Dynamic Searchable Symmetric Encryption With Strong Security and Robustness

Haochen Dou, Zhenwu Dan, Peng Xu, *Member, IEEE*, Wei Wang, *Member, IEEE*, Shuning Xu, Tianyang Chen, and Hai Jin, *Fellow, IEEE*

***Abstract*— Dynamic Searchable Symmetric Encryption (DSSE) is a prospective technique in the field of cloud storage for secure search over encrypted data. A DSSE client can issue `update` queries to an honest-but-curious server for adding or deleting his ciphertexts to or from the server and delegate keyword `search` over those ciphertexts to the server. Numerous investigations focus on achieving strong security, like *forward-and-Type-I$^-$-backward* security, to reduce the information leakage of DSSE to the server as much as possible. However, the existing DSSE with such strong security cannot keep search correctness and stable security (or *robustness*, in short) if irrational queries are issued by the client, like duplicate `add` or `delete` queries and the `delete` queries for removing non-existed entries, to the server unintentionally. Hence, this work proposes two new DSSE schemes, named $\texttt{SR-DSSE}_a$ and $\texttt{SR-DSSE}_b$, respectively. Both two schemes achieve *forward-and-Type-I$^-$-backward* security while keeping *robustness* when irrational queries are issued. In terms of performance, $\texttt{SR-DSSE}_a$ has more efficient communication costs and roundtrips than $\texttt{SR-DSSE}_b$. In contrast, $\texttt{SR-DSSE}_b$ has a more efficient search performance than $\texttt{SR-DSSE}_a$. Its search performance is close to the existing DSSE scheme with the same security but fails to achieve *robustness*.**

***Index Terms*— Dynamic searchable symmetric encryption, forward security, backward security, robustness.**

Manuscript received 6 January 2023; revised 7 September 2023 and 8 December 2023; accepted 26 December 2023. Date of publication 5 January 2024; date of current version 11 January 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2021YFB3101304, in part by the National Natural Science Foundation of China under Grant 62272186 and Grant 62372201, and in part by the Innovation Project of Jinyinhu Laboratory under Grant 2023JYH010103. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Sushmita Ruj. *(Corresponding author: Peng Xu.)*

Haochen Dou is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Laboratory, and the Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China, and also with the State Key Laboratory of Cryptology, Beijing 100878, China (e-mail: haochendou@mail.hust.edu.cn).

Zhenwu Dan, Shuning Xu, and Tianyang Chen are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Laboratory, and the Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: danzw@mail.hust.edu.cn; xusn@mail.hust.edu.cn; chentianyang@mail.hust.edu.cn).

Peng Xu is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Laboratory, and the Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China, and also with the Jinyinhu Laboratory, Wuhan 430040, China (e-mail: xupeng@mail.hust.edu.cn).

Wei Wang is with the Cyber-Physical-Social Systems Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: viviawangww@gmail.com).

Hai Jin is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Laboratory, and the Cluster and Grid Computing Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: hjin@hust.edu.cn).

Digital Object Identifier 10.1109/TIFS.2024.3350330

I. Introduction

DYNAMIC Searchable Symmetric Encryption (DSSE) [1] is a widely used technique for performing secure keyword searches over ciphertexts that are constantly changing. In DSSE applications, all data of the client is encrypted and stored in remote environments like the cloud, which helps to maintain data confidentiality. DSSE enables the client to issue `update` queries to add or delete ciphertexts to or from the cloud and delegate keyword `search` queries over his ciphertexts to the cloud while maintaining keyword confidentiality [2]. Many software products, such as the Mistubishi Information and Communication System[1] and the Crypteron security platform,[2] have made extensive use of DSSE.

Recently, numerous researchers have paid attention to developing DSSE with strong security to restrict the information leakage of DSSE as much as possible. To address these concerns, Stefanov et al. paid an apparent effort by defining two new kinds of security, named *forward* security and *backward* security [3]. The former restricts that information about the earlier queries' keywords is not leaked by any new `update` query, while the latter guarantees that an attacker cannot learn "too much" information about `update` queries issued between any two adjacent `search` queries. Following the seminal work, Bost et al. categorized *backward* security into three different types (from the strongest one to the weakest one), which is denoted as *Type-I*, *Type-II*, and *Type-III*, respectively, to restrict the information leakage in the degree from strong to weak [4]. To restrict the information leakage further, Zuo et al. proposed *Type-I$^-$-backward* security, which is the strongest one so far as we know [5].

In brief, *Type-I$^-$-backward* security requires that the information leakage caused by a `search` query contains which files match the query and when the related `update` queries

[1] https://www.mitsubishielectric.com/en/about/rd/research/highlights/communications

[2] https://www.crypteron.com/

are issued. Note that in *Type-I⁻-backward* security, an attacker cannot distinguish add and delete queries. Compared with the *Type-I⁻-backward* security, the *Type-I-backward* security allows the search query to have an additional leakage of the time when the related add queries are issued. In *Type-II-backward* security, leakage of the search query additionally consists of when the related add and delete queries are issued. Finally, compared with *Type-II-backward* security, the weaker *Type-III-backward* security also allows a search query to leak the relationships between the related add and delete queries, namely which add queries a delete query wants to remove.

A. Motivation

For the time being, FB−DSSE is firstly proposed to achieve *forward-and-Type-I⁻-backward* security [5]. It uses a bitmap index to represent all possible files' identifiers. Each FB−DSSE ciphertext contains a keyword and an assigned bitmap index to denote which files pair with the keyword. When receiving a keyword search query, the server computes corresponding indexes at the beginning. Using these indexes, the server retrieves relevant ciphertexts, aggregates those ciphertexts into one by an addition homomorphic operation, and finally returns the client the aggregated ciphertext. The returned ciphertext contains an assigned bitmap index to denote all the files matching the search query. Later, three other DSSE schemes, named FBDSSE−CQ, SFBDSSE−CQ [9], and FBDSSE−RQ [10], respectively, were proposed to obtain the *forward-and-Type-I⁻-backward* security. In particular, both FBDSSE−CQ and SFBDSSE−CQ aim at achieving conjunctive keyword search, whereas FBDSSE−RQ aims at solving range keyword search.

All the aforementioned works have achieved *forward-and-Type-I⁻-backward* security. However, they fail to ensure stable search correctness and security (referred to as *robustness* for brevity) in case the client unintentionally issues irrational queries. The *robustness* of DSSE was first investigated by Xu et al. [8]. They demonstrated that a practical DSSE scheme must be robust since it is very hard to avoid the mistake caused by the careless client, like issuing duplicate add or delete queries, or removing non-existed entries by delete queries either. And they constructed a DSSE scheme named ROSE to obtain the *robustness* and the *forward-and-Type-III-backward* security. Hence, a natural open problem is thus: *"Could we construct a DSSE scheme to obtain the* robustness *and the* forward-and-Type-I⁻-backward *security simultaneously?"*

B. Our Contributions

We propose a solid answer to the question in this work. First, before giving our solutions, we have to redefine *forward-and-Type-I⁻-backward* security, such that the new definition allows an attacker to issue irrational queries to simulate the careless client (in Section II). Note that the traditional definition of *forward-and-Type-I⁻-backward* security implicitly assumes that irrational queries are not considered. Second, we find that the bitmap index adapted in FB−DSSE cannot correctly represent the client's update queries during search queries. For example, if two duplicate add queries are issued to insert the same keyword and file, it is natural to require that the correct search results contain this file. But the bitmap index returned by the search query of FB−DSSE represents that this file is removed. We give an efficient solution to this problem, which is constructing a new kind of bitmap index, named *bi-bitmap index*, and designing a particular boolean circuit to support the ciphertexts' aggregation when searching a keyword, such that the returned bi-bitmap index can represent the correct search results (in Section III).

In Section IV, we construct the first DSSE scheme, named SR−DSSE$_a$, to achieve *robustness* and *forward-and-Type-I⁻-backward* security simultaneously. SR−DSSE$_a$ applies our bi-bitmap index and particular boolean circuit. To achieve the particular boolean circuit, SR−DSSE$_a$ applies Torus Fully Homomorphic Encryption (TFHE) [11]. If the client issues a search query and sends the trapdoor to the server, the SR−DSSE$_a$ server itself can retrieve and aggregate corresponding ciphertexts. Hence, SR−DSSE$_a$ achieves the non-interactive aggregation of ciphertexts. The search process of SR−DSSE$_a$ takes one communication roundtrip. SR−DSSE$_a$ also saves the client overhead when searching a keyword.

To improve the search performance, we construct the second DSSE scheme, named SR−DSSE$_b$, in Section V. SR−DSSE$_b$ has the same *robustness* and strong security as SR−DSSE$_a$. When searching a keyword, SR−DSSE$_b$ applies an interactive method to achieve the aggregation of ciphertexts. Specifically, after the server finds all matching ciphertexts, these ciphertexts are returned to the client. When receiving them, the client performs decryption and aggregates their contained bi-bitmap indexes. Finally, the client re-encrypts the aggregated index. This result is uploaded to the server for the next keyword search. Compared with SR−DSSE$_a$, SR−DSSE$_b$ prevents the server from running the expensive aggregation process and saves the search overhead of the server. Although SR−DSSE$_b$ increases the communication roundtrips and the client overhead, the total search performance is still much better than SR−DSSE$_a$.

Table I compares SR−DSSE$_a$ and SR−DSSE$_b$ with some previous DSSE schemes that achieve robustness (Moneta and ROSE), at least forward-and-Type-I-backward security (ORION and FB−DSSE), or state-of-the-art practical performance and bitmap-based index (IM−DSSE$_{\mathrm{II}}$ and IM−DSSE$_{\mathrm{I+II}}$). Compared to Moneta and ROSE, SR−DSSE$_a$ and SR−DSSE$_b$ achieve higher backward security. Compared to ORION and FB−DSSE, our proposed schemes achieve robustness. Finally, compared to IM−DSSE$_{\mathrm{II}}$ and IM−DSSE$_{\mathrm{I+II}}$, SR−DSSE$_a$ and SR−DSSE$_b$ achieve both robustness and higher level of backward security. Particularly, SR−DSSE$_b$ achieves higher search computation efficiency than Moneta and ROSE, and higher update computation efficiency than Moneta, ORION, IM−DSSE$_{\mathrm{II}}$, and IM−DSSE$_{\mathrm{I+II}}$.

In the experiment part, we test SR−DSSE$_a$ and SR−DSSE$_b$ and compare them with FB−DSSE in Section VI. First, we test the above three schemes regarding the client overhead during a keyword search. The experimental results show that SR−DSSE$_a$ has a constant client time cost. Both SR−DSSE$_b$

TABLE I

COMPARISONS WITH PRIOR DSSE WORKS. N IS THE TOTAL NUMBER OF KEYWORD/FILE-IDENTIFIER PAIRS, $|\mathbf{W}|$ DENOTES THE AMOUNT OF ALL DIFFERENT KEYWORDS, AND $|\mathbf{F}|$ DENOTES THE TOTAL NUMBER OF DISTINCT FILES. FOR KEYWORD w, a_w IS THE TOTAL NUMBER OF INSERTED ENTRIES, d_w IS THE NUMBER OF DELETE QUERIES, n_w IS THE NUMBER OF FILES CURRENTLY CONTAINING w, s_w IS THE NUMBER OF SEARCH QUERIES THAT OCCURRED, i_w IS THE TOTAL NUMBER OF ADD QUERIES, AND s'_w IS A NUMBER HAVING $s'_w \leq s_w$. ALL SCHEMES EXCEPT ROSE HAVE $a_w = n_w + d_w$. SPECIFICALLY, ROSE HAS $a_w = n_w + s'_w + d_w$. RT IS THE NUMBER OF ROUND TRIPS FOR SEARCH UNTIL THAT THE CLIENT OBTAINS THE MATCHING FILE IDENTIFIERS. FS AND BS STAND FOR FORWARD SECURITY AND BACKWARD SECURITY, RESPECTIVELY. COMP. AND COMM. ARE ABBREVIATIONS OF COMPUTATION AND COMMUNICATION, RESPECTIVELY. THE NOTATION $\widetilde{O}$ HIDES POLYLOGARITHMIC FACTORS

Scheme	Robust	FS	BS	Search Efficiency			Update Efficiency		Client Storage
				Comp.	Comm.	RT	Comp.	Comm.	
IM-DSSE$_{\text{II}}$ [6]	✗	✓	III	$O(\|\mathbf{F}\|)$	$O(\|\mathbf{F}\|)$	1	$O(\|\mathbf{W}\|)$	$O(\|\mathbf{W}\|)$	$O(\|\mathbf{W}\| + \|\mathbf{F}\|)$
IM-DSSE$_{\text{I+II}}$ [6]	✗	✓	III	$O(\|\mathbf{F}\|)$	$O(\|\mathbf{F}\|)$	1	$O(\|\mathbf{W}\|)$	$O(\|\mathbf{W}\|)$	$O(\|\mathbf{W}\| + \|\mathbf{F}\|)$
ORION [7]	✗	✓	I	$O(n_w \log^2 N)$	$(n_w \log^2 N)$	$O(\log N)$	$O(\log^2 N)$	$O(\log^2 N)$	$O(1)$
FB-DSSE [5]	✗	✓	I$^-$	$O(a_w)$	$O(\|\mathbf{F}\|)$	1	$O(1)$	$O(\|\mathbf{F}\|)$	$O(\|\mathbf{W}\| \times \log\|\mathbf{F}\|)$
ROSE [8]	✓	✓	III	$O((n_w + s'_w + 1)d_w)$	$O(n_w)$	1	$O(1)$	$O(1)$	$O(\|\mathbf{W}\| \times \log\|\mathbf{F}\|)$
Moneta [4]	✓	✓	I	$\widetilde{O}(a_w \log N + \log^3 N)$	$\widetilde{O}(a_w \log N + \log^3 N)$	2	$\widetilde{O}(\log^2 N)$	$\widetilde{O}(\log^3 N)$	$O(1)$
SR-DSSE$_a$ (Section IV)	✓	✓	I$^-$	$O(a_w\|\mathbf{F}\|)$	$O(\|\mathbf{F}\|)$	1	$O(\|\mathbf{F}\|)$	$O(\|\mathbf{F}\|)$	$O(\|\mathbf{W}\| \times \log\|\mathbf{F}\|)$
SR-DSSE$_b$ (Section IV)	✓	✓	I$^-$	$O(a_w)$	$O(\|\mathbf{F}\|)$	2	$O(1)$	$O(\|\mathbf{F}\|)$	$O(\|\mathbf{W}\| \times \log\|\mathbf{F}\|)$

and FB-DSSE have an increasing client time cost which is linear with the total amount of retrieved ciphertexts. For search bandwidth cost, both SR-DSSE$_a$ and FB-DSSE are constant, and SR-DSSE$_b$ takes a linear cost with the number of matching ciphertexts. Furthermore, if the number of matching ciphertexts is less than 4,210, the bandwidth cost of SR-DSSE$_b$ is cheaper than that of SR-DSSE$_a$.

Secondly, we test the total search time cost of SR-DSSE$_b$ and compare it with FB-DSSE. Note that the total search time cost consists of both the server's and the client's time cost during a keyword search. Both SR-DSSE$_b$ and FB-DSSE have the linear search time cost with the amount of matching ciphertexts. And SR-DSSE$_b$ is better than FB-DSSE in practice if a keyword has been searched several times. The main reasons are that the client time cost of SR-DSSE$_b$ relies on the increasing number of matching ciphertexts between two adjacent search queries. However, the client time cost of FB-DSSE is always determined by the total amount of matching ciphertexts.

In summary, our contributions are:

1) We redefine DSSE and its *forward-and-Type-I$^-$-backward* security in the context of *robustness* and design the bi-bitmap index and its boolean circuit as building blocks of our DSSE schemes.
2) We construct two new DSSE schemes, SR-DSSE$_a$ and SR-DSSE$_b$, to achieve *robustness* and *forward-and-Type-I$^-$-backward* security simultaneously. The two proposed schemes outperform previous DSSE works in many aspects, e.g., robustness, security, or performance.
3) Finally, we test SR-DSSE$_a$ and SR-DSSE$_b$ and compare them with FB-DSSE. The numerical results show that SR-DSSE$_a$ has a better client time cost, and the total search time cost of SR-DSSE$_b$ is better.

II. ROBUST DSSE AND ITS SECURITY DEFINITIONS

A robust DSSE scheme must keep search correctness and stable security even if the client issues irrational update queries, like the duplicate add or delete queries and the delete query to remove the nonexistent entry. Because the correctness and the security of DSSE are separately defined and not unified, we integrate robustness to those two properties, respectively. In this section, we will redefine the formal concept of DSSE and its *forward-and-Type-I$^-$-backward* security in the context of robustness.

Definition 1 (Robust DSSE): Three protocols are the core compose of a robust DSSE scheme Σ. They are:

- $\Sigma.\mathsf{Setup}(\lambda, n)$*: With the inputted security parameter λ and the maximum number n of files, the client initializes an empty encrypted database* ***EDB*** *(kept remotely), a master secret key K_Σ and a secret status σ (both kept locally by the client);*
- $\Sigma.\mathsf{Update}(K_\Sigma, \sigma, op, (w, \mathcal{F}); \textbf{EDB})$*: To update (add or delete) some files containing the same keyword w to the server, the client takes K_Σ, σ, and the entry $(w, \mathcal{F})$ as inputs, where $\mathcal{F}$ is the set of those files' identifiers, generates an update tokens and sends it to the server. Finally, the server updates* ***EDB*** *as the client's will;*
- $\Sigma.\mathsf{Search}(K_\Sigma, w, \sigma; \textbf{EDB})$*: Given the master secret key K_Σ, an expected keyword w, and the secret status σ, a corresponding search trapdoor is generated by the client and sent to the server. Then, all the ciphertexts containing keyword w are retrieved from* ***EDB****. Finally, the client outputs the file identifiers that are corresponding to the files containing keyword w.*

A robust DSSE must be consistent in any scenarios. That is, for any pair of keyword w and file identifier f, no matter how many times to update (add or delete) this pair, the output of protocol $\Sigma.\mathsf{Search}(K_\Sigma, w, \sigma; \textbf{EDB})$ always contain f if the final update is a add one, otherwise the output does not contain f.

Before redefining *forward-and-Type-I$^-$-backward* security, we redefine the $\mathcal{L}$-adaptive-security of a robust DSSE scheme Σ, where $\mathcal{L} = (\mathcal{L}^{Setup}, \mathcal{L}^{Update}, \mathcal{L}^{Search})$ includes DSSE setup, update, and search leakage functions, which denote the information leaked in each protocol. Compared to traditional security definition, the redefined security allows the adversary to issue irrational update queries. The adaptive security definition always includes two games: a game presenting the actual interactions named `Real` and a game presenting the simulated one named `IDEAL`. In the real game, an adversary can issue any `update` or `search` query (including the irrational queries) multi-times. The interactions generate real transcripts and can be observed by the adver ary. On the contrary, in the ideal one, same queries as in the real game can be issued by the adversary $\mathcal{A}$, and a simulator takes $\mathcal{L}$ as input to forge the corresponding transcripts for the adversary. If the adversary is unable to distinguish the real game from the ideal game, the robust DSSE is said to be adaptively secure. The formal definition is as follows.

Definition 2 ($\mathcal{L}$-adaptive-security of A Robust DSSE): For a robust DSSE scheme Σ, if for any adversary $\mathcal{A}$, we can construct an efficient simulator $\mathcal{S}$ (with the input $\mathcal{L}$) having that $|Pr[REAL_{\mathcal{A}}(\lambda) = 1] - Pr[IDEAL_{\mathcal{A},\mathcal{S}}(\lambda) = 1]|$ is negligible, where $REAL_{\mathcal{A}}(\lambda)$ and $IDEAL_{\mathcal{A},\mathcal{S}}(\lambda)$ are as follows:

- $REAL_{A}(\lambda)$*: In the real game, the implementation of DSSE protocols is exactly the same as in the real world. Arbitrary `update` or `search` queries (including the irrational queries) can be issued by the adversary $\mathcal{A}$. $\mathcal{A}$ observes the transcripts of protocols' execution and finally outputs a bit $b \in \{0, 1\}$;*
- $IDEAL_{A,S}(\lambda)$*: Like the real game, the adversary $\mathcal{A}$ issues the same `update` or `search` queries. With the input of $\mathcal{L}$, the simulator $\mathcal{S}$ simulates the transcript of protocols' execution. At the end, the adversary $\mathcal{A}$ outputs a bit $b \in$* $\{0, 1\}$.

In the *forward-and-Type-I$^-$-backward* security, the leakage $\mathcal{L}$ in the above definition must be less than an expected value. Hence, we define some basic leakage functions in the following content at first. Let $\mathcal{Q}$ denote the list of all issued `search` queries with the form (t, w), where t denotes the timestamp of a `search` query, and w denotes the searched keyword. Let $\mathcal{U}$ denote the list of all issued `update` queries with the form $(t, op, (w, \mathcal{F}))$, where t denotes the timestamp of a `update` query, $op \in \{\texttt{add}, \texttt{delete}\}$, $(w, \mathcal{F})$ denotes the pair of updated keyword and the modified file identifiers in the `update` query. Some basic leakage functions are defined as follows:

- $\Delta_{srch}(w) = \{t \mid (t, w) \in \mathcal{Q}\}$: The search pattern leakage function inputs a searched keyword (denoted as w). It outputs the timestamps of the historical `search` queries of w;
- $\Delta_{rst}(w) = \{\mathcal{F}' \mid \forall (t, op, (w, \mathcal{F})) \in \mathcal{U}, \mathcal{F}'$ consists of the non-deleted file identifiers in $\mathcal{F}\}$: The result pattern leakage function outputs the non-deleted file identifiers matching a given keyword w in current;
- $\Delta_{Time}(w) = \{t \mid (t, op, (w, \mathcal{F})) \in \mathcal{U}\}$: This leakage function outputs the inserted time (denoted as t) of all the historical `add` and `delete` queries associated with a given keyword w.

With the basic leakage functions defined above, we can define the *forward-and-Type-I$^-$-backward* security of a robust DSSE scheme as follows. Note that because the forward-and-Type-I$^-$-backward security leaks quite little information, the leakage functions defined below are quite similar to those defined for `FB-DSSE`. But we emphasize that they have different essence. Because our leakage functions are defined over the assumption that the client may issue irrational `update` queries, while those of `FB-DSSE` are defined with the opposite assumption.

Definition 3 (The Forward-and-Type-I$^-$-Backward Security): For a robust and $\mathcal{L}$-adaptively-secure DSSE scheme Σ, iff *its `search` and `update` leakage functions L^{Update} and L^{Search} can be written as*

$$\mathcal{L}^{Update}(op, (w, \mathcal{F})) = \mathcal{L}'(op),$$
$$\mathcal{L}^{Search}(w) = \mathcal{L}''(\Delta_{srch}(w), \Delta_{rst}(w), \Delta_{Time}(w))$$

where both $\mathcal{L}'$ and $\mathcal{L}''$ are stateless, we say that Σ is forward-and-Type-I$^-$-backward secure.

III. The Bi-Bitmap Index

The bitmap index was used in `FB-DSSE` to represent the file identifiers that the client wants to update. Suppose the system can support up to n files, then the binary size of the bitmap index is also n, and each bit of the bitmap index denotes a file. Let the least significant bit of the bitmap index denote file f_1, and the i-th bit of the bitmap index denote file f_i. To add (or delete) a keyword w and the associated files $\mathcal{F}$, the `FB-DSSE` client sets the corresponding bits of the bitmap index to be "1" according to $\mathcal{F}$, encrypts w and the assigned bitmap index, and uploads the generated ciphertext to the server. When the client hopes to search keyword w and a related query is issued, the server receives the search trapdoor, retrieves corresponding ciphertexts and aggregates them into one. The aggregation of those matching ciphertexts means doing the binary addition on the bitmap indexes that are contained in those matching ciphertexts.

For example, suppose $n = 6$, and the client has added entries $(w, \mathcal{F} = \{f_4, f_2\})$ and $(w, \mathcal{F} = \{f_5, f_3\})$ to the server successively. Suppose that the client now issues a `search` query for w. It generates w's search trapdoor and sends it to the server. For the server, it has to retrieve two matching ciphertexts and make the aggregation. Figure 1 shows the bitmap indexes contained in those two ciphertexts and the resulted bitmap index contained in the aggregated ciphertext. Now, suppose to delete entry $(w, \mathcal{F} = \{f_2\})$, then the client uploads a new `FB-DSSE` ciphertext containing the bitmap index "000010" to the server. When searching the keyword w again, the aggregated ciphertext contains the bitmap index "011100". It means that files $\{f_5, f_4, f_3\}$ are still valid and matching the keyword w. Obviously, `FB-DSSE` can keep search correctness if all `update` queries are rational; otherwise, it cannot. In the prior example, if the client adds file f_3 repeatedly and then searches the keyword w, the resulted bitmap index contained in the aggregated ciphertext is "011000". It causes a mistake that the file f_3 is removed.

To achieve the *robustness*, we propose the bi-bitmap index to represent files and construct a particular boolean circuit to guarantee that the aggregated bi-bitmap index can keep search correctness even if the client's `update` queries are

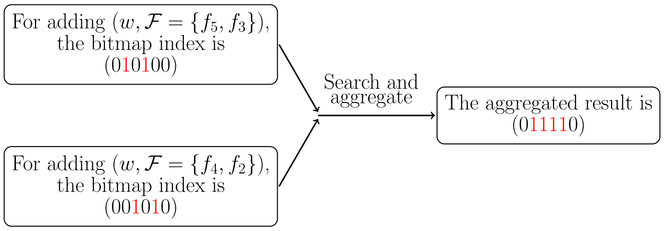


Fig. 1. An example about the bitmap index in FB-DSSE.

TABLE II
THE TRUTH TABLE FOR KEEPING THE *Robustness*

$bs_c[i]$ \ $(bs_a[i], bs_b[i])$	$(0,0)$	$(0,1)$	$(1,1)$	$(1,0)$
0	0	0	1	0
1	1	1	1	0

irrational. The bi-bitmap index consists of two bitmap indexes. The first bitmap index denotes files, and the second bitmap index denotes the operations of files. When adding a file, the corresponding bits in the first and second bitmap indexes will be set to "1". When deleting a file, the corresponding bits in the first and second bitmap indexes will be set to "1" and "0", respectively. To update (`add` or `delete`) an entry, the generated ciphertext contains a bi-bitmap index. If the client generates a `search` query of the inputted keyword, the server repeats to find out a new matching ciphertext and aggregates it with the last aggregated ciphertext until all matching ciphertexts are found. Note that the aggregated ciphertext contains a bitmap index, not a bi-bitmap index. Let bs_c denote the bitmap index contained in the last aggregated ciphertext (the initial value of bs_c is all zero), and (bs_a, bs_b) denote the bi-bitmap index contained in a found matching ciphertext. The essence of aggregating the matching ciphertext and the last aggregated ciphertext is to compute the boolean circuit

$$bs_c[i] = (\overline{bs_a[i]} \wedge bs_c[i]) \oplus (bs_a[i] \wedge bs_b[i])$$

for each bit, where $bs[i]$ denotes the i-th bit in the bitmap index bs.

Here, we show why the above boolean circuit can keep the *robustness*. Recall that a robust DSSE must keep search consistency even if there are irrational `update` queries. Without loss of generality, for a file f_i, the above boolean circuit must satisfy the following conditions:

- Case 1: $bs_c[i] = 0$, namely the file f_i has been removed or never be added. In this case, to keep search consistency, we have $bs_c[i] = 1$ only if $bs_a[i] = 1$ and $bs_b[i] = 1$; otherwise, we still have $bs_c[i] = 0$.
- Case 2: $bs_c[i] = 1$, namely the file f_i has been added and is still valid. In this case, to keep search consistency, we have $bs_c[i] = 0$ only if $bs_a[i] = 1$ and $bs_b[i] = 0$; otherwise, we still have $bs_c[i] = 1$.

According to the above conditions, we have a truth table shown as Table II and construct the following boolean circuit to satisfy those conditions by Karnaugh map reduction [12].

$$\begin{aligned} bs_c[i] &= (\overline{bs_a[i]} \wedge bs_b[i] \wedge bs_c[i]) \oplus (\overline{bs_a[i]} \wedge \overline{bs_b[i]} \wedge \overline{bs_c[i]}) \\ &\quad \oplus (bs_a[i] \wedge bs_b[i] \wedge \overline{bs_c[i]}) \oplus (bs_a[i] \wedge bs_b[i] \wedge bs_c[i]) \\ &= (\overline{bs_a[i]} \wedge bs_c[i]) \oplus (bs_a[i] \wedge bs_b[i]) \end{aligned} \quad (1)$$

Hence, the above boolean circuit on the bi-bitmap index can help guarantee *robustness*.

IV. SR-DSSE$_a$: OUR FIRST DSSE SCHEME

This section gives the construction of the first DSSE scheme SR-DSSE$_a$. The server is allowed to aggregate the corresponding ciphertexts it retrieved, such that the bi-bitmap indexes contained in those ciphertexts are aggregated according to the above boolean circuit. Since the aggregation of ciphertexts is a kind of homomorphic boolean computation, we employ TFHE to achieve such operations.

A. TFHE Review

TFHE was proposed by Chilloti et al. in 2016 [11]. The security foundation of TFHE is the Learning With Errors (LWE) hardness assumption [13], [14]. The following content reviews the main functions of TFHE. More details can be found in [11]. The TFHE scheme $\mathcal{T}$ consists of the following four algorithms.

- $\mathcal{T}$.KeyGen(λ): With the input of a security parameter λ, this algorithm generates a secret key sk and an evaluation key pk;
- $\mathcal{T}$.Enc(sk, m): Taking sk and a message $m \in \{0, 1\}$, this algorithm generates a TFHE ciphertext C;
- $\mathcal{T}$.Dec(sk, C): This algorithm takes sk as input. With an assigned TFHE ciphertext C, the algorithm decrypts C and outputs a confined message $m \in \{0, 1\}$;
- $\mathcal{T}$.Eval(`gate`, pk, C_1, C_2): Given a logical gate `gate` $\in$ {`AND`, `XOR`, `NOT`} and two TFHE ciphertexts C_1 and C_2, with input of pk, the algorithm generates a new TFHE ciphertext C', such that after decryption, the plaintext (denoted as m') has that $m' =$ `gate`(m_1, m_2), where m_1 and m_2 are the messages contained in C_1 and C_2, respectively. Note that the inputted C_2 is empty if `gate` = `NOT`.

Compared with other FHE schemes, TFHE supports logical operations, like `XOR`, `NOT`, and `AND`. TFHE also has the fastest bootstrapping to the best of our knowledge [15]. Hence, it has a good tool for us to construct SR-DSSE$_a$.

B. Some Basic Functions

Here, we construct some basic functions, such as $\mathcal{B}$.Enc, $\mathcal{B}$.Dec, and $\mathcal{B}$.Eval. They will be employed in SR-DSSE$_a$. Function $\mathcal{B}$.Enc aims to encrypt a given bitmap index, where each bit in the given bitmap index is encrypted by TFHE independently. Function $\mathcal{B}$.Dec is the corresponding decryption function of $\mathcal{B}$.Enc. Function $\mathcal{B}$.Eval takes an encrypted bi-bitmap index as input and aggregates it with an encrypted bitmap index by TFHE. And the aggregated result satisfies

Algorithm 1 Functions $\mathcal{B}$.Enc, $\mathcal{B}$.Dec, and $\mathcal{B}$.Eval

$\mathcal{B}$.Enc(sk, bs, n)

Take a TFHE secret key sk and a bitmap index bs with size n as inputs;
Initialize an empty vector $\mathcal{V}$ with size n;
for $i \leftarrow 1$ to n **do**
  Compute $\mathcal{V}[i] \leftarrow \mathcal{T}$.Enc$(sk, bs[i])$;
end for
return $\mathcal{V}$;

$\mathcal{B}$.Dec$(sk, \mathcal{V}_c, n)$

Take a TFHE secret key sk and a vector $\mathcal{V}_c$ with size n as inputs, where each element of $\mathcal{V}_c$ is a TFHE ciphertext;
Initialize an empty bitmap index bs with size n;
for $i \leftarrow 1$ to n **do**
  Compute $bs[i] \leftarrow \mathcal{T}$.Dec$(sk, \mathcal{V}_c[i])$;
end for
return bs;

$\mathcal{B}$.Eval$(pk, (\mathcal{V}_a, \mathcal{V}_b), \mathcal{V}_c, n)$

Take an TFHE evaluation key pk, the bi-bitmap index ciphertext $(\mathcal{V}_a, \mathcal{V}_b)$, and the ciphertext $\mathcal{V}_c$ of a bitmap index as inputs, where $\mathcal{V}_a$, $\mathcal{V}_b$, and $\mathcal{V}_c$ have the same size n;
Initialize an empty and temporary vector $\mathcal{V}_t$ with size n;
for $i \leftarrow 1$ to n **do**
  Compute $\mathcal{V}_t[i] \leftarrow \mathcal{T}$.Eval$(\texttt{NOT}, pk, \mathcal{V}_a[i])$;
  Compute $\mathcal{V}_c[i] \leftarrow \mathcal{T}$.Eval$(\texttt{AND}, pk, \mathcal{V}_t[i], \mathcal{V}_c[i])$;
  Compute $\mathcal{V}_t[i] \leftarrow \mathcal{T}$.Eval$(\texttt{AND}, pk, \mathcal{V}_a[i], \mathcal{V}_b[i])$;
  Compute $\mathcal{V}_c[i] \leftarrow \mathcal{T}$.Eval$(\texttt{XOR}, pk, \mathcal{V}_c[i], \mathcal{V}_t[i])$;
end for
return $\mathcal{V}_c$;

that particular boolean circuit introduced in Section III. Let n be the binary size of a bitmap index. Algorithm 1 gives the details of those functions.

C. Construction

With the above functions, we construct our first robust DSSE scheme $\texttt{SR-DSSE}_a$ in Algorithm 2. To update an entry $(w, \mathcal{F})$, the client of $\texttt{SR-DSSE}_a$ transforms this entry to a bi-bitmap index according to the rules introduced in Section III, encrypts the index by function $\mathcal{B}$.Enc, and sends the generated ciphertext to the server for storage. Then, if a `search` query containing the inputted keyword w is issued, $\texttt{SR-DSSE}_a$'s server retrieves all corresponding ciphertexts with a search trapdoor from the client, aggregates those ciphertexts into one ciphertext by function $\mathcal{B}$.Eval, and return the resulted ciphertext to the client. In the end, the client makes the decryption of received ciphertexts by function $\mathcal{B}$.Dec and obtains the matching-and-still-valid files. More explanations are as follows.

In protocol $\texttt{SR-DSSE}_a$.Setup, the client initializes some hash functions, a pseudo-random function, acceptable TFHE keys, a secret key, and some data structures to store the client's states and the server's states, respectively. Particularly, the client encrypts an all-zero bitmap index by function $\mathcal{V}_0 \leftarrow \mathcal{B}$.Enc$(sk, bs, n)$ (sk denotes the initialized secret key of TFHE, $bs = 0^n$, and n denotes the maximum n files the system supports). The generated $\mathcal{V}_0$ will be used by the server as the original state to aggregate the matching ciphertexts when searching a keyword.

In protocol $\texttt{SR-DSSE}_a$.Update, the client transforms the chosen update type (`add` or `delete`) and the updated entry $(w, \mathcal{F})$ into a bi-bitmap index, encrypts the resulted bi-bitmap index by function $\mathcal{B}$.Enc, and generates a searchable ciphertext of keyword w. All those ciphertexts are sent to the server. Finally, the client updates his local states. These states will be used to generate the corresponding keyword search trapdoor if the client performs a search after the previous `update` query.

In $\texttt{SR-DSSE}_a$.Search, a keyword search trapdoor for the corresponding `search` query is generated via the client's secret key and current state. With this keyword search trapdoor, the server is able to retrieve corresponding ciphertexts, which can be categorized into two types: one is for adding some files, and the other one is for deleting some files. Then, the server aggregates all found ciphertexts into one ciphertext. During the aggregation process, the deleted files will be really removed, and only the valid files will be contained in the resulted ciphertext. Moreover, the essence of the aggregation process is to compute the bi-bitmap indexes, that are contained in all those found ciphertexts, according to the rule defined in Equation 1. Hence, $\texttt{SR-DSSE}_a$ also guarantees the *robustness* of DSSE.

D. Correctness and Security Analysis

Correctness: $\texttt{SR-DSSE}_a$'s correctness depends on the fact that hash functions H_1 and H_2 are collision-resistant. Briefly speaking, upon searching an updated keyword w, the client sends the current random string R_c, the hash function key K_w and two counters c and c_0 to the server. With these parameters, the server repeats computing hash value $\mathsf{H}_1(K_w, R_i)$, obtaining the distinct indexes and computing the previous random string by computing $C_i \oplus \mathsf{H}_2(K_w, R_i)$ for i, which is decreasing successively from c to c_0. The uniqueness of hash value H_1 guarantees that all ciphertexts are indexed by distinct values. Similarly, the uniqueness of hash value H_2 guarantees that the server can compute the specified random string by XORing the hash value with the protected mask. This process is always correct. Because all counters used to generate w's ciphertexts are distinct, regardless of whether there are irrational update queries, i.e., the counter is monotone increasing. Therefore, the server can correctly find all unsearched encrypted bi-bitmaps of w from **EDB**. Similarly, the uniqueness of K_w guarantees that the server can correctly retrieve $\mathcal{V}_w$ from $\mathbf{S}_S$.

Next, the server evaluates the boolean circuit defined in $\mathcal{B}$.Eval over those retrieved bi-bitmaps. Specifically, according to Table II that $\mathcal{B}$.Eval is designed to implement, given a file f_i, an `add` update query (i.e., $(\mathcal{V}_a[i], \mathcal{V}_b[i]) = (1, 1)$) always maintains the existence of f_i (i.e., the resulting bit is always 1), and a `delete` update query (i.e., $(\mathcal{V}_a[i], \mathcal{V}_b[i]) = (1, 0)$) guarantees that f_i is deleted (i.e., the resulting bit is always 0). In the meanwhile, invalid update queries (i.e., $(\mathcal{V}_a[i], \mathcal{V}_b[i]) = (0, 0)$ or $(0, 1)$) will not change the existence state of f_i in **EDB** (i.e., the resulting bit is always $\mathcal{V}_w[i]$). Hence, whether

Algorithm 2 Scheme $\mathtt{SR\text{-}DSSE}_a$

$\mathtt{SR\text{-}DSSE}_a.\mathsf{Setup}(\lambda, n)$

Take λ and the maximum number n of files as inputs;
Choose two secure and independent hash functions H_1 and H_2 both with the form $\{0,1\}^\lambda \times \{0,1\}^\lambda \rightarrow \{0,1\}^\lambda$;
Choose a secure pseudorandom function $\mathsf{P} : \{0,1\}^\lambda \times \mathbf{W} \rightarrow \{0,1\}^\lambda$, where $\mathbf{W}$ denote the keyword space;
Generate a pair of TFHE keys $(sk, pk) \leftarrow \mathcal{T}.\mathsf{KeyGen}(\lambda)$;
Choose a random secret key $K_\Sigma \xleftarrow{\$} \{0,1\}^\lambda$;
Generate an encrypted bitmap index $\mathcal{V}_0 \leftarrow \mathcal{B}.\mathsf{Enc}(sk, bs, n)$, where $bs = 0^n$;
Initialize three empty maps $\mathbf{S}_C$, $\mathbf{S}_S$, and $\mathbf{EDB}$, where $\mathbf{S}_C$ and $\mathbf{S}_S$ are used to store the states of the client and the sever, respectively;
Store $(pk, \mathcal{V}_0, \mathbf{S}_S, \mathbf{EDB})$ in the server;
Store $(K_\Sigma, sk, \mathbf{S}_C)$ in the client privately;

$\mathtt{SR\text{-}DSSE}_a.\mathsf{Update}((K_\Sigma, sk), \mathbf{S}_C, op, (w, \mathcal{F}); \mathbf{EDB})$

Client:

Compute the secret key K_w of keyword w by running $K_w \leftarrow \mathsf{P}(K_\Sigma, w)$;
Retrieve the client state about keyword w by $(c_0, c, R_c) \leftarrow \mathbf{S}_C[w]$;
if $\mathbf{S}_C[w] = NULL$ **then**
    Set $c_0 \leftarrow 0$, $c \leftarrow -1$, and $R_c \xleftarrow{\$} \{0,1\}^\lambda$;
end if
Choose a random value $R_{c+1} \xleftarrow{\$} \{0,1\}^\lambda$;
Compute $I \leftarrow \mathsf{H}_1(K_w, R_{c+1})$ and $C \leftarrow \mathsf{H}_2(K_w, R_{c+1}) \oplus R_c$;
Set a bi-bitmap (bs_a, bs_b) according to the inputted op and $\mathcal{F}$;
Encrypt (bs_a, bs_b) by computing $\mathcal{V}_a \leftarrow \mathcal{B}.\mathsf{Enc}(sk, bs_a, n)$ and $\mathcal{V}_b \leftarrow \mathcal{B}.\mathsf{Enc}(sk, bs_b, n)$;
Send ciphertext $(I, C, (\mathcal{V}_a, \mathcal{V}_b))$ to the server;
Finally, update the client state by setting $\mathbf{S}_C[w] \leftarrow (c_0, c+1, R_{c+1})$;

Server:

Set $\mathbf{EDB}[I] \leftarrow (C, (\mathcal{V}_a, \mathcal{V}_b))$ to store the received ciphertext;

$\mathtt{SR\text{-}DSSE}_a.\mathsf{Search}((K_\Sigma, sk), w, \mathbf{S}_C; pk, \mathcal{V}_0, \mathbf{S}_S, \mathbf{EDB})$

Client:

Compute the secret key K_w of keyword w by running $K_w \leftarrow \mathsf{P}(K_\Sigma, w)$;
Retrieve the client state about keyword w by $(c_0, c, R_c) \leftarrow \mathbf{S}_C[w]$;
if $\mathbf{S}_C[w] = NULL$ **then**
    return $\perp$;
end if
Send a search trapdoor (K_w, R_c, c_0, c) to the server;
Update the client state by setting $\mathbf{S}_C[w] \leftarrow (c+1, c, R_c)$;

Server:

if $\mathbf{S}_S[K_w] = NULL$ **then**
    Set $\mathcal{V}_w \leftarrow \mathcal{V}_0$;
else
    Set $\mathcal{V}_w \leftarrow \mathbf{S}_S[K_w]$;
end if
Initialize an empty map $\mathbf{E}$;
for $i = c$ to c_0 **do**
    Compute $I \leftarrow \mathsf{H}_1(K_w, R_i)$;
    Retrieve ciphertext $(C, (\mathcal{V}_a, \mathcal{V}_b)) \leftarrow \mathbf{EDB}[I]$;
    Store the retrieved and encrypted bi-bitmap $\mathbf{E}[i - c_0] \leftarrow (\mathcal{V}_a, \mathcal{V}_b)$;
    Remove ciphertext $\mathbf{EDB}[I]$
    Set $R_{i-1} \leftarrow C \oplus \mathsf{H}_2(K_w, R_i)$;
end for
for $i = c_0$ to c **do**
    Retrieve the encrypted bi-bitmap $(\mathcal{V}_a, \mathcal{V}_b) \leftarrow \mathbf{E}[i - c_0]$;
    Compute $\mathcal{V}_w \leftarrow \mathcal{B}.\mathsf{Eval}(pk, (\mathcal{V}_a, \mathcal{V}_b), \mathcal{V}_w, n)$;
end for
Update the server state by setting $\mathbf{S}_S[K_w] \leftarrow \mathcal{V}_w$;
Send $\mathcal{V}_w$ to the client;

Client:

Decrypt the received $\mathcal{V}_w$ by running $bs_w \leftarrow \mathcal{B}.\mathsf{Dec}(sk, \mathcal{V}_w, n)$;
Parse bs_w into file identifiers $\mathcal{F}$;
return $\mathcal{F}$;

---

f_i appears in the search result only depends on the final valid update queries (i.e., $(\mathcal{V}_a[i], \mathcal{V}_b[i]) = (1,1)$ or $(1,0)$). To sum up, $\mathtt{SR\text{-}DSSE}_a$ achieves the correctness property defined in Definition 1.

Security: For security, the following theorem shows that $\mathtt{SR\text{-}DSSE}_a$ achieves the *forward-and-Type-I⁻-backward* security, which is defined in 3. The detailed proof is moved to Appendix A.

Theorem 1: Suppose that P is a secure and efficient PRF function, H_1 and H_2 are two random oracles. We say that the scheme $\mathtt{SR\text{-}DSSE}_a$ achieves robustness with the adaptive security of leakage functions $\mathcal{L}^{Setup}(\lambda, n) = (\lambda, n)$, $\mathcal{L}^{Update}(op, (w, \mathcal{F})) = \emptyset$, and $\mathcal{L}^{Search}(w) = (\Delta_{srch}(w), \Delta_{rst}(w), \Delta_{Time}(w))$.

V. SR-DSSE$_b$: Our Second DSSE Scheme

This section gives the construction of another robust DSSE scheme **SR-DSSE**$_b$, which also has the *forward-and-Type-I$^-$-backward* security but more efficient time-cost than **SR-DSSE**$_a$. The main difference **SR-DSSE**$_b$ and **SR-DSSE**$_a$ is their aggregation process when searching a keyword. In short, **SR-DSSE**$_a$ allows the server to achieve the aggregation process. But, to keep the confidentiality, the aggregation process of **SR-DSSE**$_a$ must be executed in the scenario of ciphertext. On the contrary, the aggregation process of **SR-DSSE**$_b$ is achieved by the client in the scenario of plaintext. Hence, it is clear that **SR-DSSE**$_b$ has a more efficient time-cost than **SR-DSSE**$_a$. Although **SR-DSSE**$_b$ takes more round-trips when searching a keyword, it is more suitable for

Algorithm 3 Protocols SR-DSSE$_b$'s Setup and Update

SR-DSSE$_b$.Setup(λ, n)

Take λ and the maximum number n of files as inputs;
Choose five secure and independent hash functions H_1, H_2, H_3 H_4 and H_5, among which H_1, H_2 are formed as $\{0, 1\}^\lambda \times \{0, 1\}^\lambda \rightarrow \{0, 1\}^\lambda$, H_3, H_4, H_5 are formed as $\{0, 1\}^\lambda \times \mathbb{Z} \rightarrow \{0, 1\}^\lambda$;
Choose a secure pseudorandom function $\mathsf{P}' : \{0, 1\}^\lambda \times \mathbf{W} \rightarrow \{0, 1\}^\lambda \times \{0, 1\}^\lambda$, where $\mathbf{W}$ denote the keyword space;
Initialize three empty maps $\mathbf{S}_C$, $\mathbf{S}_S$, and $\mathbf{EDB}$, where $\mathbf{S}_C$ and $\mathbf{S}_S$ are used to store the states of the client and the sever, respectively;
Store $(\mathbf{S}_S, \mathbf{EDB})$ in the server;
Store $(K_\Sigma, \mathbf{S}_C)$ in the client privately;

SR-DSSE$_b$.Update($K_\Sigma, \mathbf{S}_C, op, (w, \mathcal{F})$; $\mathbf{EDB}$)

Client:

Compute the secret keys K_w and K'_w of keyword w by running $(K_w, K'_w) \leftarrow \mathsf{P}'(K_\Sigma, w)$;
Retrieve the client state about keyword w by $(c_0, c, R_c) \leftarrow \mathbf{S}_C[w]$;
if $\mathbf{S}_C[w] = NULL$ **then**
  Set $c_0 \leftarrow 0$, $c \leftarrow -1$, and $R_c \xleftarrow{\$} \{0, 1\}^\lambda$;
end if
Choose a random value $R_{c+1} \xleftarrow{\$} \{0, 1\}^\lambda$;
Compute $I \leftarrow \mathsf{H}_1(K_w, R_{c+1})$ and $C \leftarrow \mathsf{H}_2(K_w, R_{c+1}) \oplus R_c$;
Set a bi-bitmap (bs_a, bs_b) according to the inputted op and $\mathcal{F}$;
Encrypt bs_a and bs_b by $e_a \leftarrow \mathsf{H}_3(K'_w, c+1) \oplus bs_a$ and $e_b \leftarrow \mathsf{H}_4(K'_w, c+1) \oplus bs_b$, respectively;
Send $(I, C, (e_a, e_b))$ to the server
Finally, update the client state by setting $\mathbf{S}_C[w] \leftarrow (c_0, c+1, R_{c+1})$;

Server:

Set $\mathbf{EDB}[I] \leftarrow (C, (e_a, e_b))$ to store the received ciphertext;

the application in which the less search time-cost is a key requirement.

A. Construction

When updating an entry $(w, \mathcal{F})$, the client of SR-DSSE$_b$ transforms the update type (add or delete) and the entry into a bi-bitmap index as SR-DSSE$_a$ does, encrypts the bi-bitmap index by normal encryption (here is different with SR-DSSE$_a$), and generates a searchable ciphertext of keyword w. The client generates a corresponding trapdoor upon searching a keyword w and sends it to the SR-DSSE$_b$'s server, which retrieves all matching ciphertexts by the trapdoor. These ciphertexts are returned back. Then, the client makes the decryption of all bi-bitmap indexes. Next, the plaintext bi-bitmap-indexes are aggregated into one bitmap index according to the rule of Equation 1. The resulted bi-bitmap index shows the matching-and-non-deleted files.

Algorithm 4 Protocol SR-DSSE$_b$.Search

SR-DSSE$_b$.Search(K_Σ, w, $\mathbf{S}_C$; $\mathbf{S}_S$, $\mathbf{EDB}$)

Client:

Compute the secret keys K_w and K'_w of keyword w by running $(K_w, K'_w) \leftarrow \mathsf{P}'(K_\Sigma, w)$;
Retrieve the client state about keyword w by $(c_0, c, R_c) \leftarrow \mathbf{S}_C[w]$;
if $\mathbf{S}_C[w] = NULL$ **then**
  return $\perp$;
end if
Send a search trapdoor (K_w, R_c, c_0, c) to the server;

Server:

if $\mathbf{S}_S[K_w] = NULL$ **then**
  Set $e_w \leftarrow 0^n$;
else
  Set $e_w \leftarrow \mathbf{S}_S[K_w]$;
end if
Initialize an empty map $\mathbf{E}$;
for $i = c$ to c_0 **do**
  Compute $I \leftarrow \mathsf{H}_1(K_w, R_i)$;
  Retrieve ciphertext $(C, (e_a, e_b)) \leftarrow \mathbf{EDB}[I]$;
  Store the retrieved and encrypted bi-bitmap $\mathbf{E}[i - c_0] \leftarrow (e_a, e_b)$;
  Remove ciphertext $\mathbf{EDB}[I]$;
  Set $R_{i-1} \leftarrow C \oplus \mathsf{H}_2(K_w, R_i)$;
end for
Send $e_w, \mathbf{E}$ to the client;

Client:

Initialize an bitmap $bs_w \leftarrow 0^n$ to record the matching files;
if $e_w \neq 0^n$ **then**
  Decrypt the files' states by running: $bs_w \leftarrow e_w \oplus \mathsf{H}_5(K'_w, c_0)$;
end if
for $i = c_0$ to c **do**
  Retrieve ciphertexts by running: $(e_a, e_b) \leftarrow \mathbf{E}[i - c_0]$;
  Decrypt and get bi-bitmap-index by running: $(bs_a, bs_b) \leftarrow (e_a \oplus \mathsf{H}_3(K'_w, i), e_b \oplus \mathsf{H}_4(K'_w, i))$;
  Compute the files' states in plaintext version by running: $bs_w \leftarrow (\overline{bs_a} \wedge bs_w) \oplus (bs_a \wedge bs_b)$;
end for
Update the client state by setting $\mathbf{S}_C[w] \leftarrow (c+1, c, R_c)$;
Re-encrypt the files' states by running: $e_w \leftarrow bs_w \oplus \mathsf{H}_5(K'_w, c+1)$;
Send new encrypted states e_w to the server;
Parse bs_w into file identifiers $\mathcal{F}$;
return $\mathcal{F}$;

Server:

Update the server state by setting $\mathbf{S}_S[K_w] \leftarrow e_w$;

Since the aggregation process of SR-DSSE$_b$ also satisfies Equation 1, SR-DSSE$_b$ has the *robustness*. More explanations are as follows.

In protocol $\texttt{SR-DSSE}_b$.Setup, the client initializes more hash functions than $\texttt{SR-DSSE}_b$.Setup, a pseudo-random function, a secret key, and some data structures to store the client's states and the server's states, respectively. Hash functions are implemented to encrypt the bi-bitmap index when updating an entry. It is different with protocol $\texttt{SR-DSSE}_a$.Setup that protocol $\texttt{SR-DSSE}_b$ does not need the client to encrypt an all-zero bi-bitmap index, since the aggregation process for searching a keyword is achieved by the client not the server.

The main idea of protocol $\texttt{SR-DSSE}_b$.Update is similar with protocol $\texttt{SR-DSSE}_a$.Update. Their main difference is the method to encrypt a bi-bitmap index. After transforming the chosen update type (add or delete) and the updated entry $(w, \mathcal{F})$ into a bi-bitmap index, protocol $\texttt{SR-DSSE}_b$.Update encrypts the resulted bi-bitmap index by some hash functions not function $\mathcal{B}$.Enc. This encryption method is a simple one. When searching a keyword in protocol $\texttt{SR-DSSE}_b$.Search, the client can decrypt all matching ciphertexts from the server efficiently by the simple encryption method. The details are shown in the following.

In $\texttt{SR-DSSE}_b$.Search, a search trapdoor for the searched keyword w is generated locally. The generation of the trapdoor depends on the client's secret key and the current state of the queried keyword. When the keyword search trapdoor is received by the server, it is utilized to search corresponding ciphertexts. With the encryption of last aggregated bitmap index, retrieved ciphertexts are returned to the client. Note that the last aggregated bitmap index is 0^n if it is the first time to search a keyword. Then, the client decrypts several bi-bitmap indexes and a bitmap index from all received ciphertexts and aggregate these indexes into one bitmap index according to Equation 1. The resulted bitmap index tells the client which files match the search query and are non-deleted. Finally, the bitmap index are re-encrypted as the ciphertext and stored in the server.

Scalability: In both $\texttt{SR-DSSE}_a$ and $\texttt{SR-DSSE}_b$, the length of the bi-bitmap index, which also indicates the number of maximum files, is fixed at the setup phase. One may worry that this makes the proposed schemes lack scalability to manage constantly growing large datasets. Fortunately, we can apply the following steps to extend the proposed schemes to improve their scalability:

1) Select fair parameters according to the dataset so that the length of the bi-bitmap index is not so small.
2) As the dataset grows, if the current scheme instance cannot accommodate more files, the client can then download the encrypted database from the server and use secret key to extract plaintext data. Then the client select setups a new instance of the scheme where the length of the bi-bitmap index are fixed to a larger number. Finally, the client embeds the extracted plaintext data to the new instance and uploads the newly generated encrypted database to the server. This approach is solely the technique that transfers static SSE schemes to dynamic ones [16]. In this step, all the decryption is performed on the client side. Hence there is no extra leakage except the number of distinct keywords currently in the database and the new length of the bi-bitmap index.

The above steps can effectively tackle the scalability problem of the proposed schemes, at the cost of amortized $O(|\mathbf{W}|)$ computation and communication overhead.

B. Correctness and Security Analysis

Correctness: For correctness, the way that $\texttt{SR-DSSE}_b$ finds and aggregates matching ciphertexts is essentially the same as that of $\texttt{SR-DSSE}_a$, except that $\texttt{SR-DSSE}_b$ decrypts and aggregates the matching ciphertexts on the client side. It is easy to find that $\texttt{SR-DSSE}_b$ also satisfies the correctness property defined in Definition 1. Hence, we omit the correctness proof of $\texttt{SR-DSSE}_b$ here.

Security: For security, the following theorem shows that $\texttt{SR-DSSE}_b$ achieves the *forward-and-Type-I$^-$-backward* security, which is defined in Definition 3. The detailed proof is moved to Appendix B.

Theorem 2: Suppose that P' is a secure and efficient PRF function, H_1, H_2, H_3, H_4 and H_5 are random oracles. We say that the scheme $\texttt{SR-DSSE}_b$ achieves robustness with the adaptive security of leakage functions $\mathcal{L}^{Setup}(\lambda, n) = (\lambda, n)$, $\mathcal{L}^{Update}(op, (w, \mathcal{F})) = \emptyset$, and $\mathcal{L}^{Search}(w) = (\Delta_{srch}(w), \Delta_{rst}(w), \Delta_{Time}(w))$.

VI. Experiment Analysis

In this section, we empirically evaluate $\texttt{SR-DSSE}_a$ and $\texttt{SR-DSSE}_b$ and compare their performance with FB-DSSE and $\texttt{IM-DSSE}_{\text{I+II}}$. FB-DSSE is the only state-of-the-art DSSE scheme of *forward-and-Type-I$^-$-backward* security. $\texttt{IM-DSSE}_{\text{I+II}}$ is quite performant and is selected as the baseline. All the evaluated schemes employ a bitmap-based index. In a nutshell, the baseline scheme $\texttt{IM-DSSE}_{\text{I+II}}$ outperforms $\texttt{SR-DSSE}_a$, $\texttt{SR-DSSE}_b$, and FB-DSSE, and $\texttt{SR-DSSE}_a$ achieves close client search overhead to $\texttt{IM-DSSE}_{\text{I+II}}$. $\texttt{SR-DSSE}_a$ is advantageous in saving the client's search time and communication bandwidth, and $\texttt{SR-DSSE}_b$ costs the least time to complete the search. Meanwhile, these two schemes can be accelerated with hardware-based accelerating techniques to gain higher search performance.

A. Experiment Setup

1) Hardware Platform: We perform all experiments on a workstation with an AMD 5950X processor, an NVIDIA RTX 2080Ti, 128GB RAM, and 64-bit Ubuntu 20.04 operating system.

2) Programming Environment: We implement all schemes with C++. Specifically, we use the GMP [17] big integer data structure to represent bi-bitmap-index. The storage structures $\mathbf{S}_S$, $\mathbf{S}_C$, and **EDB** are implemented with the container class unordered map provided by the C++ STL library to eliminate the extra overheads caused by disk I/O.

3) Cryptographic Primitives: We use OpenSSL library [18] to instantiate most of the cryptographic functions. For example, PRF functions P and P' are implemented by hmac-md5 and hash functions H_1, H_2 are implemented by hmac-sha family. Hash functions H_3, H_4, and H_5 are implemented by shake128 hash function.[3] Finally, we adopt

[3] We refer the source code from https://github.com/MockingHawk/shake128.

TABLE III

SELECTED KEYWORDS AND FREQUENCIES

Dataset I, n = 714				**Dataset** II, n = 840,499			
Word	Freq.	Word	Freq.	Word	Freq.	Word	Freq.
shred	2	correct	10	sauna	109	epigraph	204
filter	4	know	13	rangoon	125	ryu	228
african	5	partner	15	uncensor	140	delimit	252
novel	7	king	16	gemma	165	unravel	277
item	9	presid	20	silica	182	backpack	299

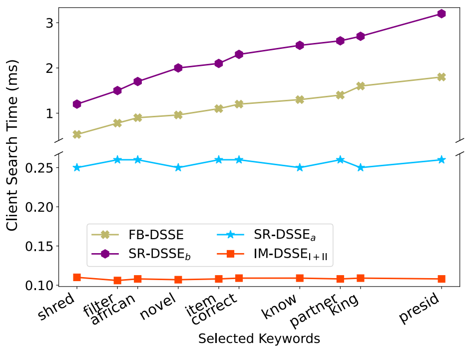


Fig. 2. Client Search time cost of SR−DSSE$_a$, SR−DSSE$_b$ and FB−DSSE.

TFHE lib [19] to implement TFHE and set its parameters as the developers recommend. We opened the source code of the evaluated schemes on Github.[4]

4) Dataset: We leverage English Wikimedia[5] as the main dataset. Specifically, we use WikiExtractor [20] to convert it into JSON documents and then extract keywords from them. Since the entire dataset is too large, we select two smaller subsets of it as our test datasets. We name those two datasets **Dataset** I and **Dataset** II. **Dataset** I contains 714 files and **Dataset** II is comprised of 840,499 files. In the following experiments, we set the number of files that the corresponding dataset contains as the maximum files the system supports, that is the length of bitmap index n. Each dataset contains 10 randomly selected keywords. Table.III shows the details of the two datasets.

5) Evaluated Metrics: Our experiments focus on the performance metrics of the search process, namely, search bandwidth and search time costs. Specifically, search bandwidth cost counts the total size of data exchanged when the client and the server execute the search protocol. The search time cost is computed by the addition of the client's token generation time, the server's search time, and the client's decryption and re-update time. We do not evaluate and report the update performance since, in practice, the search performance is more important, especially when the client manages a large-scale database.

B. Experimental Results

1) Client Search Time Cost: This experiment is performed over **Dataset** I, and the result is reported in Figure 2. The result shows that IM−DSSE$_{I+II}$ outperforms other three evaluated schemes. SR−DSSE$_a$ outperforms FB−DSSE and SR−DSSE$_b$ on the client side during the search. For example, when searching for keyword "presid", SR−DSSE$_a$ only takes the client 0.3 milliseconds, 11× and 6× faster than FB−DSSE and SR−DSSE$_b$, respectively. On the other hand, SR−DSSE$_a$ achieves the closest client search performance to other schemes. For example, the average client search time cost to find one matching file of IM−DSSE$_{I+II}$ is 0.011 milliseconds, while those of SR−DSSE$_a$, SR−DSSE$_b$, and FB−DSSE are 0.025, 0.216, and 0.115 milliseconds, respectively.

[4]https://github.com/HustSecurityLab/SR-DSSE

[5]https://dumps.wikimedia.org/enwiki/20210501/

TABLE IV

SEARCH BANDWIDTH OF SR−DSSE$_a$, SR−DSSE$_b$ AND FB−DSSE

Scheme	SR−DSSE$_a$	SR−DSSE$_b$	FB−DSSE	IM−DSSE$_{I+II}$
Search Roundtrip	1	2	1	1
Search Communication Cost	1,768KB	$(a_w + 0.5) \cdot 430$B	215B	722B

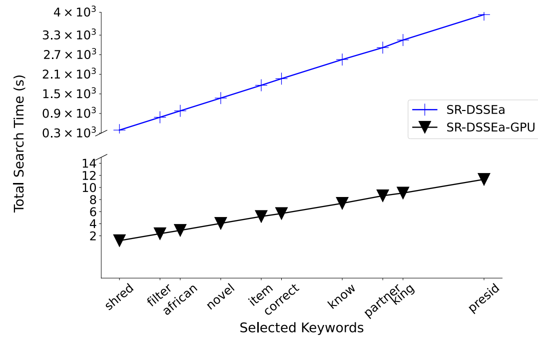


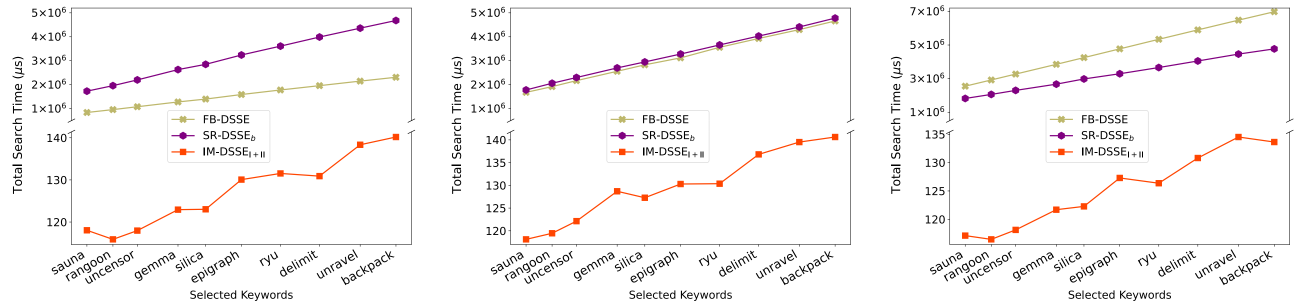
Fig. 3. Total search time cost of SR−DSSE$_a$ on CPU and GPU.

2) Search Bandwidth Cost: Table IV lists the search bandwidth costs of the evaluated schemes. SR−DSSE$_a$, IM−DSSE$_{I+II}$, and FB−DSSE achieve the optimal search roundtrip, while SR−DSSE$_b$ introduces one more search roundtrip. Although, the search roundtrip of SR−DSSE$_b$ is still constant and practical. FB−DSSE consumes the least bandwidth, namely, 215 Bytes. IM−DSSE$_{I+II}$ costs the second least bandwidth. Although SR−DSSE$_a$ consumes more bandwidth to complete the search, its cost is totally acceptable in practice (only 1,768 KB, about 1.73 MB). In terms of SR−DSSE$_b$, its search bandwidth depends on how many historical updates related to the queried keyword w (denoted as a_w) are inserted before the search query. When the historical updates of w is less than 4,210, SR−DSSE$_b$ saves bandwidth compared to SR−DSSE$_a$. Otherwise, SR−DSSE$_b$ will cost more bandwidth. Actually, the search bandwidth of SR−DSSE$_b$ is still practical and efficient. For example, suppose a_w =10,000, the total bandwidth is only about 4.1 MB. Hence, we can conclude that both SR−DSSE$_a$ and SR−DSSE$_b$ achieve practical search bandwidth performance.

3) Total Search Time Cost: SR−DSSE$_a$ is based on TFHE. Hence, it is feasible to accelerate the search process of SR−DSSE$_a$ by adopting the optimizations used in TFHE, e.g., Compute Unified Device Architecture (CUDA) [21], [22]. Hence, in this part, we evaluate and compare SR−DSSE$_a$'s search performance on CPU and GPU platforms.

Figure 3 reports the result. In the figure, ttSR-DSSE$_a$ denotes the CPU version while SR−DSSE$_a$-GPU denotes the GPU version that is implemented with CuFHE.[6] The

[6]https://github.com/vernamlab/cuFHE



(a) First Round (b) Second Round (c) Third Round

Fig. 4. Total search time cost of SR-DSSE$_b$ and FB-DSSE.

numerical results show that the GPU version achieves about 350× acceleration compared to the CPU version. For example, when searching for keyword "king", SR-DSSE$_a$ takes 3,149.4 seconds to complete the search, while SR-DSSE$_a$-GPU only needs 9.1 seconds, saving about 3,140.3 seconds. Considering that GPU has been commonly deployed in data centers nowadays, and TFHE is also actively developing [23], the total search time cost of SR-DSSE$_a$ is practical and acceptable.

To show the high efficiency of SR-DSSE$_b$, we evaluate it over **Dataset** II and compare the results with FB-DSSE. An important property of SR-DSSE$_b$ is that its search performance will increase with historical search queries. To show this property, this part of experiment contains three rounds of search. For example, the search process of keyword "sauna" can be described as: (1) in the first round, we issue 36 insertion queries and then run search, (2) in the second round, we issue another 36 insertion queries and then run search, and (3) in the last round, we execute final 37 insertion queries and then execute search. Figure 4 shows the result. IM-DSSE$_{\text{I+II}}$ keeps its advantages in performace. It is about four magnitudes faster than SR-DSSE$_b$ and FB-DSSE. With the increase of search times, the search performance of SR-DSSE$_b$ is improving. Take keyword "backpack" as an example. In the first round, SR-DSSE$_b$ needs to take 4.7 seconds to complete the search, while in the third round the time cost is 4.8 seconds. In the third round, SR-DSSE$_b$ outperforms FB-DSSE. For example, to complete the search of keyword "backpack", FB-DSSE needs 6.9 seconds, incurring extra 2.1 seconds compared to SR-DSSE$_b$. In practice, it is common for a client to search for a keyword many times. Hence, SR-DSSE$_b$ is more practical in real-world applications.

SR-DSSE$_b$ can also be accelerated via hardware-based techniques. Different from SR-DSSE$_a$, SR-DSSE$_b$ mainly leverages the CPU-based technique to accelerate, i.e., the multi-threading technique. Figure 5 shows the performance of accelerating SR-DSSE$_b$ using OpenMP[7] with the different number of threads. The results of the experiment indicate that the multi-threading technique significantly improves the search performance of SR-DSSE$_b$. For example, when searching for keyword "unravel" with 16 threads, it takes only 1.2 seconds, saving 1,092% time compared to the case using only a single thread.

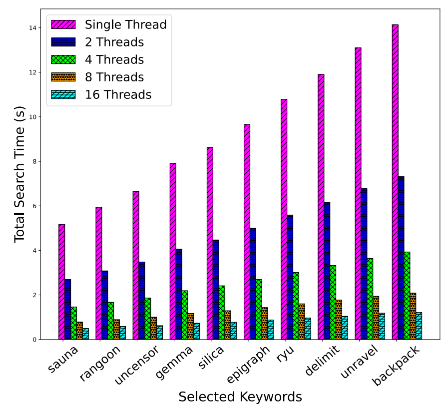


Fig. 5. Time cost with different threads.

In conclusion, the above experiments show that, although both SR-DSSE$_a$ and SR-DSSE$_b$ are inferior to IM-DSSE$_{\text{I+II}}$, considering they are robust and achieve stronger backward security, they are still practical and efficient. Specifically, SR-DSSE$_a$ has advantages in saving the client time and SR-DSSE$_b$'s whole search process is faster. Notably, with the increase of search times, SR-DSSE$_b$ achieves higher search efficiency compared to FB-DSSE.

However, compared with FB-DSSE and IM-DSSE$_{\text{I+II}}$, the proposed schemes trade the communication overhead (i.e., roundtrips or bandwidth) for robustness. Although the extra communication overhead is acceptable in practice, one may wonder whether we can eliminate it. Fortunately, with the help of the Trusted Execution Environment, like SGX, we can avoid that overhead. More concretely, we can evaluate ii-bitmap index inside the Trusted Execution Environment. There have been many DSSE works showing how the Trusted Execution Environment helps improve efficiency while maintaining high security [24], [25], [26], [27], [28]. We leave the detailed construction as an open problem to interested readers.

VII. Related Works

A. Forward and Backward Private DSSE

DSSE and its adaptive security were first formulated by Kamara et al. in 2012 [1]. Stefanov et al. gave the introduction and explaination of a series of DSSE forward and backward

[7]https://www.openmp.org/

privacy concepts in 2014 [3]. Specifically, with leakage functions, a formal definition of forward privacy was proposed and accepted. However, in fact, it is Chang and Mitzenmacher who proposed the earliest prototype of DSSE schemes trying to achieve forward privacy in 2005 [29]. In 2016, Bost constructed an optimized forward-private DSSE scheme with trapdoor permutation [30]. In the meanwhile, by the implementation of TWORAM [31], Garg et al. succeeded to give a forward-private DSSE scheme. Their scheme traded much performance for security. In 2017, Xu et al. proposed a DSSE scheme combining logical and physical deletions to reduce information leakage during update phases [32].

In 2017, Bost et al. firstly proposed definitions of backward privacy with leakage functions [4]. They categorized backward privacy into three types: Type-I, II, and III, among which Type-I is the strongest and Type-III is the weakest. With these new definitions, they constructed some DSSE schemes achieving different strength of backward security. Later, Sun et al. [33], [34], Chamani et al. [7], [35], Demertzis et al. [16], and Wang and Chow [36] proposed various DSSE schemes to achieve non-interactive search, high theoretic search performance, constant client storage, and range queries etc. In 2019, Zuo et al. introduced the definition of first Type-I$^-$ backward privacy and gave the construction of a corresponding DSSE scheme [5]. However, none of the aforementioned works found or addressed the robustness problem in DSSE.

Besides the forward and backward security, there are also many DSSE works diving into higher security to eliminate harmful information leakage [37] and mitigate attacks [38], [39], [40]. Among those works, there is an important research line that leverage real-world security techniques to achieve the security goal. For example, aforementioned Trusted Execution Environment and distributed trust [41], [42], [43]. There are also other research works exploring to equip DSSE with additional properties, such as shareability [44] and post-compromise security [45].

B. Robust DSSE

In 2022, Xu et al. formally defined the robustness of DSSE [8]. In the context of robustness, a DSSE client may issue rational update queries (e.g., duplicate add queries or deletion queries of non-existent ciphertexts). A robust DSSE scheme must guarantee the desired correctness and claimed security when the client issues irrational update queries. Unfortunately, up to now, besides the scheme `ROSE` proposed by Xu et al., only `MONETA` [4] and `Bestie` [46] achieve robustness. However, those robust DSSE schemes fail to achieve Type-I$^-$ backward privacy.

VIII. Conclusion

In this work, we identify the robustness problem existing in forward-and-Type-I$^-$-backward private DSSE schemes. To solve this problem, the definition of Type-I$^-$ backward security is extended. With the new definitions, we constructed two novel robust DSSE schemes, both of which achieves the security aim of forward and Type-I$^-$ backward privacy, i.e., $\texttt{SR-DSSE}_a$ and $\texttt{SR-DSSE}_b$. The constructions of these two schemes leverage our newly proposed Bi-bitmap-index data structure and a boolean circuit evaluation method. The experimental results show that $\texttt{SR-DSSE}_a$ is client-friendly and $\texttt{SR-DSSE}_b$ has higher search performance. The experiments show that $\texttt{SR-DSSE}_a$ and $\texttt{SR-DSSE}_b$ are not as performant as $\texttt{IM-DSSE}_{\text{I+II}}$, thereby, may not be very suitable for some performance-intensive scenarios. Fortunately, their practical performance can be further improved with the hardware-based acceleration technique, which makes the proposed schemes quite suitable for managing real databases. Additionally, their robustness can tolerate irrational client update queries. Hence, we recommend $\texttt{SR-DSSE}_a$ and $\texttt{SR-DSSE}_b$ for real-world deployment.

Appendix

A. Security Proof of $\texttt{SR-DSSE}_a$

Proof: In the security proof of $\texttt{SR-DSSE}_a$, we build of a simulator $\mathcal{S}$ with the input of the protocols' leakage. That are, $\mathcal{L}^{Setup}(\lambda, n) = (\lambda, n)$, $\mathcal{L}^{Update}(op, (w, \mathcal{F})) = \emptyset$, and $\mathcal{L}^{Search}(w) = (\Delta_{srch}(w), \Delta_{rst}(w), \Delta_{Time}(w))$. Then, $\mathcal{S}$ can simulate $\texttt{SR-DSSE}_a$'s three protocols respectively. We will prove that the ideal $\texttt{SR-DSSE}_a$ is indistinguishable from the real one under the adaptive attack and describe the simulator in Algorithm 5. Concretely speaking, the simulator $\mathcal{S}$ contains the following three phases.

Setup Phase: The simulator $\mathcal{S}$ takes the function $\mathcal{L}^{Setup}(\lambda, n) = (\lambda, n)$ as inputs and initializes three maps **RandomStrList**, **CipherList**, and **EDB**. **EDB** is sent to the server as the real game does, and the client keeps the other two maps as the internal states. **RandomStrList** records each `update`'s random string. **CipherList** records ciphertexts generated by $\mathcal{S}$. Clearly, it is hard for the adversary $\mathcal{A}$ to distinguish the simulated *Setup* phase and the real one.

Update Phase: When the adversary $\mathcal{A}$ issues an `update` query with the input of $op, (w, \mathcal{F})$, the simulator $\mathcal{S}$ takes the leakage function $\mathcal{L}^{Update}(op, (w, \mathcal{F}))$ as the input, computes a timestamp u, picks some randomly chosen string R, index I, a protected mask C, and a bi-bitmap index (bs_a, bs_b), and encrypts this bi-bitmap index. According to the randomness of oracles H_1 and H_2 and the security of $\mathcal{T}$, the simulated $(R, I, C, (\mathcal{V}_a, \mathcal{V}_b))$ have the same distribution as the real one generated by $\texttt{SR-DSSE}_a$.Update in the RO model. Hence, it is hard for the adversary $\mathcal{A}$ to distinguish the simulated *Update* phase and the real one.

Search Phase: When the adversary $\mathcal{A}$ issues a search query with the input of keyword w, the simulator $\mathcal{S}$ takes the function $\mathcal{L}^{Search}(w) = (\Delta_{srch}(w), \Delta_{rst}(w), \Delta_{Time}(w))$ as the input. To begin with, it checks the historical update queries about w and aborts if there is no updates previously (refer to Step 4). Next, the simulator must program the two random oracles H_1 and H_2, so that the computations of search trapdoors are valid in the view of the adversary $\mathcal{A}$ (refer to Steps 5 to 11). The core work is to guarantee that all simulated ciphertexts of keyword w can be retrieved by the server with a randomly generated search trapdoor. Hence, from the latest to the earliest query (refer to Step 6), the simulator $\mathcal{S}$ programs H_1 and

Algorithm 5 Simulator of Ideal $\texttt{SR-DSSE}_a$

Setup($\mathcal{L}^{Setup}(\lambda, n)$)

1: Initialize three empty map structures: **EDB**, **RandomStrList**, **CipherList**. Send **EDB** to the server and keep others locally;
2: Initialize a timestamp parameter $u \leftarrow -1$;

Update($\mathcal{L}^{Update}(op, (w, \mathcal{F}))$)

1: Add one time to the total timestamp by $u \leftarrow u + 1$;
2: Randomly generate the string $R \xleftarrow{\$} \{0, 1\}^\lambda$, the index $I \xleftarrow{\$} \{0, 1\}^\lambda$ and the protected mask $C \xleftarrow{\$} \{0, 1\}^\lambda$;
3: Randomly generate a bi-bitmap index (bs_a, bs_b) and encrypt it into the ciphertext $(\mathcal{V}_a, \mathcal{V}_b)$;
4: Record R by **RandomStrList**$[u] \leftarrow R$; Record I, C and $(\mathcal{V}_a, \mathcal{V}_b)$ by **CipherList**$[u] \leftarrow (I, C, (\mathcal{V}_a, \mathcal{V}_b))$;
5: Send I, C and $(\mathcal{V}_a, \mathcal{V}_b)$ to the server for saving;

Search($\mathcal{L}^{Search}(w) = (\Delta_{srch}(w), \Delta_{rst}(w), \Delta_{Time}(w))$)

1: Add one time to the total timestamp by $u \leftarrow u + 1$;
2: Obtain the timestamp u_s of the last search query from $\Delta_{srch}(w)$, where $u_s = -1$ if $\texttt{sp(w)} = \emptyset$;
3: Obtain all timestamps $u_{s+1}, \ldots, u_t$ between u_s and u from $\texttt{Time(w)}$, where $u_i < u_j$ if $i < j$;
4: Abort if $t = -1$ and $u_s = -1$ (that is, there are no historical update queries for keyword w);
5: Choose a key $K_w \xleftarrow{\$} \{0, 1\}^\lambda$ for the searched keyword w;
6: **for** $i = t$ to $s + 1$ **do**
7: Retrieve the simulated ciphertext $(I_{u_i}, C_{u_i}, (\mathcal{V}_a, \mathcal{V}_b)) \leftarrow$ **CiphertextList**$[u_i]$;
8: Retrieve two consecutive random strings $R_{u_i} \leftarrow$ **RandomStrList**$[u_i]$, $R_{u_{i-1}} \leftarrow$ **RandomStrList**$[u_{i-1}]$;
9: Program oracle H_1 such that $\mathsf{H}_1(K_w, R_{u_i}) = I_{u_i}$;
10: Program oracle H_2 such that $\mathsf{H}_2(K_w, R_{u_i}) = C_{u_i} \oplus R_{u_{i-1}}$;
11: **end for**
12: Send a search trapdoor (K_w, R_t, s, t) to the server
13: **return** the file identifiers contained in $\Delta_{rst}(w)$ when the client receives the server's response

H_2 according to the real $\texttt{SR-DSSE}_a$.Update. In the end, a randomly generated search trapdoor is sent to the server. Hence, it is hard for the adversary $\mathcal{A}$ to distinguish the simulated *Search* phase and the real one.

To summarize, we can construct a simulator $\mathcal{S}$ to simulate $\texttt{SR-DSSE}_a$ with the given leakage functions. And the simulated $\texttt{SR-DSSE}_a$ is indistinguishable from the real one. Thus, Theorem 1 is true. □

B. Security Proof of $\texttt{SR-DSSE}_b$

Proof: In the security proof of $\texttt{SR-DSSE}_b$, we build a simulator $\mathcal{S}$ with the input of the protocols' leakage. That are $\mathcal{L}^{Setup}(\lambda, n) = (\lambda, n)$, $\mathcal{L}^{Update}(op, (w, \mathcal{F})) = \emptyset$, and $\mathcal{L}^{Search}(w) = (\Delta_{srch}(w), \Delta_{rst}(w), \Delta_{Time}(w))$. Then, the simulator $\mathcal{S}$ can simulate $\texttt{SR-DSSE}_b$'s three protocols, respectively. We will prove that the ideal $\texttt{SR-DSSE}_b$ is

Algorithm 6 Simulator of Ideal $\texttt{SR-DSSE}_b$

Setup($\mathcal{L}^{Setup}(\lambda, n)$)

1: Initialize four empty map structures: **EDB**, **RandomStrList**, **CipherList**, **BiKeyList**. Send **EDB** to the server and keep others locally;
2: Initialize a timestamp parameter $u \leftarrow -1$;

Update($\mathcal{L}^{Update}(op, (w, \mathcal{F}))$)

1: Add one time to the total timestamp by $u \leftarrow u + 1$;
2: Randomly generate the string $R \xleftarrow{\$} \{0, 1\}^\lambda$, the index $I \xleftarrow{\$} \{0, 1\}^\lambda$ and the protected mask $C \xleftarrow{\$} \{0, 1\}^\lambda$;
3: Randomly choose two keys $sk_a \xleftarrow{\$} \{0, 1\}^\lambda$ and $sk_b \xleftarrow{\$} \{0, 1\}^\lambda$ and a bi-bitmap index (bs_a, bs_b); Encrypt the chosen bi-bitmap index into the ciphertext by $(e_a, e_b) \leftarrow (bs_a \oplus sk_a, bs_b \oplus sk_b)$;
4: Record R by **RandomStrList**$[u] \leftarrow R$; Record I, C and (e_a, e_b) by **CipherList**$[u] \leftarrow (I, C, (e_a, e_b))$; Record (sk_a, sk_b) by **BiKeyList**$[u] \leftarrow (sk_a, sk_b)$;
5: Send I, C and (e_a, e_b) to the server for saving;

Search($\mathcal{L}^{Search}(w) = (\Delta_{srch}(w), \Delta_{rst}(w), \Delta_{Time}(w))$)

1: Accumulate the timestamp parameter by $u \leftarrow u + 1$;
2: Obtain the timestamp u_s of the last search query from $\Delta_{srch}(w)$, where $u_s = -1$ if $\texttt{sp(w)} = \emptyset$;
3: Obtain all timestamps $u_{s+1}, \ldots, u_t$ between u_s and u from $\Delta_{Time}(w)$, where $u_i < u_j$ if $i < j$;
4: Abort if $t = -1$ and $u_s = -1$ (that is, there are no historical update queries for keyword w);
5: Randomly choose two keys $K_w \xleftarrow{\$} \{0, 1\}^\lambda$, $K'_w \xleftarrow{\$} \{0, 1\}^\lambda$ for the searched keyword w;
6: **for** $i = t$ to $s + 1$ **do**
7: Retrieve the simulated ciphertext $(I_{u_i}, C_{u_i}, (\mathcal{V}_a, \mathcal{V}_b)) \leftarrow$ **CiphertextList**$[u_i]$;
8: Retrieve two consecutive random strings $R_{u_i} \leftarrow$ **RandomStrList**$[u_i]$, $R_{u_{i-1}} \leftarrow$ **RandomStrList**$[u_{i-1}]$;
9: Retrieve two keys $(sk_a, sk_b) \leftarrow$ **BiKeyList**$[u_i]$
10: Program oracle H_1 such that $\mathsf{H}_1(K_w, R_{u_i}) = I_{u_i}$;
11: Program oracle H_2 such that $\mathsf{H}_2(K_w, R_{u_i}) = C_{u_i} \oplus R_{u_{i-1}}$;
12: Program oracle H_3 such that $\mathsf{H}_3(K'_w, i) = sk_a$;
13: Program oracle H_4 such that $\mathsf{H}_4(K'_w, i) = sk_b$;
14: **end for**
15: Randomly choose a key $sk \xleftarrow{\$} \{0, 1\}^\lambda$ and program oracle H_5 such that $\mathsf{H}_5(K'_w, t) = sk$ if $s < t$ (namely, there are update queries between the two search queries);
16: Send a search trapdoor (K_w, R_t, s, t) to the server
17: **return** the file identifiers contained in $\Delta_{rst}(w)$ when the client receives the server's response

%endmulticols

indistinguishable from the real one under the adaptive attack and describe the simulator in Algorithm 6. It is similar to the security proof of $\texttt{SR-DSSE}_a$ that the simulator $\mathcal{S}$ contains the following three phases, and we omit the duplicate details in the description.

Setup Phase: The simulator $\mathcal{S}$ takes the leakage function $\mathcal{L}^{Setup}(\lambda, n) = (\lambda, n)$ as the input. The simulator $\mathcal{S}$

additionally initializes a map **BiKeyList** for recording the bi-bitmap-index encryption keys and keeps the map as one of the internal states. Clearly, it is hard for the adversary $\mathcal{A}$ to distinguish the simulated *Setup* phase and the real one.

Update Phase: When an update query with the input of $op, (w, \mathcal{F})$ is issued, the simulator $\mathcal{S}$ takes the leakage function $\mathcal{L}^{Update}(op, (w, \mathcal{F}))$ as the input. Besides picking a randomly generated trapdoor and ciphertexts, the simulator $\mathcal{S}$ also randomly picks two randomly chosen keys sk_a and sk_b and records them into **BiKeyList**$[u]$. In the same way, the distribution of simulated $(R, I, C, (e_a, e_b))$ is the same as the real one, which is generated by SR-DSSE$_b$.Update in the scenario of the RO model. Therefore, it is hard for the adversary $\mathcal{A}$ to distinguish the simulated *Update* phase and the real one..

Search Phase: When a search query with the input of keyword w is issued, the simulator $\mathcal{S}$ takes the leakage function $\mathcal{L}^{Search}(w) = (\Delta_{srch}(w), \Delta_{rst}(w), \Delta_{Time}(w))$ as the input. Before programming the oracles, the simulator $\mathcal{S}$ chooses two random keys K_w and K'_w (refer to Step 5). Then, during the programming, the simulator $\mathcal{S}$ programs oracles H_3 and H_4 with the input of K'_w (refer to Steps 12 to 13). In addition, if there are some update queries between the two search queries, a random key sk is generated and programmed to oracle H_5 for re-encrypting the new result (refer to Step 15). Hence, it is hard for the adversary $\mathcal{A}$ to distinguish the simulated *Search* phase and the real one.

In summary, with the input of the given leakage functions, we are able to give the construction of a simulator $\mathcal{S}$ to simulate SR-DSSE$_b$. And the simulated SR-DSSE$_b$ is indistinguishable from the real one. Thus, Theorem 2 is true. □

ACKNOWLEDGMENT

The authors would like to express their grateful appreciation to the associate editor and the anonymous reviewers for their valuable efforts in greatly improving this article.

REFERENCES

[1] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. ACM Conf. Comput. Commun. Secur.*, Oct. 2012, pp. 965–976.
[2] S. Lu, J. Zheng, Z. Cao, Y. Wang, and C. Gu, "A survey on cryptographic techniques for protecting big data security: Present and forthcoming," *Sci. China Inf. Sci.*, vol. 65, no. 10, pp. 1–34, Oct. 2022.
[3] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–15.
[4] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1465–1482.
[5] C. Zuo, S. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *Proc. ESORICS*. Cham, Switzerland: Springer, 2019, pp. 283–303.
[6] T. Hoang, A. A. Yavuz, and J. Guajardo, "A secure searchable encryption framework for privacy-critical cloud storage services," *IEEE Trans. Services Comput.*, vol. 14, no. 6, pp. 1675–1689, Nov. 2021.
[7] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili, "New constructions for forward and backward private symmetric searchable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 1038–1055.
[8] P. Xu et al., "ROSE: Robust searchable encryption with forward and backward security," *IEEE Trans. Inf. Forensics Security*, vol. 17, pp. 1115–1130, 2022.
[9] C. Zuo, S. Sun, J. K. Liu, J. Shao, J. Pieprzyk, and G. Wei, "Forward and backward private dynamic searchable symmetric encryption for conjunctive queries," *IACR Cryptol. ePrint Arch.*, p. 1357, Oct. 2020.
[10] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, J. Pieprzyk, and L. Xu, "Forward and backward private DSSE for range queries," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 1, pp. 328–338, Jan. 2022.
[11] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Proc. ASIACRYPT*. Berlin, Germany: Springer, 2016, pp. 3–33.
[12] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Trans. Amer. Inst. Electr. Eng., I, Commun. Electron.*, vol. 72, no. 5, pp. 593–599, Nov. 1953.
[13] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *J. ACM*, vol. 56, no. 6, pp. 1–40, Sep. 2009.
[14] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, pp. 1–35, Nov. 2013.
[15] K. Matsuoka, R. Banno, N. Matsumoto, T. Sato, and S. Bian, "Virtual secure platform: A five-stage pipeline processor over TFHE," in *Proc. USENIX Secur.*, 2021, pp. 4007–4024.
[16] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–18.
[17] Free Software Foundation. *The GNU MP Bignum Library*. Accessed: Jun. 11, 2021. [Online]. Available: https://gmplib.org/
[18] Open Software Foundation. *OpenSSL*. Accessed: Jun. 11, 2021. [Online]. Available: https://www.openssl.org/
[19] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. (Aug. 2016). *TFHE: Fast Fully Homomorphic Encryption Library*. [Online]. Available: https://tfhe.github.io/tfhe/
[20] G. Attardi. (2015). *Wikiextractor*. [Online]. Available: https://github.com/attardi/wikiextractor
[21] W. Wang, Z. Chen, and X. Huang, "Accelerating leveled fully homomorphic encryption using GPU," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Jun. 2014, pp. 2800–2803.
[22] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *Proc. BalkanCryptSec*. Cham, Switzerland: Springer, 2015, pp. 169–186.
[23] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE," in *Proc. ASIACRYPT*. Cham, Switzerland: Springer, 2017, pp. 377–408.
[24] G. Amjad, S. Kamara, and T. Moataz, "Forward and backward private searchable encryption with SGX," in *Proc. 12th Eur. Workshop Syst. Secur.*, Mar. 2019, pp. 4:1–4:6.
[25] V. Vo, S. Lai, X. Yuan, S. Nepal, and J. K. Liu, "Towards efficient and strong backward private searchable encryption with secure enclaves," in *Proc. ACNS*. Cham, Switzerland: Springer, 2021, pp. 50–75.
[26] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz, "Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset," *Proc. Privacy Enhancing Technol.*, vol. 2019, no. 1, pp. 172–191, Jan. 2019.
[27] T. Hoang, R. Behnia, Y. Jang, and A. A. Yavuz, "MOSE: Practical multi-user oblivious storage via secure enclaves," in *Proc. 10th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2020, pp. 17–28.
[28] Y. Huang et al., "Cetus: An efficient symmetric searchable encryption against file-injection attack with SGX," *Sci. China Inf. Sci.*, vol. 64, no. 8, pp. 1–18, Aug. 2021.
[29] Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Proc. ACNS*. Berlin, Germany: Springer, 2005, pp. 442–455.
[30] R. Bost, "$\sum o\varphi o\varsigma$: Forward secure searchable encryption," in *Proc. ACM CCS*, 2016, pp. 1143–1154.
[31] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption," in *Proc. CRYPTO*. Berlin, Germany: Springer, 2016, pp. 563–592.
[32] P. Xu, S. Liang, W. Wang, W. Susilo, Q. Wu, and H. Jin, "Dynamic searchable symmetric encryption with physical deletion and small leakage," in *Proc. ACISP*. Cham, Switzerland: Springer, 2017, pp. 207–226.
[33] S.-F. Sun et al., "Practical backward-secure searchable encryption from symmetric puncturable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 763–780.
[34] S.-F. Sun et al., "Practical non-interactive searchable encryption with forward and backward privacy," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.

[35] J. G. Chamani, D. Papadopoulos, M. Karbasforushan, and I. Demertzis, "Dynamic searchable encryption with optimal search in the presence of deletions," in *Proc. USENIX Secur.*, 2022, pp. 2425–2442.

[36] J. Wang and S. S. M. Chow, "Forward and backward-secure range-searchable symmetric encryption," *Proc. Privacy Enhancing Technol.*, vol. 2022, no. 1, pp. 28–48, Jan. 2022.

[37] E. M. Kornaropoulos, N. Moyer, C. Papamanthou, and A. Psomas, "Leakage inversion: Towards quantifying privacy in searchable encryption," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 1829–1842.

[38] S. Oya and F. Kerschbaum, "Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption," in *Proc. USENIX Secur.*, 2021, pp. 127–142.

[39] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia, "Response-hiding encrypted ranges: Revisiting security via parametrized leakage-abuse attacks," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 1502–1519.

[40] X. Zhang, W. Wang, P. Xu, L. T. Yang, and K. Liang, "High recovery with fewer injections: Practical binary volumetric injection attacks against dynamic searchable encryption," in *Proc. USENIX Secur.*, 2023, pp. 5953–5970.

[41] T. Hoang, A. A. Yavuz, F. B. Durak, and J. Guajardo, "Oblivious dynamic searchable encryption on distributed cloud systems," in *Proc. IFIP Annu. Conf. Data Appl. Secur. Privacy*, 2018, pp. 113–130.

[42] T. Hoang, A. A. Yavuz, F. B. Durak, and J. Guajardo, "A multi-server oblivious dynamic searchable encryption framework," *J. Comput. Secur.*, vol. 27, no. 6, pp. 649–676, Oct. 2019.

[43] E. Dauterman, E. Feng, E. Luo, R. A. Popa, and I. Stoica, "DORY: An encrypted search system with distributed trust," in *Proc. USENIX OSDI*, 2020, pp. 1101–1119.

[44] W. Wang, D. Liu, P. Xu, L. T. Yang, and K. Liang, "Keyword search shareable encryption for fast and secure data replication," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 5537–5552, 2023.

[45] T. Chen et al., "The power of bamboo: On the post-compromise security for searchable symmetric encryption," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–18.

[46] T. Chen, P. Xu, W. Wang, Y. Zheng, W. Susilo, and H. Jin, "Bestie: Very practical searchable encryption with forward and backward security," in *Proc. ESORICS*. Cham, Switzerland: Springer, 2021, pp. 3–23.

Haochen Dou received the B.E. degree in information security from Xidian University, Xi'an, China, in 2020. He is currently pursuing the master's degree in cyberspace security with the Huazhong University of Science and Technology. His research interests include applied cryptography and post-quantum cryptography.

Zhenwu Dan received the B.S. degree in mathematics and applied mathematics from the Huazhong University of Science and Technology, Wuhan, China, in 2020, where he is currently pursuing the master's degree in cyberspace security. His research interests include cryptography and searchable encryption.

Peng Xu (Member, IEEE) received the Ph.D. degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2010. He was a Post-Doctoral Researcher with the Huazhong University of Science and Technology from 2010 to 2013. He was an Associate Research Fellow with the University of Wollongong, Australia, from 2018 to 2019. He is currently a Full Professor with the Huazhong University of Science and Technology. He has authored over 30 research articles, 19 patents, and two books. He was the PI of 20 grants, including three NSF projects. His research interests include the field of cryptography.

Wei Wang (Member, IEEE) received the B.E. and Ph.D. degrees in electronic and communication engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2006 and 2011, respectively. Currently, she was a Researcher with the Cyber-Physical-Social Systems Laboratory, Huazhong University of Science and Technology. She has authored more than 20 papers in international journals and conferences. Her research interests include cloud security, network coding, and multimedia transmission.

Shuning Xu received the B.E. degree in information security from Zhengzhou University, China, in 2021. She is currently pursuing the master's degree in cyberspace security with the Huazhong University of Science and Technology. Her research interests include encrypted search and cryptography.

Tianyang Chen received the B.E. degree in information security from the Huazhong University of Science and Technology, Wuhan, China, in 2017, where he is currently pursuing the Ph.D. degree in cyberspace security. His research interests include cryptography and the IoT.

Hai Jin (Fellow, IEEE) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology in 1994. He received the German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz, Germany, in 1996. He worked at The University of Hong Kong from 1998 to 2000. He was a Visiting Scholar with the University of Southern California from 1999 to 2000. He received the Excellent Youth Award from the National Science Foundation of China in 2001. He is a Cheung Kung Scholars Chair Professor of computer science and engineering with the Huazhong University of Science and Technology. He has coauthored 22 books and published over 800 research articles. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a fellow of CCF and a member of ACM.