguishable from a randomly sampled value from $\mathcal{Y}_{\mathbf{F}}$, except for a negligible probability in the parameter λ. Bestie also uses a semantically secure and probabilistic symmetric encryption scheme ξ. ξ comprises of a probabilistic encryption algorithm $\mathcal{E} : \mathcal{K}_{\xi} \times \mathcal{M}_{\xi} \rightarrow \mathcal{C}_{\xi}$ and the corresponding decryption algorithm $\mathcal{D} : \mathcal{K}_{\xi} \times \mathcal{C}_{\xi} \rightarrow \mathcal{M}_{\xi} \cup \{\perp\}$, where $\mathcal{K}_{\xi} = \{0,1\}^{\lambda}$ is the symmetric key space, $\mathcal{M}_{\xi} = \{0,1\}^{*}$ is the plaintext space, and $\mathcal{C}_{\xi} = \{0,1\}^{*}$ is the ciphertext space. Algorithm 1 presents the pseudocode of protocols Bestie.Setup, Bestie.Update, and Bestie.Search. The following content explains the details of Bestie.

Bestie.**Setup Protocol.** Upon inputting a security parameter λ, this protocol mainly initializes two keys K_{ξ} and S and some empty storage structures **Count** and $\mathbf{EDB} = (\mathbf{CDB}, \mathbf{GRP})$. Key K_{ξ} will be used as a symmetric key to encrypt (or decrypt, resp.) file identifiers when updating an entry (or receiving the search results from the server, resp.). Key S is a master key. When issuing an Update query or a Search query of a keyword, the client will dynamically generate two sub-keys from the master key for the keyword to complete the query. Such a method avoids storing many keys on the client-side. Structure **Count** stores the update times of a keyword after the last search of the keyword and the keyword search times. The client must locally store structure **Count** and ensure its privacy. The server uses structure **CDB** to store all newly updated ciphertexts that are never searched and uses group **GRP** to group the still-valid ciphertexts of the same keyword when receiving a Search query.

Bestie.**Update Protocol.** This protocol enables the client to issue an Update query for adding or deleting a keyword-and-file-identifier entry, such as (w, id). For a given entry (w, id), the client locally queries structure **Count** to obtain the existing update times c_w^{updt} after the last search of keyword w and the existing search times c_w^{srch} of keyword w (refer to Steps 1 to 3). Then, the client computes two special keys K_w and K'_w with the master key S and the search times c_w^{srch} of keyword w (refer to Step 4). As a result, the client will use a new key K_w to generate the subsequent ciphertexts of keyword w after each search of this keyword. Each ciphertext consists of three parts, namely, (L, D, C). Part L is taken as a unique address to store the ciphertext in structure **CDB**. Part D encrypts the inputted operation type op and a hash value $\mathbf{G}(K'_w, id)$. When searching

Algorithm 1. Protocols Bestie.Setup, Bestie.Update and Bestie.Search.

Setup(λ)

1: Choose two hash functions $\mathbf{H} : \{0,1\}^* \rightarrow \{0,1\}^{\lambda'+\lambda+1}$, where $\lambda' = \text{poly}(\lambda)$ and $\mathbf{G} : \{0,1\}^* \rightarrow \{0,1\}^{\lambda}$
2: Initialize three empty maps **Count**, **CDB**, and **GRP**, and let $\mathbf{EDB} = (\mathbf{CDB}, \mathbf{GRP})$
3: Let $op \in \{add, del\}$ with the binary codes $add = 1$ and $del = 0$
4: Pick a symmetric key $K_\xi \xleftarrow{\$} \mathcal{K}_\xi$ and a secret key $S \xleftarrow{\$} \mathcal{K}_{\mathbf{F}}$, and set $K_\Sigma = (S, K_\xi)$
5: Store **EDB** on the server

Update(K_Σ, **Count**, op, (w, id); **EDB**)

Client:

1: Retrieve the current count values $(c_w^{updt}, c_w^{srch}) \leftarrow \mathbf{Count}[w]$ with the inputted keyword w
2: Initialize $(c_w^{updt}, c_w^{srch}) \leftarrow (0,0)$ if $\mathbf{Count}[w]$ is NULL
3: Accumulate the update times of keyword w by setting $c_w^{updt} \leftarrow c_w^{updt}+1$ and update such modification to $\mathbf{Count}[w]$
4: Compute two special keys $K_w \leftarrow \mathbf{F}(S, w||c_w^{srch})$ and $K'_w \leftarrow \mathbf{F}(S, w|| - 1)$ for encrypting keyword w
5: Encrypt keyword w as $(L||D) \leftarrow \mathbf{H}(K_w, c_w^{updt}) \oplus (0^{\lambda'}||op||\mathbf{G}(K'_w, id))$, where L represents the most significant λ' bits of the result and D denotes the remaining $\lambda+1$ bits (Note that we have assumed that id cannot be 0^λ in Section 2)
6: Encrypt the inputted file identifier as $C \leftarrow \mathcal{E}(K_\xi, id)$
7: Send ciphertext (L, D, C) to the server

Server:

1: Store the received ciphertext by setting $\mathbf{CDB}[L] \leftarrow (D, C)$

Search(K_Σ, **Count**, w; **EDB**)

Client:

1: Retrieve the current count values $(c_w^{updt}, c_w^{srch}) \leftarrow \mathbf{Count}[w]$
2: Abort if $\mathbf{Count}[w]$ is NULL
3: Compute two special keys $K_w \leftarrow \mathbf{F}(S, w||c_w^{srch})$ and $K'_w \leftarrow \mathbf{F}(S, w|| - 1)$ for generating the search trapdoor of keyword w
4: Compute index $I_w^{grp} \leftarrow \mathbf{G}(K'_w, 0^\lambda)$ for the server to group the search results of this time with the results of the previous searches of the same keyword
5: Send search trapdoor $(c_w^{updt}, K_w, I_w^{grp})$ to the server

Server:

1: Initialize an empty set $\mathcal{D}$, and set $i \leftarrow c_w^{updt}$
2: **repeat**
3: Break if $i = 0$
4: Compute $(L||D') \leftarrow \mathbf{H}(K_w, i)$, where L represents the most significant λ' bits of the hash value and D' denotes the remaining $\lambda+1$ bits
5: Retrieve ciphertext $(D, C) \leftarrow \mathbf{CDB}[L]$ according to index L
6: Decrypt $(op||X) \leftarrow D \oplus D'$
7: **if** $op = del$ **then**
8: Record X into $\mathcal{D} \leftarrow \mathcal{D} \cup \{X\}$
9: Remove all old ciphertexts $(X', C') \in \mathbf{GRP}[I_w^{grp}]$ with $X' = X$ that already exist in $\mathbf{GRP}[I_w^{grp}]$ before this search
10: **else if** $op = add$ **then**
11: Record (X, C) into group $\mathbf{GRP}[I_w^{grp}]$ if $X \notin \mathcal{D}$
12: **end if**
13: Set $i = i - 1$
14: **until** $(i = 0)$
15: Remove all above ciphertexts found from structure **CDB**
16: Return all remaining file identifier ciphertexts in group $\mathbf{GRP}[I_w^{grp}]$ to the client

Client:

1: Use the symmetric key K_ξ to decrypt all received ciphertexts and return the obtained file identifiers
2: Accumulate the search times of keyword w by setting $c_w^{srch} \leftarrow c_w^{srch} + 1$ and updating $\mathbf{Count}[w]$
3: Update $\mathbf{Count}[w]$ by setting $c_w^{updt} = 0$

keyword w in the future, this hash value will inform the server which ciphertext must be deleted if $op = del$. Part C encrypts the inputted file identifier.

Bestie.**Search Protocol.** This protocol allows the client and the server to complete a search task together. The protocol contains three steps: the client generates and uploads a search trapdoor to the server, the server then finds and returns the matching ciphertexts, and finally, the client receives and decrypts the returned file identifiers. For searching a keyword w, the generated search trapdoor consists of three parts: the current counter value c_w^{updt}, the sub-key K_w, and a group index I_w^{grp}. The value c_w^{updt} denotes the number of newly updated ciphertexts of keyword w after the last search of keyword w. Upon receiving both c_w^{updt} and K_w, the server can find the matching ciphertexts from these newly updated

ones. The group index I_w^{grp} allows the server to retrieve the historical matching ciphertexts from group **GRP**. The final search results include these two parts.

When searching a keyword w, first, the server finds all new matching ciphertexts from structure **CDB** (refer to Steps 2 to 14). The server handles these matching ciphertexts as two cases according to their contained operation types (refer to Steps 7 to 12). In the case of $op = del$, the decrypted value X means that the server will remove a ciphertext from either structure **CDB** or group $\mathbf{GRP}[I_w^{grp}]$ if the ciphertext contains the same value X. Note that when using the value X to delete ciphertexts from group $\mathbf{GRP}[I_w^{grp}]$, the server only removes the old ciphertexts found in previous search queries. This guarantees that a Delete query only removes the early corresponding Add queries, not the subsequent Add queries. In the case of $op = add$, the server will add the corresponding ciphertext into group $\mathbf{GRP}[I_w^{grp}]$ if the ciphertext is still valid (not deleted). Second, the server retrieves all matching and still-valid ciphertexts from group $\mathbf{GRP}[I_w^{grp}]$ and returns them to the client. Finally, the client decrypts the matching file identifiers, accumulates the search times c_w^{srch}, and clears the counter value c_w^{updt}.

Correctness of `Bestie`. The correctness of `Bestie` comes straightforwardly from the collision-resistant property of hash functions **H** and **G**. Concretely, when searching a keyword w, the server repeats computing hash value $\mathbf{H}(K_w, i)$ for i decreasing successively from c_w^{updt} to 1 and obtains the number of c_w^{updt} distinct addresses. These addresses guarantee the search correctness of finding all matching ciphertexts from structure **CDB**. The uniqueness of hash value $I_w^{grp} = \mathbf{G}(K'_w, 0^\lambda)$ guarantees that group $\mathbf{GRP}[I_w^{grp}]$ contains all historical matching ciphertexts. In addition, the uniqueness of hash value $\mathbf{G}(K'_w, id)$ guarantees that the server can correctly delete the expected ciphertext.

Security of `Bestie`. In terms of security, `Bestie` is *forward-and-Type-III-backward secure*. Formally, we have the following theorem, which is proven in Appendix A.

Theorem 1. *Suppose that the hash function* $\boldsymbol{H}$ *is a random oracle,* $\boldsymbol{G}$ *is a cryptographic hash function, function* $\boldsymbol{F}$ *is a secure PRF function, and* ξ *is a CPA-secure symmetric encryption scheme; then,* `Bestie` *is an adaptively secure DSSE scheme with leakage functions* $\mathcal{L}^{Stp}(\lambda) = \lambda$, $\mathcal{L}^{Updt}(op, w, id) = \emptyset$, *and* $\mathcal{L}^{Srch}(w) = \{sp(w), TimeDB(w), DelHist(w)\}$.

Remarks on Improving the Performance. To reduce the hash function iterations as much as possible, `Bestie` computes the partial ciphertexts L and D by only one hash computation (refer to Step 5 in protocol Update), instead of independently generating them by running a hash function twice. This approach is very effective in reducing the time cost. For example, our experiment shows that the time cost to run hash function SHA-256 twice by the OpenSSL library is approximately 20% more than running hash function SHA-512 once.

When searching for a keyword, the server can find the matching ciphertexts by a parallel method to improve the performance. Refer to Steps 2 to 14 in protocol Search. The repeated computations to find all matching ciphertexts from structure **CDB** can be transformed into a parallel algorithm since each

computation relies on the sub-key K_w and the parameter $i \in [1, c_w^{updt}]$. The server can traverse all possible values of parameter i in parallel to improve the search performance. We will show a parallel execution of protocol Search to demonstrate its significant advantage in Sect. 4.

3.2 An Example of Bestie

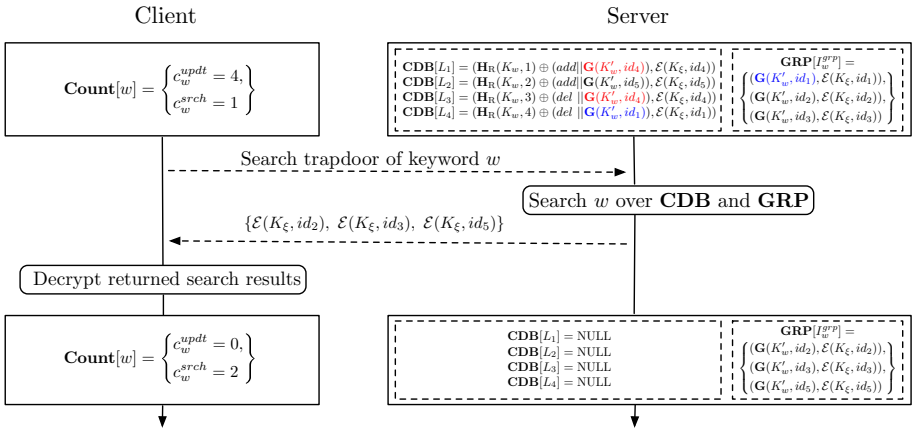


Fig. 1. Example of Bestie. Note that $\mathbf{H}_R$ denotes the least significant $\lambda + 1$ bits of the output of hash function $\mathbf{H}$. The ciphertexts with $op = del$ colored with red or blue mean to delete the corresponding ciphertexts with $op = add$ of the same color, respectively.

We introduce an example to clearly present the search procedures of Bestie. For keyword w, suppose that in the first time search of this keyword, the server finds three matching ciphertexts from structure **CDB** and transfers them to group $\mathbf{GRP}[I_w^{grp}]$. Then, the client issues four Update queries for adding entries (w, id_4) and (w, id_5) and deleting entries (w, id_4) and (w, id_1). The two upper-side rectangles of Fig. 1 shows the states of both the client and the server after completing the above operations, respectively. The client records the current counter values. The server stores the four newly uploaded ciphertexts in structure **CDB**.

Now, suppose the client to issue the search query of keyword w again. Upon receiving the corresponding search trapdoor, the server first finds the four matching ciphertexts $\mathbf{CDB}[L_4]$, $\mathbf{CDB}[L_3]$, $\mathbf{CDB}[L_2]$, and $\mathbf{CDB}[L_1]$ and decrypts $del||\mathbf{G}(K'_w, id_1)$, $del||\mathbf{G}(K'_w, id_4)$, $add||\mathbf{G}(K'_w, id_5)$, and $add||\mathbf{G}(K'_w, id_4)$, sequentially. Secondly, the server performs the following steps: (1) use $del||\mathbf{G}(K'_w, id_1)$ to delete entry $(\mathbf{G}(K'_w, id_1), \mathcal{E}(K_\xi, id_1))$ from group $\mathbf{GRP}[I_w^{grp}]$ and temporarily store $\mathbf{G}(K'_w, id_1)$ for future (possible) deletions, (2) temporarily store $\mathbf{G}(K'_w, id_4)$ for future (possible) deletions, (3) store entry $(\mathbf{G}(K'_w, id_5), \mathcal{E}(K_\xi, id_5))$ in group $\mathbf{GRP}[I_w^{grp}]$, and (4) ignore ciphertext $\mathbf{CDB}[L_1]$ since it contains $\mathbf{G}(K'_w, id_4)$.

Thirdly, the server empties structures $\mathbf{CDB}[L_4/L_3/L_2/L_1]$ and returns the three matching and still-valid ciphertexts in group $\mathbf{GRP}[I_w^{grp}]$ to the client. Finally, the client decrypts the returned search results and updates his counter values. The lower-side rectangles of Fig. 1 shows the new states of both the client and the server after the search, respectively.

4 Evaluation

We code `Bestie` and comprehensively evaluate its performance. To show the significant advantages of `Bestie`, we compare `Bestie` with five existing *forward-and-backward-secure* DSSE schemes, such as `Fides` and $\texttt{Diana}_{del}$ both from [2], `Mitra` and `Mitra`* both from [4], and `Janus++` [29].

4.1 Implementation

We code `Bestie` and recode the above five existing DSSE schemes for unifying their security parameters and testing environments.

Table 2. Hardware and software configuration

Hardware platform			
CPU	Intel E5-2630 v3	Memory	64 GB
Software environment			
Operation system	CentOS 7.3.1611 x64	Compiler	GCC 4.8.5
Cryptographic library	OpenSSL 1.1.1c [10]	Mathematical library	GMP Library 6.1.2 [9]

Programming Environment. Table 2 lists the hardware and software configurations. We code the above mentioned six DSSE schemes with C++ language and choose suitable security parameters to guarantee that they satisfy the 128-bit security level. To mitigate the influence of hard disk I/O, we store all generated ciphertexts in memory with the C++ STL library's data structures, such as *unordered_map*. Some graphs of our experiment are produced with Matplotlib [15].

Cryptographic Primitives. We apply the OpenSSL library and choose the SHA-2 hash family, the SHA-256-based HMAC, and AES-128 encryption to realize the hash function, the PRF function, and the symmetric encryption that appear in above mentioned six DSSE schemes, respectively. `Janus++` is the most complex one to be coded compared with others. In `Janus++`, each ciphertext includes a unique tag. When the client wants to delete a ciphertext, he generates a punctured key according to the ciphertext tag. This key can revoke the capability of the server to decrypt the ciphertext. The chosen binary length of the tag (namely, 16 bits) in [29] is too short to support a large volume of data. Hence, we apply the SHA-256-based HMAC algorithm to securely generate 256-bit tag. In addition, we set the maximum deletion number of `Janus++` to be 2,000 and apply $\texttt{Diana}_{del}$ as a building block to realize `Janus++`. We apply the GMP library to realize the 2048-bit RSA-based trapdoor permutation employed by `Fides`.

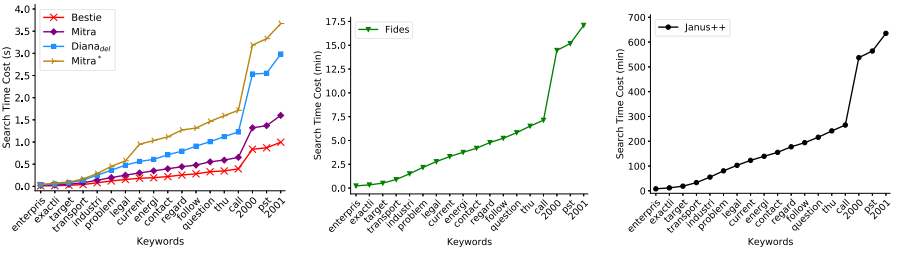
4.2 Data Description

Table 3. The 18 representative keywords and their frequencies

Keyword	Freq.	Keyword	Freq.	Keyword	Freq.	Keyword	Freq.	Keyword	Freq.
2001	246,613	pst	218,860	2000	208,551	call	102,807	thu	93,835
question	83,882	follow	75,409	regard	68,923	contact	60,270	energi	54,090
current	47,707	legal	39,923	problem	31,282	industri	21,472	transport	12,879
target	7,311	exactli	4,644	enterpris	3,130				

We run the *Porter Stemming* algorithm [24] on the Enron dataset [20] to extract keywords as [16] does. We extract a total of 42,014,587 keyword-and-file-identifier entries after excluding stop words, such as "a", "an", and "the". The total number of distinct keywords is 856,131. To succinctly present experimental results, we choose the 18 representative keywords listed in Table 3. These keywords have distinct frequencies. In particular, we choose the keyword "2001", which has the highest frequency, as a representative one. Ultimately, our test dataset includes 1,381,588 keyword-and-file-identifier entries. In addition, for each email, we set its file identifier to be the hexadecimal representation of the SHA-256 hash digest of its full path. Each file identifier consists of 64 bytes.

4.3 Experimental Results



(a) `Bestie`, `Mitra`, `Diana`$_{del}$, and `Mitra`*. (b) `Fides`. (c) `Janus++`.

Fig. 2. Total Search time cost without deletion.

Search Performance without Any Deletion. The search performance consists of three respects, such as total Search time cost, bandwidth cost, and client time cost. Note that the search performance here does not contain any deletion operation. Figure 2 compares the above-mentioned six schemes in terms of total Search time cost, namely the sum of both the server and the client time costs. The numerical results demonstrate that `Bestie` is the best one. It achieves a

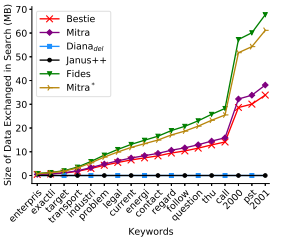
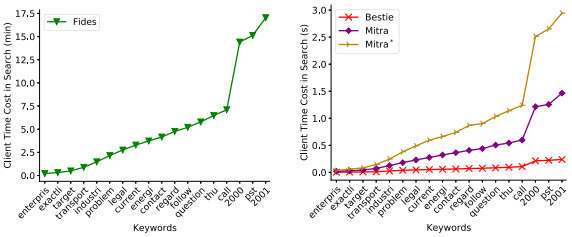


Fig. 3. Bandwidth cost without deletion.



(a) `Fides`. (b) `Bestie`, `Mitra`, and `Mitra`*.

Fig. 4. Client time cost without deletion.

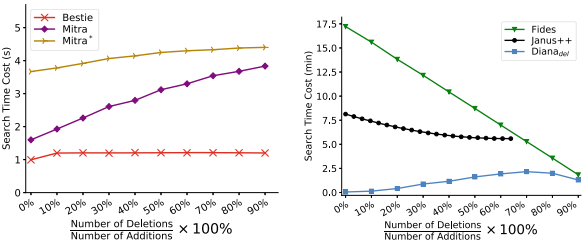
Search time cost that is at least 2, 1,032, and 38,332 times faster than both `Mitra`* and `Diana`$_{del}$, `Fides`, and `Janus++`, respectively. `Bestie` is also faster than `Mitra`. `Bestie` takes approximately 3.66 microseconds to find one matching ciphertext in average.

Figure 3 gives comparisons in terms of bandwidth cost. The bandwidth cost means the transferred data during a search. Figure 3 shows that `Bestie` achieves the lowest bandwidth cost compared with those schemes that need the client to return file identifiers to the server, like `Mitra`, `Mitra`*, and `Fides`. Both `Janus++` and `Diana`$_{del}$ achieve the cheapest bandwidth cost, since they allow the server to learn the file identifiers without the help of the client.

Figure 4 compares `Bestie`, `Mitra`, `Mitra`*, and `Fides` in terms of client time cost. We omit the comparisons with both `Janus++` and `Diana`$_{del}$, since both of them require quite small client time cost during a search. The numerical results show that the client time cost of `Bestie` to find one matching ciphertext is approximately 0.89 microsecond, which is the minimum one in Fig. 4.

Search Performance with Deletions. In this part, we consider deletion operations and test the search performance again. This experiment selects keyword "2001" as a representation and its corresponding file identifiers as the test dataset for testing `Bestie`, `Mitra`, `Mitra`*, `Fides`, and `Diana`$_{del}$. In contrast, we select keyword "enterpris" for testing `Janus++`. The difference is caused by the reasons that the maximum deletion number of `Janus++` is set to be 2,000, and this number is much smaller than the total number of file identifiers of keyword "2001". If we set the maximum deletion number to be a much bigger one, the Search time cost of `Janus++` is too high. To test the search performance, we add 246,613 (3,130, resp.) ciphertexts for keyword "2001" ("enterpris", resp.) at first. Secondly, we issue different number of Delete queries and then search keyword "2001" ("enterpris", resp.) over these ciphertexts. Each Delete query is to delete a randomly chosen file identifier.

Figure 5 shows how the different number of Delete queries affect the total Search time cost of a scheme. The numerical results show that `Bestie` achieves the lowest total Search time cost compared with others. Moreover, the increasing number of Delete queries has very little influence on the total Search time cost of



(a) `Bestie`, `Mitra`, and `Mitra`*. (b) `Fides`, `Janus++`, and `Diana`$_{del}$.

Fig. 5. Total Search time cost with deletions.

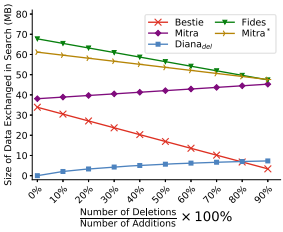
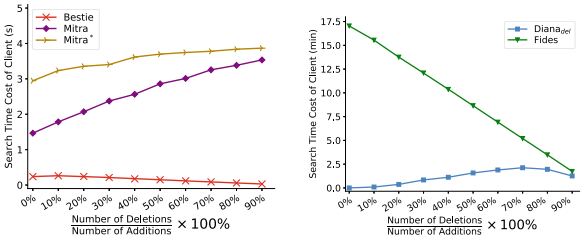


Fig. 6. Bandwidth cost with deletions.

`Bestie`. This is because that the total number of memory-writing operations in protocol `Bestie`.Search is decreasing along with the increasing number of Delete queries. The saved time cost caused by the decreasing number of memory-writing operations nearly counteracts the time cost caused by the increasing number of Delete queries.

Figure 6 show that `Bestie` achieves the lowest bandwidth cost compared with `Mitra`, `Mitra`*, and `Fides`. Moreover, the bandwidth cost of `Bestie` is decreasing along with the increasing number of Delete queries. We omit the comparison with `Janus++` in terms of bandwidth cost, since `Janus++` does not need the client to help the server get the matching file identifiers.

Figure 7 shows that `Bestie` saves at least 80% client time cost compared with others except `Janus++` if $\frac{\text{Number of Deletions}}{\text{Number of Additions}} \times 100\% \geq 10\%$. We also omit to test the client time cost of `Janus++` due to the same reason as the last experiment.



(a) `Bestie`, `Mitra`, and `Mitra`*. (b) `Diana`$_{del}$ and `Fides`.

Fig. 7. Client time cost with deletions.

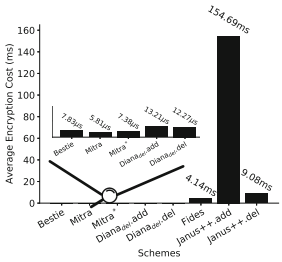


Fig. 8. Update time cost.

Update Time Cost. This experiment tests the time cost to generate one Update query, including Add and Delete queries. The above-mentioned six schemes has an identical time cost in generating that two kinds of queries, except `Diana`$_{del}$ and `Janus++`. Figure 8 shows that `Bestie` has a little more Update time cost only than both `Mitra` and `Mitra`*. But `Bestie` is more useful since both `Mitra` and `Mitra`* cannot achieve non-interactive real deletion.

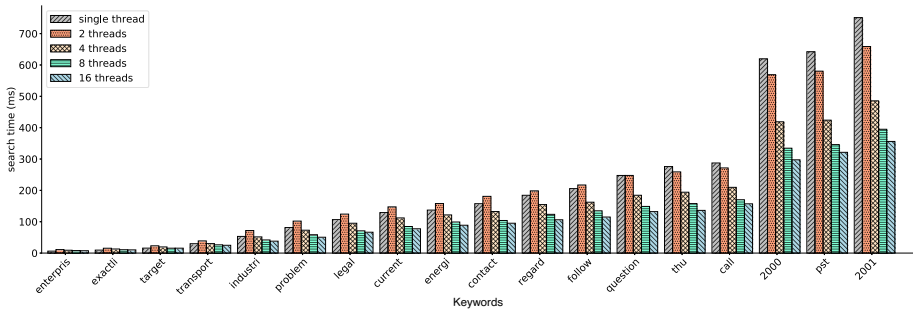


Fig. 9. Time costs of `Bestie.Search` with different threads.

Parallel Keyword Search. Section 3 explains that `Bestie` can achieve parallel keyword search. This means that compared with other schemes, `Bestie` can also retain its performance advantage when searching a keyword in the multi-threading setting. To highlight this feature, we implement the search procedures of the `Bestie.Search` protocol with different numbers of threads, such as two, four, eight, and sixteen threads. According to the numerical results shown in Fig. 9, generally, the more threads the server uses, the more efficient the search performance `Bestie` can obtain. The opposite results are observed only in the case of two threads when the number of matching ciphertexts is too small. The main reason for such an exception is the competition among threads for writing memory.

5 Other Related Works

Traditional SSE Schemes. The seminal work of Song *et al.* in 2000 [25] started the research on SSE. The proposed SSE scheme needs to scan the whole database during the search therefore is not efficient. In 2006, Curtmola *et al.* firstly improved the search efficiency of SSE to the sub-linear setting [7]. However, their construction with sub-linear search efficiency is not secure against adaptive chosen-keyword attacks (CKA2). Chase *et al.* exhibited the first CKA2-secure SSE scheme that has sub-linear search complexity [6] in 2010. Their work adopts padding in the encryption algorithm, which results in waste of storage and bandwidth resources. In 2012, Islam *et al.* introduced the first attack against SSE, which relies on the access pattern leakage of SSE and some background knowledge [16], such as the distribution of the keyword space.

Early DSSE Schemes. In 2010, Liesdonk *et al.* proposed the first CKA2-secure DSSE scheme, for which the search efficiency is logarithmic with the number of unique keywords [22]. However, the underlying cryptographical primitives produce redundant data, which results in a non-optimal search bandwidth. Additionally, the scheme is not scalable enough to support a large number of file identifiers. In 2012, Kamara *et al.* formally defined the security notion of DSSE

over information leakage [7] and presented a CKA2-secure DSSE construction with sub-linear search efficiency [18]. In 2013, Kamara *et al.* presented a CKA2-secure DSSE scheme that is built upon a keyword red-black tree, and the scheme supports parallel search [17]. However, the generated encrypted index of their construction is very large. Cash *et al.* later proposed a counter-based DSSE scheme that supports very large datasets [3]. In the same year, Stefanov *et al.* proposed a DSSE scheme that is forward secure [27]. However, the Update protocol of their scheme is costly because it requires the client to interact with the server to perform re-encryption operations. In 2014, Hahn *et al.* constructed an efficient DSSE scheme [12]. However, when adding searchable ciphertexts, the scheme directly classifies the ever-searched keywords and stores the corresponding ciphertexts in specific places, which reveals the semantic information of the ciphertexts. In 2017, Xu *et al.* proposed a DSSE scheme with sub-linear search efficiency and small leakage [30]. The scheme constructs a hidden chain in those generated ciphertexts that contain the same keyword and combines the physical and logical deletion. In 2016, Zhang *et al.* presented the file-injection attack against DSSE [31]. The file-injection attack is a proactive attack wherein the adversary injects files into the target scheme.

Forward-and-Backward-Secure DSSE Schemes. As illustrated by Zhang *et al.*, forward security is an important property that can defend against file-injection attacks [31]. The first forward-secure DSSE scheme was proposed by Chang *et al.* in 2005 [5]. In 2014, Stefanov *et al.* first formally defined forward security with leakage functions [27]. Later, Bost improved the performance of the proposed scheme in the aspects of both computation and bandwidth by using the trapdoor permutation [1]. In 2016, Garg *et al.* constructed a forward-secure DSSE scheme based on their TWORAM [11], but it suffers from low Update and Search efficiency. In 2017, Kim *et al.* utilized the dual dictionary to construct a forward-secure DSSE scheme that supports real deletion [19]. In 2018, Song *et al.* proposed a counter-based forward-secure DSSE scheme with real deletion support [26]. Their proposed scheme achieves I/O efficiency by caching historical search results.

In 2014, Stefanov *et al.* proposed backward security, which requires that search queries cannot be executed over deleted ciphertexts [27]. In 2016, Hoang *et al.* presented *forward-and-backward-secure* DOD-DSSE [14]. The core idea of DOD-DSSE is to let the client fetch all related data from the server and to perform Search or Update operations locally. In 2017, Bost *et al.* formulated backward security with leakage functions [2]. In their work, they defined three types of backward security for ensuring different levels of security strength. In work [2], Bost *et al.* constructed four *forward-and-backward-secure* DSSE schemes: `Fides`, $\texttt{Diana}_{Del}$, `Moneta` and `Janus`. `Moneta` is based on TWORAM, which is not practical. In 2019, Li *et al.* constructed a *forward-and-backward-secure* DSSE with the hidden pointer ciphertext structure and partition search technique [21]. In 2020, He *et al.* [13] presented a *forward-and-backward-secure* DSSE scheme with constant client storage. He *et al.*'s approach is to combine the counter and chain structure and use the global counter to find all chain structures of ciphertexts.

In the same year, Demertzis *et al.* [8] proposed three *forward-and-backward-secure* DSSE scheme with constant client storage. The first two schemes in [8] achieve interactive real deletion, and the third scheme uses *oblivious map* and a tree-based encrypted index as building blocks. Recently in 2021, Sun *et al.* [28] proposed a *forward-and-backward-secure* DSSE scheme `Aura`, which achieves non-interactive real deletion in the cost of extra client-side storage resources to stash Delete queries.

6 Conclusion

In this paper, we analyse the requirements for a DSSE scheme to be practical and note that most existing *forward-and-backward-secure* DSSE schemes are not practical enough in terms of performance or real deletion. This fact motivates us to propose `Bestie`, a very practical and *forward-and-backward-secure* DSSE scheme. `Bestie` also supports non-interactive real deletion and parallel keyword search. The experimental results show that `Bestie` achieves the best Search time cost compared with `Fides`, `Mitra`, `Mitra`*, `Diana`$_{del}$, and `Janus++`. During a search, `Bestie` achieves the lowest client time cost and the lowest bandwidth cost compared with `Fides`, `Mitra`, and `Mitra`*. Although `Bestie` introduces a little more client time cost and bandwidth cost during a search than `Janus++` and `Diana`$_{del}$, it is much faster than `Janus++` during a search and achieves non-interactive real deletion that `Diana`$_{del}$ fails to achieve. In summary, `Bestie` is very practical, especially for managing large-scale data.

Acknowledgements. We would like to thank our shepherd Prof. Xun Yi and the anonymous reviewers for their insightful comments and valuable suggestions. This work was partly supported by the National Natural Science Foundation of China under Grant No. 61872412, the Wuhan Applied Foundational Frontier Project under Grant No. 2020010601012188, and the Guangdong Provincial Key R&D Plan Project under Grant No. 2019B010139001.

A Proof of Theorem 1

Proof. To prove the *forward and Type-III-backward security* of the proposed `Bestie`, we construct a $\mathcal{S}$ simulator, which takes as inputs leakage functions $\mathcal{L}^{Stp}(\lambda) = \lambda$, $\mathcal{L}^{Updt}(op, w, id) = \emptyset$, and $\mathcal{L}^{Srch}(w) = \{\text{sp}(w), \text{TimeDB}(w), \text{DelHist}(w)\}$ to simulate protocols `Bestie`.Setup, `Bestie`.Update, and `Bestie`.Search, respectively. We will demonstrate that the simulated `Bestie` is indistinguishable from the real `Bestie` under the adaptive attacks. Algorithm 2 describes the simulator $\mathcal{S}$. Specifically, the simulator $\mathcal{S}$ consists of the following three phases.

Protocol $\mathcal{S}$.Setup**.** This protocol simulates protocol `Bestie`.Setup. In this protocol, simulator $\mathcal{S}$ takes leakage function $\mathcal{L}^{Stp}(\lambda) = \lambda$ as input and initializes five empty maps **CDB**, **GRP**, **CipherList**, **GIndlist**, and **Xlist**, where

Algorithm 2. Construction of Simulator $\mathcal{S}$ in the Ideal Game of `Bestie`

$\mathsf{Setup}(\mathcal{L}^{Stp}(\lambda))$

1: Initialize five empty maps **CDB**, **GRP**, **CipherList**, **GIndlist** and **Xlist** and a timestamp parameter $u \leftarrow 0$
2: Send $\mathbf{EDB} \leftarrow (\mathbf{CDB}, \mathbf{GRP})$ to the server

$\mathsf{Update}(\mathcal{L}^{Updt}(op, (w, id)))$

1: Accumulate the timestamp parameter by setting $u \leftarrow u + 1$
2: Randomly pick a triplet $(L, D, C) \xleftarrow{\$} \{0,1\}^{\lambda'} \times \{0,1\}^{\lambda+1} \times \mathcal{C}_\xi$ as the generated ciphertext
3: Record the triplet (L, D, C) in $\mathbf{CipherList}[u]$
4: Send the triplet (L, D, C) to the server

$\mathsf{Search}(\mathcal{L}^{Srch}(w) = (\mathrm{sp}(w), \mathrm{TimeDB}(w), \mathrm{DelHist}(w))$

1: Accumulate the timestamp parameter by setting $u \leftarrow u + 1$
2: Extract the timestamp u_0 of the last Search query from $\mathrm{sp}(w)$, where $u_0 = 0$ if $\mathrm{sp}(w) = \emptyset$
3: Extract all timestamps $\{u_1, ..., u_n\}$ between u_0 and u from both $\mathrm{TimeDB}(w)$ and $\mathrm{DellHist}(w)$, where $u_i < u_j$ if $i < j$
4: Abort if $n = 0$ and $u_0 = 0$ (namely, keyword w never appears in any historical Update query)
5: Extract the timestamp u_s of the first Search query from $\mathrm{sp}(w)$, where $u_s = u$ if no earlier Search query of keyword w occurs
6: If the index $I^{grp}_{u_s}$ stored in $\mathbf{GIndlist}[u_s]$ is NULL, then randomly choose $I^{grp}_{u_s} \xleftarrow{\$} \{0,1\}^\lambda$ and update $\mathbf{GIndlist}[u_s] \leftarrow I^{grp}_{u_s}$
7: Randomly choose a key $K \xleftarrow{\$} \{0,1\}^\lambda$ for searching keyword w, and retrieve $I^{grp}_{u_s} \leftarrow \mathbf{GIndlist}[u_s]$
8: **for** $i = 1$ to n **do**
9: Retrieve the simulated ciphertext $(L_{u_i}, D_{u_i}, C_{u_i}) \leftarrow \mathbf{CipherList}[u_i]$
10: If the timestamp u_i was extracted from $\mathrm{TimeDB}(w)$ in the above Step 3, then set parameter $u_{min} = u_i$
11: Otherwise (namely, the timestamp u_i was extracted from $\mathrm{DelHist}(w)$), set parameter u_{min} to be the smaller of the timestamp u_i and the timestamp paired with u_i in $\mathrm{DelHist}(w)$
12: If $\mathbf{Xlist}[u_{min}]$ is NULL, then randomly set $\mathbf{Xlist}[u_{min}] \xleftarrow{\$} \{0,1\}^\lambda$
13: If the Update query at timestamp u_i has operation type add, then program oracle $\mathbf{H}$ such that $\mathbf{H}(K, i) = (L_{u_i} || D_{u_i}) \oplus (0^{\lambda'} || add || \mathbf{Xlist}[u_{min}])$
14: Otherwise, program oracle $\mathbf{H}$ such that $\mathbf{H}(K, i) = (L_{u_i} || D_{u_i}) \oplus (0^{\lambda'} || del || \mathbf{Xlist}[u_{min}])$
15: **end for**
16: Send search trapdoor (n, K, I^{grp}_w) to the server
17: **return** all file identifiers in $\mathrm{TimeDB}(w)$ after receiving the response from the server

$(\mathbf{CDB}, \mathbf{GRP})$ will be sent to the server and the remaining maps are kept as internal states of simulator $\mathcal{S}$. The map **CipherList** records the ciphertexts generated by simulator $\mathcal{S}$. The map **GIndList** records the group indexes for the following Search queries. Map **XList** records the simulated values of hash function **G**. Clearly, the simulated protocol is indistinguishable from the real one in the view of adversary $\mathcal{A}$.

Protocol $\mathcal{S}$.Update. This protocol simulates protocol `Bestie.Update`. In this protocol, simulator $\mathcal{S}$ takes nothing as input. It randomly chooses a random triplet (L, D, C) as the generated ciphertext and uploads it to the server. In the real world, **H** is a collision-free hash function, and ξ is a semantically secure symmetric encryption scheme. Hence, the random triplet is indistinguishable from a real ciphertext if adversary $\mathcal{A}$ does not know the corresponding search trapdoor. The following content will prove that the random triplet is still indistinguishable from a real ciphertext, even in the opposite case.

Protocol $\mathcal{S}$.Search**.** This protocol simulates protocol Bestie.Search. In this protocol, simulator $\mathcal{S}$ takes the leaked information $\text{sp}(w)$, $\text{TimeDB}(w)$, and $\text{DelHist}(w)$ as inputs. Simulator $\mathcal{S}$ first checks whether there exists any historical Update query about keyword w. If not, simulator $\mathcal{S}$ aborts, as the real protocol does (refer to Steps 2 to 4). Otherwise, simulator $\mathcal{S}$ sets or retrieves the group index $I_{u_s}^{grp}$ of keyword w (refer to Steps 5 and 6). In the following content, simulator $\mathcal{S}$ must program oracle $\mathbf{H}$ such that the randomly generated search trapdoor is still valid in the view of adversary $\mathcal{A}$ (refer to Steps 8 and 15).

In this part, simulator $\mathcal{S}$ mainly achieves two aims: (1) simulate hash values of function $\mathbf{G}$ for all Update queries of keyword w as well as guarantee that the Update queries of the same keyword-and-file-identifier entry have the same hash value (refer to Steps 10 to 12) and (2) program oracle $\mathbf{H}$ such that all simulated ciphertexts of keyword w can be correctly found by the server with the randomly generated search trapdoor (refer to Steps 13 and 14). Finally, simulator $\mathcal{S}$ sends the randomly generated search trapdoor to the server. The transcripts generated by the simulated Search protocol are indistinguishable from those of the real protocol since all operations are consistent with the real protocol in the view of adversary $\mathcal{A}$.

To summarize, there exists a $\mathcal{S}$ simulator to simulate Bestie with the given leakage functions, and the simulation is indistinguishable from the real Bestie. Thus, Theorem 1 is true. □

References

1. Bost, R.: $\sum o\varphi o\varsigma$: forward secure searchable encryption. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, pp. 1143–1154 (2016)
2. Bost, R., Minaud, B., Ohrimenko, O.: Forward and backward private searchable encryption from constrained cryptographic primitives. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, pp. 1465–1482 (2017)
3. Cash, D., et al.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA (2014)
4. Chamani, J.G., Papadopoulos, D., Papamanthou, C., Jalili, R.: New constructions for forward and backward private symmetric searchable encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, pp. 1038–1055 (2018)
5. Chang, Y.-C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005). https://doi.org/10.1007/11496137_30
6. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 577–594. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17373-8_33
7. Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of

the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, pp. 79–88 (2006)
8. Demertzis, I., Chamani, J.G., Papadopoulos, D., Papamanthou, C.: Dynamic searchable encryption with small client storage. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA. The Internet Society (2020)
9. Foundation, F.S.: The GNU MP bignum library. https://gmplib.org/. Accessed 8 Oct 2019
10. Foundation, O.S.: OpenSSL. https://www.openssl.org/. Accessed 8 Oct 2019
11. Garg, S., Mohassel, P., Papamanthou, C.: TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016, Part III. LNCS, vol. 9816, pp. 563–592. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53015-3_20
12. Hahn, F., Kerschbaum, F.: Searchable encryption with secure and efficient updates. In: Ahn, G., Yung, M., Li, N. (eds.) Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, pp. 310–320. ACM (2014)
13. He, K., Chen, J., Zhou, Q., Du, R., Xiang, Y.: Secure dynamic searchable symmetric encryption with constant client storage cost. IEEE Trans. Inf. Forensics Secur. **16**, 1538–1549 (2021)
14. Hoang, T., Yavuz, A.A., Guajardo, J.: Practical and secure dynamic searchable encryption via oblivious access on distributed data structure. In: Schwab, S., Robertson, W.K., Balzarotti, D. (eds.) Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, pp. 302–313. ACM (2016)
15. Hunter, J.D.: Matplotlib: a 2D graphics environment. Comput. Sci. Eng. **9**(3), 90–95 (2007)
16. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA (2012)
17. Kamara, S., Papamanthou, C.: Parallel and dynamic searchable symmetric encryption. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 258–274. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_22
18. Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: the ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, pp. 965–976 (2012)
19. Kim, K.S., Kim, M., Lee, D., Park, J.H., Kim, W.: Forward secure dynamic searchable symmetric encryption with efficient updates. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, pp. 1449–1463. ACM (2017)
20. Klimt, B., Yang, Y.: Introducing the Enron corpus. In: CEAS 2004 - First Conference on Email and Anti-Spam, Mountain View, California, USA (2004)
21. Li, J., et al.: Searchable symmetric encryption with forward search privacy. IEEE Trans. Dependable Secur. Comput. **18**(1), 460–474 (2021)
22. van Liesdonk, P., Sedghi, S., Doumen, J., Hartel, P., Jonker, W.: Computationally efficient searchable symmetric encryption. In: Jonker, W., Petković, M. (eds.) SDM 2010. LNCS, vol. 6358, pp. 87–100. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15546-8_7

23. Parliament, E., Council: on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation) (2016). https://eur-lex.europa.eu/eli/reg/2016/679/oj. Accessed 16 Jan 2020
24. Porter, M.F.: An algorithm for suffix stripping. Program **14**(3), 130–137 (1980)
25. Song, D.X., Wagner, D.A., Perrig, A.: Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, pp. 44–55 (2000)
26. Song, X., Dong, C., Yuan, D., Xu, Q., Zhao, M.: Forward private searchable symmetric encryption with optimized I/O efficiency. IEEE Trans. Dependable Secur. Comput. **17**(5), 912–927 (2020)
27. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA (2014)
28. Sun, S., et al.: Practical non-interactive searchable encryption with forward and backward privacy. In: 28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, 21–25 February 2021. The Internet Society (2021). https://www.ndss-symposium.org/ndss-paper/practical-non-interactive-searchable-encryption-with-forward-and-backward-privacy/
29. Sun, S., et al.: Practical backward-secure searchable encryption from symmetric puncturable encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, pp. 763–780 (2018)
30. Xu, P., Liang, S., Wang, W., Susilo, W., Wu, Q., Jin, H.: Dynamic searchable symmetric encryption with physical deletion and small leakage. In: Pieprzyk, J., Suriadi, S. (eds.) ACISP 2017, Part I. LNCS, vol. 10342, pp. 207–226. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60055-0_11
31. Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: the power of file-injection attacks on searchable encryption. In: 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, pp. 707–720 (2016)
32. Zuo, C., Sun, S.-F., Liu, J.K., Shao, J., Pieprzyk, J.: Dynamic searchable symmetric encryption with forward and stronger backward privacy. In: Sako, K., Schneider, S., Ryan, P.Y.A. (eds.) ESORICS 2019, Part II. LNCS, vol. 11736, pp. 283–303. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29962-0_14