IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 26 October 2023, accepted 14 December 2023, date of publication 19 December 2023,
date of current version 29 December 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3345024

RESEARCH ARTICLE

Shedding Light on Blind Spot of Backward Privacy in Dynamic Searchable Symmetric Encryption

HYUNDO YOON[1], MUNCHEON YU[1], CHAEWON KWAK[2], CHANGHEE HAHN[3], DONGYOUNG KOO[4], AND JUNBEOM HUR[1], (Member, IEEE)

[1]Department of Computer Science and Engineering, Korea University, Seoul 02841, South Korea
[2]Department of Computer Science, Dongduk Women's University, Seoul 02748, South Korea
[3]Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul 01811, South Korea
[4]Department of Convergence Security, Hansung University, Seoul 02876, South Korea

Corresponding authors: Junbeom Hur (jbhur@korea.ac.kr) and Changhee Hahn (chahn@seoultech.ac.kr)

This work was supported in part by the Military Crypto Research Center funded by Defense Acquisition Program Administration (DAPA) and Agency for Defense Development (ADD) under Grant UD210027XD.

ABSTRACT Dynamic searchable symmetric encryption (DSSE) enables users to outsource their data while retaining the capability to search and update on the encrypted database. Although various DSSE schemes have been proposed to achieve higher efficiency and stronger security, many of them incurred information leakages due to the linkability between ciphertexts and queries as side information. The notions of forward and backward privacy are defined to capture such information leakage in DSSE formally. In particular, backward privacy guarantees that queries do not reveal their relationship with the deleted database, which is further classified into four types (Type-I, I$^-$, II, and III) based on the types of information leakage. In this study, we provide a backward privacy attack that exploits the information leakages and apply it to Type-I$^-$ backward private schemes to lower their security level to Type-III. We then propose a new DSSE framework, which is robust against the proposed attack. We apply our framework to the previous DSSE scheme (Zuo et al., ESORICS 2019) to build the first forward and backward Type-I private DSSE scheme under the backward privacy attack, and demonstrate its efficacy.

INDEX TERMS Dynamic searchable symmetric encryption, forward privacy, backward privacy, information leakage.

I. INTRODUCTION

Searchable symmetric encryption (SSE) is a type of symmetric encryption technique that allows users to search for encrypted data. In the majority of SSE implementations, the client or data owner encrypts the database with a secret key, creates search indices, and sends the database with search indices to the server. The client then can use a search query token associated with the keyword to search encrypted documents matching the keyword without decrypting them. After SSE was first introduced in [1], many SSE studies have introduced multiple lines of works on supporting search operations over static data. In order to prevent the untrusted server from gaining any knowledge of the contents of documents and queries, cryptographic primitives with strong security properties, such as oblivious RAM (ORAM) [2] and fully homomorphic encryption [3], were adopted to SSE schemes. Unfortunately, due to unacceptable computational overhead of ORAMs [4], [5] and fully homomorphic encryption for satisfying the security requirements, many subsequent studies on SSE have focused on improving efficiency at the expense of some information leakage using well-defined leakage functions [6].

Based on the static SSE schemes supporting simple search, several dynamic SSE (DSSE) schemes have also been proposed recently to support dynamic update of keywords related to outsourced data [7], [8], [9]. However, conventional DSSE schemes incur linkability information leakages between queries and data to the server. For example, the server may be able to know that some documents to

The associate editor coordinating the review of this manuscript and approving it for publication was S. K. Hafizul Islam.

be inserted currently contains keywords that have already been searched before, or the previously deleted document contains keywords that are being searched. These leakages may allow the server to recover the contents of the database or the queries. Thus, minimizing information leakage while achieving high efficiency becomes one of the most important problems in the DSSE literature.

Two kinds of formal privacy definitions have been proposed: forward privacy and backward privacy. Forward privacy [7], [10] guarantees that newly updated data entries cannot be related to the previous search queries. On the other hand, backward privacy guarantees that queries do not reveal their relationship to the deleted data. Three formal definitions of backward privacy (Type-I, II, and III) were introduced in [11] based on its information leakage types. Recently, Zuo et al. [12] formalized a new type of backward privacy, Type-I$^-$, and presented FB-DSSE satisfying the Type-I$^-$ backward privacy. (Formal definitions of different types of backward privacy are defined in Section-II-E.)

In this paper, we propose a *backward privacy attack*, which can disclose deletion history of the target scheme. We show this leakage can be exploited to downgrade a target scheme of the backward privacy Type-I$^-$ to Type-III. The attack utilizes identifiers and timestamps associated with a specific keyword in the database leaked during the update and search process. The list of document identifiers matching the search keyword observed at the server side, namely the result pattern leakage, is trivially revealed if the client retrieves the outsourced documents [13]. On the other hand, the timestamps of update operations are leaked in schemes that apply counter-based techniques to generate update addresses for achieving indistinguishability from the server. To the best of our knowledge, our attack is the first backward privacy attack which discloses the deletion history by abusing the update timestamp history and search result on keyword w.

We then propose a new framework that can be applied to the existing forward-private DSSE schemes to hide update timestamps, thereby achieving resilience against the exploitation mentioned above. Note that our framework can only be applied to schemes that utilize the counter for update operations, which is a well-known method to achieve forward privacy [11], [12], [14]. Our framework utilizes two techniques: the bitmap index and dummy updates. Specifically, we apply the proposed framework to FB-DSSE [12] as an example of instantiation for other counter-based schemes, and use a client-side cache for performance optimization. Through a rigorous security analysis, we show the proposed scheme guarantees forward and Type-I backward privacy, which is resilient against backward privacy attack.

Our contributions are summarized as follows:

- We present a *backward privacy attack*, with a generic attack scenario targeting previous backward-private DSSE schemes.
- We define a framework, which builds an update history hiding scheme from any forward private scheme.
- We implement our framework on FB-DSSE with the utilization of caching technique, and demonstrate our scheme is resilient to our attack and achieves Type-I backward privacy by hiding the update history.

The rest of the paper is organized as follows. In Section II, we provide cryptographic backgrounds. In Section III, we introduce our backward privacy attack. In Section IV and V, we propose our scheme and analyze its security. In Section VI, we conduct a performance analysis of our scheme. In Section VII, we provide a brief overview of the prior works. Finally, in Section VIII, we provide a conclusion and discuss the future work.

II. CRYPTOGRAPHIC BACKGROUND

We introduce the notions and the cryptographic background required for our work. We denote a security parameter by $\lambda \in \mathbf{N}$, and the concatenation by $||$. By $\nu(\lambda)$ we denote a negligible function in λ. PPT stands for probabilistic polynomial time. $P(x; y)$ represents a protocol running between the client and server with inputs x and y, respectively.

A. PSEUDORANDOM FUNCTIONS

Let $Gen(1^\lambda) \in \{0,1\}^\lambda$ be a key generation function, and $G : \{0,1\}^\lambda \times \{0,1\}^l \to \{0,1\}^{l'}$ be a pseudorandom function (PRF) family. G is secure PRF if for all PPT adversaries $\mathcal{A}$, $|Pr[K \leftarrow Gen(1^\lambda); \mathcal{A}^{G_K(\cdot)}(1^\lambda) = 1] - Pr[\mathcal{A}^{R(\cdot)}(1^\lambda) = 1]| \leq \nu(\lambda)$, where $R : \{0,1\}^l \to \{0,1\}^{l'}$ is a truly random function. Henceforth, $G_K(x)$ is represented as $G(K, x)$ in this paper.

B. DYNAMIC SYMMETRIC SEARCHABLE ENCRYPTION

A database DB consists of pairs of document identifiers and a keyword w. We write a pair as $(id_i, w) \in DB$ if and only if the document with identifier id_i contains the keyword w. Let W denote the set of all keywords that appear in DB, N denote the number of document/keyword pairs, and $DB(w)$ denote the set of documents that contain keyword w.

A DSSE scheme Σ =*(Setup, Search, Update)* consists of algorithm *Setup*, and protocols *Search*, *Update* between a client and a server:

- $(\sigma, EDB) \leftarrow Setup(1^\lambda, DB)$: This algorithm runs by the data owner or the client. For security parameter λ and a database DB, the algorithm outputs (σ, EDB), where σ is the client's local state, and EDB is an (empty) encrypted database that is sent to the server.
- $(DB(w); \cdot) \leftarrow Search(w, \sigma; EDB)$: For a client's local state σ, the client runs a protocol in order to search for documents containing a certain keyword w on the encrypted database. In this paper, we only consider search queries for a single keyword. At the end of the protocol, the client outputs a set of document identifiers $DB(w)$(empty if $w \notin W$).
- $(\sigma'; EDB') \leftarrow Update(op, in, \sigma; EDB)$: For a client's local state, the operation $op \in \{add, del\}$, and a set of input $in = (id, w)$ pairs, the client runs protocol for inserting entries to or removing entries from the

database. Finally, the server and the client retrieve the updated encrypted database EDB' and the updated local state σ', respectively.

The information that is exposed to an adversarial server during the execution of the protocol is captured by the leakage function $\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Srch}, \mathcal{L}^{Updt})$, which parameterized the confidentiality of the DSSE scheme. The leakage during setup, search, and updates is represented by $\mathcal{L}^{Stp}$, $\mathcal{L}^{Srch}$, and $\mathcal{L}^{Updt}$, respectively. A secure DSSE scheme should reveal nothing about the database DB other than these leakages.

According to the definition of [12], security of DSSE is modeled by the interaction with the Real and Ideal world, called *DSSEREAL* and *DSSEIDEAL*, respectively. *DSSEREAL* behaves exactly the same as the original DSSE. On the contrary, *DSSEIDEAL* reflects the behavior of a simulator $\mathcal{S}$, which takes the leakage function $\mathcal{L}$ of the original DSSE as input.

If an adversary $\mathcal{A}$ is able to distinguish *DSSEREAL* from *DSSEIDEAL* with a negligible advantage, the information leakage is limited to $\mathcal{L}$ only. We consider the next security game more formally. Adversary $\mathcal{A}$ interacts with one of the two worlds, *DSSEREAL* or *DSSEIDEAL*, and would like to figure out which one it is.

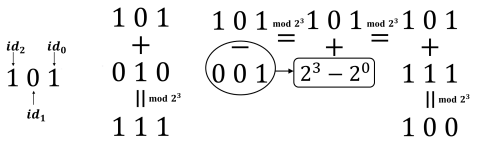
- $DSSEREAL_{\mathcal{A}}(\lambda)$: First, the adversary gets EDB by running the $Setup(1^{\lambda}, DB)$ protocol. $\mathcal{A}$ performs search queries q (or update queries (op, in)). Eventually, $\mathcal{A}$ outputs a bit b, where $b \in \{0, 1\}$.
- $DSSEIDEAL_{\mathcal{A},\mathcal{S}}(\lambda)$: Simulator $\mathcal{S}$ with the input $\mathcal{L}^{Stp}$ is executed. For search queries q (or update queries (op, in)) generated by adversary $\mathcal{A}$, the simulator $\mathcal{S}$ replies by executing the leakage function $\mathcal{L}^{Srch}(q)$(or $\mathcal{L}^{Updt}(op, in)$). Eventually, $\mathcal{A}$ outputs a bit b, where $b \in \{0, 1\}$.

Definition 1: A DSSE scheme Σ is adaptive-secure, *if*, for any PPT adversary $\mathcal{A}$ issuing a polynomial number of queries q, there exists a stateful PPT simulator $\mathcal{S}$ such that

$$|\Pr[DSSEREAL_{\mathcal{A}}(\lambda) = 1] - \Pr[DSSEIDEAL_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq negl(\lambda).$$

C. BITMAP INDEX

Bitmap index, a kind of data structure, has been widely used in the database community to represent document identifiers [15]. In detail, there is a bit-string bs of length ℓ, where ℓ is the maximum number of documents defined in a scheme. The i-th bit of bs is 1 if the i-th document exists in the database, and 0 otherwise. For example (see Fig. 1(a)), the bit-string 101 (when $\ell = 3$) denotes that there exists id_0 and id_2 in the database, but not id_1. By the bit-string definition above, we can define the addition and the deletion for a certain document using modulo addition. If we try to add document id_1 (see Fig. 1(b)), we need to generate bit-string $2^1 = 010$ and add it to the original bit-string under modulo n. Similarly, if we need to delete document id_0 (see Fig. 1(c)), we should generate bit-string $-2^0 = -001$. Since the maximum number of documents ($= \ell$) is 3, the bit-string



(a) Bitmap index (b) Add (c) Delete

FIGURE 1. An example of bitmap index.

$-2^0 = -001$ converts to $-1 = 2^3 - 2^0 = 7 \bmod 2^3$, which is 111 in binary. That is, adding bit-string 111 to the original bit-string means deletion for id_0.

In this paper, we will use the bitmap index to represent the document identifier. For each search protocol, the result is a bit-string bs, which serves as a set of document identifiers (= DB(w)). For an update operation, a bit-string bs represents the list of document identifiers to update.

D. SIMPLE SYMMETRIC ENCRYPTION WITH HOMOMORPHIC ADDITION

A simple symmetric encryption with homomorphic addition $\Pi = (Setup, Enc, Dec, Add)$ [16] consists of the following four algorithms.

- $n \leftarrow Setup(1^{\lambda})$: For the security parameter λ, it outputs a public parameter n, where $n = 2^{\ell}$ denotes the message space and ℓ is the maximum number of documents a scheme provides.
- $c \leftarrow Enc(sk, m, n)$: For a message m, the public parameter n and a random secret key $sk(0 \leq sk < n)$, it computes a ciphertext $c = (sk + m) \bmod n$, where m is the message $(0 \leq m < n)$. Note that the secret key sk needs to be stored for each encryption, and it can only be used once.
- $m \leftarrow Dec(sk, c, n)$: For the ciphertext c, the public parameter n and the secret key $sk(0 \leq sk < n)$, it recovers the message $m = (c - sk) \bmod n$.
- $\hat{c} \leftarrow Add(c_0, c_1, n)$: For two ciphertexts c_0, c_1 and the public parameter n, it computes $\hat{c} = (c_0 + c_1) \bmod n$, where $c_0 \leftarrow Enc(sk_0, m_0, n)$, $c_1 \leftarrow Enc(sk_1, m_1, n)$, $n \leftarrow Setup(1^{\lambda})$ and $0 \leq sk_0, sk_1 < n$.

A symmetric encryption with homomorphic addition Π is perfect-secure if, for any PPT adversary $\mathcal{A}$, their advantage in the perfectly-security game is negligible or

$$\mathrm{Adv}^{PS}_{\Pi,\mathcal{A}}(\lambda) = |\Pr[\mathcal{A}(Enc(sk, m_0, n)) = 1] - \Pr[\mathcal{A}(Enc(sk, m_1, n)) = 1]| \leq negl(\lambda),$$

where $n \leftarrow Setup(1^{\lambda})$ and $0 \leq m_0, m_1 < n$.

In this paper, we adopted homomorphic addition to perform update operation on encrypted bitmap index

E. FORWARD AND BACKWARD PRIVACY

Forward and backward privacy are two security properties aiming to control information leakages in DSSE. We recapitulate the following definition from [10] and [12].

Definition 2: An $\mathcal{L}-adaptively-secure$ DSSE scheme of a single keyword is forward private *if* the update leakage function $\mathcal{L}^{Updt}$ can be written as: $\mathcal{L}^{Updt}(op, w, id) = \mathcal{L}'(op, id)$ where $\mathcal{L}'$ is a stateless function, op is insertion or deletion, and id is a document identifier.

Backward privacy intends to restrict the leakages when conducting a search on a keyword w such that the server cannot discover entries that have already been removed. Three different types of backward privacy with varied leakage patterns were proposed in [11], consisting of from Type-I, which reveals the least information, to Type-III, which reveals the most. Zuo et al. [12] additionally introduced a new type of backward privacy, namely Type-I$^-$.

Consider a list Q that has elements for each executed search query. Each element consists of pair (u, w), where u is the timestamp of the search and w is the keyword. The format for an update is $(u, op, (w, bs))$, where $op \in \{add, del\}$ and bs stand for a list of document identifiers to be updated. A search pattern [6] is a type of leakage that allows an adversary to identify identical queries and is defined as $sp(w) = \{u \mid (u, w) \in Q\}$. The repetition of search queries on the same keyword w trivially reveals the search pattern. All document identifiers that currently match w are represented by the result pattern $rp(w)$. Let **TimeDB**(w) be a function that, given w, returns a list of all timestamp/document-identifier pairs associated with the keyword w that have been added to the database but not yet been removed. That is, $TimeDB(w) = \{(u, bs) \mid (u, add, (w, bs)) \in Q$ and $\forall u', (u', del, (w, bs)) \notin Q\}$. **Updates**$(w)$ is a function that lists the timestamp u of all updates related to w or the update history related to w. Formally, $Updates(w) = \{u \mid (u, op, (w, bs)) \in Q\}$. Lastly, the function **DelHist**(w) exposes the history of deleted entries by offering the adversary all (insertion timestamp, deletion timestamp) pairs. Most importantly, it reveals which additions or deletions match which ones. Finally, $DelHist(w) = \{(u^{add}, u^{del}) \mid \exists\ bs : (u^{add}, add, (w, bs)) \in Q$ and $(u^{del}, del, (w, bs)) \in Q\}$.

Given these functions, formal notions of backward privacy can be defined with additional notation bit-string bs as follows.

Definition 3 ([11], [12]): An $\mathcal{L} - adaptively - secure$ DSSE scheme has backward privacy:

Type-I (BP-I): *iff* $\mathcal{L}^{Updt}(op, w, bs) = \mathcal{L}'(op)$, and $\mathcal{L}^{Srch}(w) = \mathcal{L}''(TimeDB(w), a_w)$.

Type-I$^-$ (BP-I$^-$): *iff* $\mathcal{L}^{Updt}(op, w, bs) = \mathcal{L}'(op)$, and $\mathcal{L}^{Srch}(w) = \mathcal{L}''(rp(w), Updates(w))$.

Type-II (BP-II): *iff* $\mathcal{L}^{Updt}(op, w, bs) = \mathcal{L}'(op, w)$, and $\mathcal{L}^{Srch}(w) = \mathcal{L}''(TimeDB(w), Updates(w))$.

Type-III (BP-III): *iff* $\mathcal{L}^{Updt}(op, w, bs) = \mathcal{L}'(op, w)$, and $\mathcal{L}^{Srch}(w) = \mathcal{L}''(TimeDB(w), DelHist(w))$. $\mathcal{L}'$ and $\mathcal{L}''$ are stateless funcitons, and a_w is the total number of updates related to w.

III. PROPOSED ATTACK

In this section, we formulate leakages in traditional DSSE schemes. We then propose our *Backward Privacy Attack*, exploiting the leakages in Type-(I$^-$, II) backward private scheme to weaken the security level of the target scheme. Finally, we classify the vulnerability of existing schemes based on the leakage profile (Table 1).

TABLE 1. Security profile of prior work.

Scheme	Backward Privacy	Leakage Profile		Vulnerability to our attack
		Updates(w)	$rp(w)$	
Moneta [11]	Type-I	✗	✓	✗
Orion [14]	Type-I	✗	✓	✗
FB-DSSE [12]	Type-I$^-$	✓	✓	✓
Fides [11]	Type-II	✓	✓	✓
Mitra [14]	Type-II	✓	✓	✓
SD_a [13]	Type-II	✓	✓	✓
SD_d [13]	Type-II	✓	✓	✓
Aura [17]	Type-II	✗	✓	✗

A. LEAKAGE FORMULATION

Our attack mainly targets two leakages, which are formulated as $rp(w)$ and **Updates**(w). Before describing our attack scenario, we first confirm that the leakage functions we exploit are actually exposed in the existing schemes and acceptable in the DSSE domain. Most DSSE schemes need to execute additional protocols to retrieve the matching documents, because their search protocols only output the matching document identifiers. By observing the additional protocols for data retrieval, the server or the attacker is able to easily figure out the result of document identifiers corresponding to the search query. This leakage is unavoidable unless the actual documents are protected by means of additional security mechanisms, e.g., ORAM [13]. Furthermore, several schemes (i.e., [10], [12], [13], [14]) deterministically generate the update token itself or its address related to the keyword with a keyword update counter during the update protocol at the client side. In the search protocol, the client should initiate a token list in accordance with counters from 1 up to the number of update operations related to the searched keyword and send it to the server. This immediately reveals when each update operation took place for the keyword, since the server will receive tokens or addresses that they have observed at the previous update operations. That is, the update history of the searched keyword (i.e., **Updates**(w)) is leaked to the server, which is captured by definition 3, especially in Type-(I$^-$, II).

B. BACKWARD PRIVACY ATTACK

We now introduce *Backward Privacy Attack*. In this attack, we show how the combination of update history leakage and result pattern leakage gives the server enough information to guess the deletion history. The details will be explained below.

1) THREAT MODEL

The attacker is semi-honest such that it can observe the interactions between the client and the server. The attacker follows the specifications of the target scheme while striving

to learn as much information as possible from the leakage.The goal of the attacker is to obtain a portion of **DelHist**(w), which is not allowed to be leaked in backward private schemes stronger than Type-III.

2) ATTACK SCENARIO

Consider Q as a list of search query history and T as a list of tokens observed at the attacker side. The element format of list T is (u, *tokens*), where u is timestamp and *tokens* is the entry sent from the client. For the same timestamp u, the attacker receives the element at T, which corresponds to the element at Q. We also denote $T(w)$ as a list of tokens related to w, but the list does not contain the information disclosing what the keyword w exactly is.

The attack consists of the following two steps. In the first step, the attacker observes the list of tokens (i.e., T) according to the list of queries. By observing the search pattern leakage, the attacker should extract a list of search history, namely T^{Src}, from the original list T. This process can also be achieved by exploiting the update timestamp leakage. Each element of the list T^{Src} has the update timestamp leakage. The update timestamp history is unique for each keyword, and the history leaked from the later search protocol is a super-set of the one from the earlier search protocol. So the attacker can divide the list T by $T(w_1), T(w_2), \ldots, T(w_n)$, where n is the number of keywords that appear in Q. Note that it does not violate the definition of forward privacy because the attacker cannot relate the newly added entries to previous search queries on the specific keyword w.

In the second step, the attacker identifies whether the type of update operation performed is insertion or deletion. The details are as follows. After every search protocol, the client needs to execute an additional protocol to retrieve the actual document document in the real-world scenario. During the retrieval, the attacker observes the result pattern. We denote the result pattern leakage in the keyword w at the time u search by $rp_u(w)$.

Between two search protocols at the time i and j, the difference between $rp_i(w)$ and $rp_j(w)$ represents the change in the document identifier set. The change of identifier set lets the attacker have sufficient information to determine the operation type of update or the information which deletion cancelled which addition. We denote by $I_{i,j}(w)$ the change of identifier set, by $I^+_{i,j}(w)$ the identifier set in $rp_j(w)$ but not included in $rp_i(w)$, and by $I^-_{i,j}(w)$ the identifier set in $rp_i(w)$ but not included in $rp_j(w)$ between i and j related to w. If there are k updates between the time i and j on a particular keyword w, the adversary can learn the following information at the period:

- The addition updates are executed at least once for each identifier in $I^+_{i,j}(w)$.
- The deletion updates are executed at least once for each identifier in $I^-_{i,j}(w)$.
- The cancellation happened if $|I_{i,j}(w)| < k$, where $|I_{i,j}(w)|$ is the number of set $I_{i,j}(w)$.

TABLE 2. Attack scenario example on FB-DSSE.

Time	Operation History	Attacker's knowledge		
		$sp(w)$	Updates(w)	$rp(w)$
0	Setup(1^λ)			
1	Search(w_1)	$\{1\}$	$\emptyset$	$\emptyset$
2	Update(add, id_1, w_1)	$I^+_{1,5}(w) = \{id_1, id_2, id_3\}, I^-_{1,5}(w) = \emptyset$		
3	Update(add, id_2, w_1)			
4	Update(add, id_3, w_1)			
5	Search(w_1)	$\{1, 5\}$	$\{2, 3, 4\}$	$\{id_1, id_2, id_3\}$
6	Update(add, id_4, w_1)	$I^+_{5,8}(w) = \emptyset, I^-_{5,8}(w) = \emptyset$		
7	Update(del, id_4, w_1)			
8	Search(w_1)	$\{1, 5, 8\}$	$\{2, 3, 4, 6, 7\}$	$\{id_1, id_2, id_3\}$

Given an $id(\in I_{i,j}(w))$, the attacker can determine the operation, the identifier, and the keyword with probability $\frac{1}{k}$ by the leakage above. By collecting such information spread out on the protocol history, the attacker can learn that which deletion canceled which addition with the probability

$$\frac{1}{k_1 \times k_2} \tag{1}$$

or

$$\frac{2}{k_3 \times (k_3 - 1)}. \tag{2}$$

The probability (1) is computed in the case where the addition for id is in the period with k_1 updates, and the deletion for id is in the period with k_2 updates. For example, if there is an executed search query list for a certain keyword w and document identifier id: $(1, search, w)$, $(2, add, (w, id))$, $(3, search, w)$, $(4, del, (w, id))$, $(5, search, w)$, the attacker can simply determine the deletion that took place at time 4 cancelled the addition at time 2 deterministically. Since Bost et al. [11] mentioned that the search query on keyword w, which was queried between the addition and the deletion for document id, will clearly disclose the deleted document id, the authors excluded the case from the definition of backward privacy [18]. However, such an exclusion does not reflect the real-world scenarios in which diverse search queries for the same keyword can be made between the addition and deletion.

The probability (2) is computed in the case where the addition and the deletion for id are in the same period which contains $k_3 (> 1)$ updates. In this case, we can see the probability is equal to 1 if k_3 is 2, implying the attacker can identify the cancellation information deterministically if there are only two updates in the period, and they are addition and deletion for the same identifier. Finally, the part of **DelHist**(w) is leaked to the attacker, resulting in the violation of Type-II or higher levels of backward privacy.

3) APPLICATION

We select FB-DSSE [12] as the target scheme, which is the forward and Type-I$^-$ backward private scheme. In the paper, the authors mentioned that FB-DSSE leaks $sp(w)$, $rp(w)$ and **Updates**(w) during the search protocol. For better understanding, we first give a brief overview of FB-DSSE.

As the first step in performing an update operation on keyword w, the client obtains an encrypted bit string e by encrypting the bit string bs, using a secret key derived from the counter value c. Note that each keyword is assigned a unique c such that c is incremented each time an update is performed on w. The next step is for the client to use the randomly generated search token ST_{c+1} in order to obtain an encrypted index UT_{c+1}. Finally, the client sends an update token $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$ to the server, and the server stores in encrypted database EDB$[UT_{c+1}] \leftarrow (e_{c+1}, C_{ST_c})$.

When the client performs an update in keyword w, the client encrypts the bit string bs to get the encrypted bit string e_{c+1}. For encrypting the bit string bs, the client uses the incremented counter value $c + 1$ for generating a secret key. Note that the counter value c is incremented for every update on keyword w. In addition, the client uses a randomly generated search token ST_{c+1} for generating an encrypted index UT_{c+1}. The client finally sends an update token $(UT_{c+1}, (e_{c+1}, C_{ST_c}))$ to the server, where C_{ST_c} refers to the masked previous search token. Finally, the server stores (e_{c+1}, C_{ST_c}) in encrypted database EDB$[UT_{c+1}]$.

For the search on keyword w, the client sends search token ST_c, and a key K_w, and counter value c specific to the search keyword w. Then, the server uses K_w and ST_i to get the encrypted index UT_i, where $i \in \{0, \ldots, c\}$. The server retrieves all of the encrypted bit string e_i by accessing EDB$[UT_i]$. In this procedure, by observing the access pattern of EDB, an adversary is able to identify when the updates on keyword w have occurred (i.e., **Updates**(w)). The server adds all of the encrypted bit string e and sends it to the client. Then, the client decrypts it and outputs the final bit string for retrieval of the matching documents. When the client retrieves the matching documents, the client needs to send the final bit string to the server.

4) ATTACK EXAMPLE

Table 2 displays an example operation history on FB-DSSE and the attacker's knowledge according to the leakage function. In the example, the attacker should learn ($sp(w)$, **Updates**(w), $rp(w)$) for each search operation. Since extracting protocol for a certain keyword can be achieved by exploiting $sp(w)$ or **Updates**(w), we consider the keyword w_1 as an example without loss of generality. By comparing the result pattern between two search protocols at time 1 and time 5, the adversary can figure out the change of the result document identifiers or $I_{1,5}(w)$ $(= I^{+}_{1,5}(w) \cup I^{-}_{1,5}(w))$. Thus, the attacker should simply know that there is at least one addition operation for each identifier in $I^{+}_{1,5}(w) = \{id_1, id_2, id_3\}$. As shown in Table 2, there are three update operations between time 1 and time 5 and the attacker knows that from **Updates**(w). The attacker successfully estimates the operation/document identifier pair in the list of updates between time 1 and time 5 at a rate of $\frac{1}{3}$. The leakage information from the search protocol at time 5 and 8 is more critical than above. The attacker knows that there are two updates on the period, but no change happened between the two results. These leakages give the attacker enough information to know that the cancellation of addition and deletion occurred. The attacker can observe that the two updates cancel each other (which is a part of **DelHist**(w)). Moreover, the disclosure of confidential information leads to the downgrade of backward privacy level in FB-DSSE from Type-I^{-} to Type-III.

C. ATTACK ON PREVIOUS SCHEMES

In Table 1, we analyze the state-of-the-art forward and backward private (Type-(I,I^{-},II)) schemes under our threat model. Note that backward privacy Type-I schemes do not leak **Updates**(w). Specifically, search operations in Moneta [11] and Orion [14] do not leak timestamps of update history by applying oblivious components in their update and search protocols. FB-DSSE [12], Fides [11], and Mitra [14] are vulnerable to our attack, since their search protocols leaks **Updates**(w) and $rp(w)$. SD$_a$ and SD$_d$ schemes [13] utilize the static result-hiding technique [19]. Even though they did not explicitly address document retrieval procedure, these schemes still need to execute an additional protocol to receive matching documents. During the retrieval protocol, these schemes inevitably reveal the result pattern to the attacker. Hence, SD$_a$ and SD$_d$ are also vulnerable to our attack. Lastly, in Aura [17], the client requests the server only for the insertion operations, whereas the deletion is conducted locally by the client. Thus, the server cannot learn the timestamp of deletions, allowing Aura to be resilient to our attack.

Countermeasures: The countermeasure to our attack is preventing the server from observing result pattern leakage or/and update history leakage. As we mentioned in Section III-A, the result pattern leakage is inescapable due to the retrieval protocol, even though they use oblivious approaches such as ORAM. On the other hand, hiding update history can be achieved using oblivious components [20], [21]. However, high computational overhead and additional search roundtrips are inevitable due to the oblivious components. Therefore, in this paper, we focus on hiding the update timestamp leakage (i.e., **Updates**(w)), by leveraging the dummy access based on the bitmap index representation, which is much more lightweight than the oblivious components.

IV. SCHEME CONSTRUCTION

In this section, we propose a framework for hiding the update history with dummy updates, and our scheme construction based on it. Specifically, we targeted the leakage information caused by the counter value, thus the proposed framework is only valid for the counter-based DSSE schemes. However, considering forward privacy is the most important security requirement of DSSE and typically achieved by counter-based approaches, our work addresses an important open problem of significant impact and has high applicability.

A. A GENERIC FRAMEWORK FOR UPDATE HISTORY HIDING SCHEME

As we mentioned in Section III-A, in the traditional forward private schemes [10], [12], [13], [14], the server examines all of the update tokens for the update operations that took place before the search query to achieve forward privacy. Thus, the update history is exposed to the server. To prevent update history leakage, we let the client send dummy update operations to the server. The details of how our framework can be applied to counter-based DSSE scheme is explained below.

Assume that the DSSE scheme Σ supports update operations with the local update counter map CT. For the updated entry (op, w, id), the update token for the server is generally derived from CT[w] during the update protocol. After that, the client increases the update counter related to w by 1. To hide the update history from the server, our framework additionally sends dummy updates with the actual update. We denote the number of dummy update operations by n_{dummy}. Keywords of the dummy operation are randomly selected from the keyword space W, and identifiers for the dummy update are also generated on the client side. To minimize the client storage cost, we set the candidate identifier space size m to be less than or equal to the maximum number of documents for updated id of the dummy update. In order to retain the correctness of search results, the client should keep tracking the information of dummy updates. Note that id can be treated as a bit-string (see Fig. 1). The bit-string related to dummy updates is computed with addition without losing its data, and is stored in the local map $\mathrm{CT}_{dummy}[w]$. At the end of the search protocol, the client removes its bit-string before retrieving the actual search result. Due to the dummy update, the server cannot know which is the actual update, leading to obfuscation of the update history in the server's view.

Even though the technique we described above can hide the update history of the counter-based approaches, extra computational and communicational overheads are unavoidably incurred as a trade-off. Furthermore, additional dummy update tokens in the server may waste the server's storage as much as the number of dummy updates. In order to handle these trade-off, we propose an improved scheme by building our framework on FB-DSSE [12] with some modifications.

B. SCHEME OVERVIEW

We construct our scheme based on FB-DSSE [12], which leaks the update history during the search protocol. Homomorphic addition in FB-DSSE can be used to address the aforementioned drawbacks, and avoid the leakage of the update type (i.e., addition or deletion) to the server. Specifically, when generating update tokens with the update counter, FB-DSSE additionally generates one more token which serves as a *linker* between the recent update address and the current update address. In the search protocol, the server can check all of the update addresses with this *linker*, summate the update tokens, and store it in the new address. In the summation step, the bit-string is computed using the homomorphic addition, leading to the reduction in the overhead caused by the dummy update. However, since it is only conducted at the end of the search protocol, it is difficult to prevent redundancy of dummy update tokens if no search operation is performed for some keywords.

To avoid this problem, we re-organize the summation step (from the search protocol to the update protocol). That is, the scheme runs the update operation in which the server finds the recent update address and directly updates its token with the homomorphic addition. As a result, the update address stored in the server is limited to one per keyword all the time, which enables the server to maintain only the limited number of update tokens. Nevertheless, the proposed construction induces another concern in that the server should determine whether the current updated address is accessed during the earlier search operation, which violates the definition of forward privacy.

To guarantee forward security, we modified some procedures of the instantiation. First, the server deletes the correlated search result after the server returns it to the client. Thus, it is impossible for the server to identify the relationship between the previous search and current update when the server performs updates for the same keyword. Second, the client temporarily stores search results in the cache, ensuring the correctness of the result of later search operations related to the same keyword. Suppose that a search query for a certain keyword w is performed. If the following request is a search operation for the same keyword, the client can just return the result from the cache without interaction with the server. If it is an update operation for the same keyword, the client should send the update token which is a sum of the current update bit-string and the past search result in the cache to the server, and remove the past result from the cache. Note that the newly updated address must be different from the removed address. To maintain constant storage overhead on the client when cache removal rarely happens, we set a cache size threshold. When the cache data size becomes larger than the threshold, the client removes results stored in the cache by forcing update of it before the search operation or sending it to the server as a dummy update of the next update.

C. SCHEME CONSTRUCTION

We describe a formal construction of our scheme. The proposed scheme takes a keyed PRF F_K with a key K and simple symmetric encryption with homomorphic addition $\Pi = (Setup, Enc, Dec, Add)$ as primitives. The algorithm of the proposed scheme is described in Algorithm 1.

1) SETUP

The algorithm is run by a client. It takes as input the security parameter λ. It then selects a secret key K and an integer n, where $n = 2^{\ell}$ and ℓ is the maximum number of documents that can be supported by the scheme. The setup algorithm

Algorithm 1 Forward and Backward Type-I Private DSSEP

Setup(1^λ):

$K \leftarrow Gen(1^\lambda), n \leftarrow \Pi.Setup(1^\lambda)$
CT, CT_{cache}, $EDB \leftarrow$ empty map
$\sigma \leftarrow (n, K, \text{CT}, \text{CT}_{cache})$
Send EDB to server

Update(w, bs, σ; EDB):

Client:

$WList \leftarrow \{w\}, TList \leftarrow \emptyset$
for $i = 1$ to n_{dummy} **do**
  Randomly select w_i in W
  $WList \leftarrow WList \cup \{w_i\}$
end for
Randomly change the order of $WList$
for $j = 0$ to n_{dummy} **do**
  $w_j \leftarrow WList[j]$
  $K_{w_j} \leftarrow F_K(w_j)$, $(cnt_{src}, cnt_{upd}, bs_{pad}) \leftarrow \text{CT}[w_j]$
  if $(cnt_{src}, cnt_{upd}, bs_{pad})$ is NULL **then**
    $cnt_{src} \leftarrow 0, cnt_{upd} \leftarrow 0\ bs_{pad} \leftarrow 0$
  end if
  $addr_j \leftarrow H_1(K_{w_j}, cnt_{src})$
  $sk_j \leftarrow H_2(K_{w_j}, cnt_{src}||cnt_{upd} + 1)$
  if w_j in CT_{cache} **then**
    $bs_j \leftarrow \text{CT}_{cache}[w_j]$ and remove $\text{CT}_{cache}[w_j]$
    if $w_j = w$ **then**
      $e_j \leftarrow \Pi.Enc(sk_j, bs_j + bs, n)$
    else
      $e_j \leftarrow \Pi.Enc(sk_j, bs_j, n)$
    end if
  else
    if $w_j = w$ **then** ▷ For the actual update
      $e_j \leftarrow \Pi.Enc(sk_j, bs, n)$
    else ▷ For dummy updates
      $bs_{dummy} \leftarrow \{0,1\}^m$
      $bs_{pad} \leftarrow bs_{pad} + bs_{dummy}$
      $e_j \leftarrow \Pi.Enc(sk_j, bs_{dummy}, n)$
    end if
  end if
  $\text{CT}[w_j] \leftarrow (cnt_{src}, cnt_{upd} + 1, bs_{pad})$
  $T_j \leftarrow (addr_j, e_j)$
  $TList \leftarrow TList \cup \{T_j\}$
end for
Send $TList$ to server

Server:

for $i = 1$ *to* $TList.size$ **do**
  $(addr_i, e_i) \leftarrow TList[i]$
  if $EDB[addr_i]$ is NULL **then**
    $EDB[addr_i] \leftarrow e_i$
  else
    $EDB[addr_i] \leftarrow \Pi.Add(EDB[addr_i], e_i, n)$
  end if
end for

Search(w, σ; EDB):

Client:

if w in CT_{cache} **then**
  return $\text{CT}_{cache}[w]$
end if
$K_w \leftarrow F_K(w)$, $(cnt_{src}, cnt_{upd}, bs_{pad}) \leftarrow \text{CT}[w]$
if $cnt_{upd} = 0$ **then**
  return $\emptyset$
end if
$addr \leftarrow H_1(K_w, cnt_{src})$
Send $addr$ to server

Server:

$Sum_e \leftarrow EDB[addr]$
Send Sum_e to client and remove $EDB[addr]$

Client:

$Sum_{sk} \leftarrow 0$
for $i = 1$ *to* cnt_{upd} **do**
  $sk_i \leftarrow H_2(K_w, cnt_{src}||i)$
  $Sum_{sk} \leftarrow Sum_{sk} + sk_i \bmod n$
end for
$bs \leftarrow \Pi.Dec(Sum_{sk}, Sum_e, n) - bs_{pad}$
$\text{CT}_{cache}[w] \leftarrow bs$, $\text{CT}[w] \leftarrow (cnt_{src} + 1, 0, 0)$
return bs

also generates three empty maps, CT, CT_{cache}, and EDB, where CT is to store the counters and dummy bit-string for each keyword in W, CT_{cache} is to store the temporary search results from the server, and EDB is to store an encrypted database. Finally, it outputs the encrypted database EDB and the local state $\sigma = (n, K, \text{CT}, \text{CT}_{cache})$. The client keeps $(K, \text{CT}, \text{CT}_{cache})$ as a secret.

2) UPDATE

When performing an update operation, the client takes as input (w, bs), where w is the keyword for the update operation and bs is the bit-string representing the updated identifiers. At first, the client initiates a keyword list $WList$, which consists of updated keywords. For each keyword in the list, the client generates a location of the update token in the server $addr$ and an update token e. The location $addr$ is computed via H_1 with K_{w_j} and the search counter cnt_{src}; and the secret key sk for homomorphic addition is computed via H_2 with K_{w_j} and $cnt_{src}||cnt_{upd} + 1$. The update token e is encrypted with the key sk by using simple symmetric encryption with homomorphic addition. This process runs for every keyword in the keyword list, but there is a slight difference depending on whether the keyword is in the cache. If the keyword is in the CT_{cache}, the bit-string for the update

operation should be the sum of the current update bit-string (0 if the keyword is for dummy update) and the past search result. The entry in the CT_{cache} should be removed for the client's storage. By contrast, if the keyword is not in the CT_{cache}, the bit-string for the update operation is bit-string bs for the current update or bs_{dummy}, which is generated on-the-fly, for dummy updates. Note that the client should store the information about the dummy update to maintain the correctness of the search result. Thus, the client add bs_{dummy} and bs_{pad} which represents the sum of bit-string for the dummy update. The bit-string bs_{pad}, which is stored locally, serves as a padding on the result bit-string for each keyword, and it should be removed before getting the correct results of the search query. We should adjust the size of bs_{pad} so that it does not give too much overhead to the client storage. For instance, if we set the length of bs_{pad} to 8 bits, the corresponding client-side storage is 0.13 MB for each keyword. It can be achieved by selecting the negative number bs_{dummy}, which is a deletion update. After that, the client renews the local map CT with changed entries, and sends the update token list to the server. Since the token list is generated from the keyword list which has random order, the server cannot know which one is for the actual update. On the server side, it accesses the address $addr$, obtains the value Sum_e, adds it with the update token e, and saves it. This process was originally conducted in the search protocol of FB-DSSE, however, we re-organized it as a procedure of the update protocol as described before. Due to the property of homomorphic encryption, adding the newly update token to the original one serves as an update operation without loss of security. Also, because each memory access is independent, this task can be processed in parallel.

3) SEARCH

For the keyword w, the client first accesses the local cache CT_{cache} to get the recent search result. If it hits, the client can just return its value and terminate the search algorithm. Otherwise, the client should get the result via the interaction with the server. The client generates K_w via PRF F_K, and gets $addr$, the address in the server, via the hash function H_1 with (K_w, cnt_{src}) corresponding to $\mathrm{CT}[w]$ at the state σ. The client then retrieves the update token Sum_e by sending $addr$ during the interaction with the server. After sending the token, the server should remove its value from the EDB. The client computes the secret key Sum_{sk} for decryption, which is generated via hash functions H_2 with $(K_w, cnt_{src}||cnt_{upd})$. The client decrypts Sum_e with Sum_{sk} using the decryption function of the simple symmetric encryption with homomorphic addition. Finally, after removing the fake identifiers for update bs_{pad}, the client gets the search result bs at the end of the search protocol while storing it on the local cache for later search queries. Since the update history observed at the server contains a number of dummy updates, it is impossible to learn the actual update history.

4) DISCUSSION

The proposed framework is designed to enhance the security level and efficiency of stateful counter-based DSSE schemes. However, for unstateful DSSE schemes, it is challenging to apply our framework. For instance, the caching technique we used in our framework cannot be easily applied to the schemes that leverages revocation, because revocation typically re-generates the secret key and re-encrypts the indices in EDB. Thus, it is very challenging to identify whether the cached value and the newly generated search token are from the same document, which is an important open problem as a future work in the secure DSSE literature.

V. SECURITY ANALYSIS

In this section, we provide a security analysis of the proposed scheme. Our analysis follows the methodology in [10] and [12]. The proposed scheme is denoted by Φ.

Theorem 1 (Adaptive Security of Scheme Φ): Let F be a secure PRF, $\Pi = (Setup, Enc, Dec, Add)$ be a perfectly secure simple symmetric encryption with homomorphic addition, and H_1, H_2 be random oracles. We define $\mathcal{L}_\Phi = (\mathcal{L}_\Phi^{Srch}, \mathcal{L}_\Phi^{Updt})$, where $\mathcal{L}_\Phi^{Srch}(w) = (rp(w), a_w)$ and $\mathcal{L}_\Phi^{Updt}(op, w, bs) = op$. Then Φ is $\mathcal{L}_\Phi$ - secure.

Proof: Here, we utilize the simulator $\mathcal{S}$ for simulating the perspective of adversary $\mathcal{A}$ who uses the leakage $\mathcal{L}_\Phi = (\mathcal{L}_\Phi^{Srch}, \mathcal{L}_\Phi^{Updt})$, which follows below step:

1) SIMULATION OF SETUP

The secret key K and an integer n are set, following the Setup in Algorithm 1. Also, three empty maps, CT, CT_{cache}, and EDB are initialized.

2) SIMULATION OF UPDATE

When $\mathcal{S}$ performs the update operation, keyword w and a bit string bs are used as input. In the simulation of update operation, the keyword list $WList$ is initiated for selecting dummy updates. For all the keywords selected, the location of update token $addr$ in the server and an update token e is generated. The counter value is used to generate the key for encrypting the update token. $\mathcal{S}$ sends the generated update tokens to $\mathcal{A}$. For addition and deletion updates, there are no differences between them, thus whether the update is addition or deletion is indistinguishable from $\mathcal{A}$'s view. Therefore, $\mathcal{L}_\Phi^{Updt}(op, w, bs) = op$.

As for the keyword w that has never been searched before, a new random key is selected and stored in a table Key when querying F to generate a key. Otherwise, the key from table Key corresponding to w is used. Consequently, the probability of distinguishing F from a truly random function by an adversary $\mathcal{B}_1$ is $\mathrm{Adv}_{F,\mathcal{B}_1}^{\mathrm{PRF}}(\lambda)$.

3) SIMULATION OF SEARCH

For a search query on keyword w, if the keyword has not been searched previously, the location of the update token $addr$ is computed with H_1. $\mathcal{A}$ outputs the last update token Sum_e for

the keyword w. Finally, $\mathcal{S}$ decrypts Sum_e to retrieve the final bit string. For retrieving the documents, the final bit string is sent to server, which leaks $rp(w)$.

For the random oracle H_2, which is used to decrypt the update token Sum_e, $\mathcal{A}$ does not know the key that was used for encryption. Even though the total number of updates performed on w (i.e., a_w) is leaked, this leakage also includes the dummy updates performed on w in the perspective of $\mathcal{A}$. Thus, the probability that $\mathcal{A}$ guesses correctly for a given keyword is $1/2^\lambda$. Suppose that $\mathcal{A}$ makes polynomial p queries. Then, the probability is $p/2^\lambda$. Moreover, when decrypting the update token, $\mathcal{A}$ needs the counter value for w, which is the leakage defined in $\mathcal{L}_\Phi^{Srch}(w)$. Therefore, the probability of distinguishing the simulated H_2 from the real execution and building a reduction $\mathcal{B}_2$ is limited to $\mathrm{Adv}^{\mathrm{PS}}_{\Pi,\mathcal{B}_2}$.

4) CONCLUSION

For the updates, the indistinguishablity is guaranteed due to the dummy updates and the selection of random string. Moreover, the use of dummy updates allows $\mathcal{A}$ to not be able to distinguish the real update from the dummy one. For the searches, if the keyword has been searched beforehand, the search result stored in the CT_{cache} is used, which prevents $\mathcal{A}$ from learning any meaningful information. Lastly, the CT_{cache} hides the search pattern $sp(w)$ by retrieving bs without interacting with the server. By integrating all simulation results, a PPT adversary $\mathcal{A}$ has the following advantage:

$$\begin{aligned}\Pr[DSSEREAL^{\Phi}_{\mathcal{A}}(\lambda)=1]-\Pr[DSSEIDEAL^{\Phi}_{\mathcal{A},\mathcal{S}}(\lambda)=1]\\ \leq \mathrm{Adv}^{\mathrm{PRF}}_{F,\mathcal{B}_1}(\lambda)+\mathrm{Adv}^{\mathrm{PS}}_{\Pi,\mathcal{B}_2}(\lambda)+p/2^\lambda.\end{aligned}$$

Therefore, the probability of distinguishing the real execution from the simulated one is negligible in λ. It also shows that the proposed scheme leaks only the information permitted by the Type-I backward privacy defined in **Definition 3**. This concludes our proof. ■

Next, we show that the proposed scheme Φ guarantees forward and Type-I backward privacy. Firstly, forward privacy is guaranteed in the scheme Φ, by letting the update token be generated with incremented counter value cnt bound to a specific keyword w. This state value cnt prevents $\mathcal{A}$ from generating a search query to retrieve newly added document containing keyword w. Next, backward privacy is guaranteed because $\mathcal{A}$ is unable to learn the update history(**Updates**(w)). Specifically, the dummy updates prevent $\mathcal{A}$ from identifying which keyword w is the target for the actual update. Furthermore, the deletion and addition operations are indistinguishable because they are processed via the same protocol.

VI. EXPERIMENTAL ANALYSIS

In this section, we analyze the proposed scheme in comparison with the state-of-the-art DSSE schemes that provide stronger backward privacy than Type-III (specifically, Orion (Type-I) [22], FB-DSSE (Type-I$^-$) [12], Aura (Type-II) [23]).

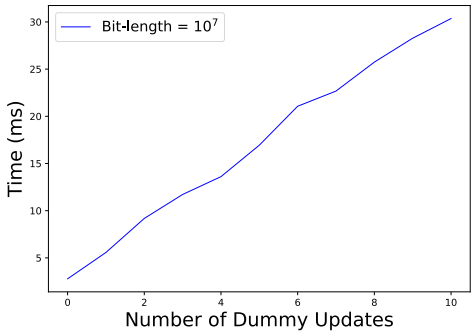


FIGURE 2. Update time according to n_{dummy}.

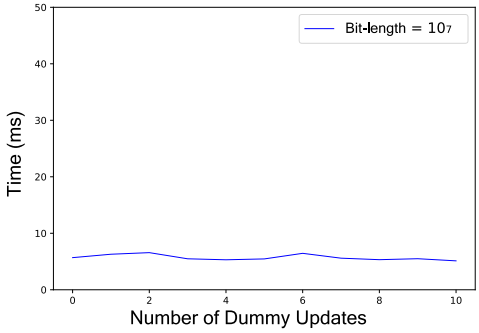


FIGURE 3. Search time according to n_{dummy}.

A. IMPLEMENTATION SETTINGS

We implement our scheme, FB-DSSE [12], Orion [22] and Aura [23] in C++ using OpenSSL [24] library to construct PRF and hash function in the system with Windows 10 operating system, Intel Core i7-10700K processor running at 3.80GHz, and 32GB of RAM. Notably, this system is utilized as both the client and the server in the evaluation. For the dummy update of our scheme, the keyword is randomly selected from the keyword space, and the length of dummy bit-string is set to 3 for the client's local storage. The time for retrieving actual documents from the search result is not measured because it is negligible.

B. PERFORMANCE EVALUATION OF OUR SCHEME

We now provide a fine evaluation of the proposed scheme. We first evaluate the computational time of our scheme according to the number of dummy update operations. We perform the update operation for each keyword 20 times on the empty database. The update time includes client token generation and server update time; the search time includes token generation, server search time, and client decryption time.

As shown in Fig. 2, the update time linearly increases to the number of dummy updates. On the other hand, the search time in our scheme remains almost constant as shown in Fig. 3, implying the number of dummy updates does not

TABLE 3. Dataset statistics.

Dataset	File	# of File	# of Keywords
Enron Mail [25]	text	500,000	12,366
COCO [26]	image	163,957	91

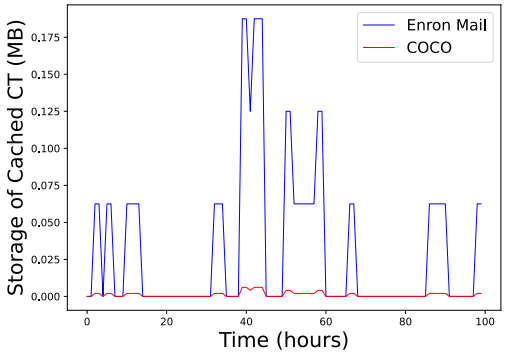


FIGURE 4. Storage overhead of CT_{cache}.

affect the search latency significantly. It is because dummy updates do not incur extra computation overhead during the search protocol. Since our search protocol needs to access only one address to get the update token, most of the search time is required for modular operations to remove the fake bit-string from the client. Note that the fake bit-string can be generated from the value which is less than or equal to the maximum number of documents supported by the scheme (three in our experiment). The removal time is reduced due to the small size of the fake bit-string. That is, the search time of our scheme is only affected by the maximum number of documents.

Next, we evaluate the storage overhead of our scheme using two datasets: Enron email and COCO image datasets, as described in Table 3. The Enron email dataset [25] comprises 500,000 text documents including 12,366 unique keywords. On the other hand, the COCO image dataset [26] consists of 163,957 image files associated with 91 unique keywords. Notably, in the COCO dataset, these keywords represent the objects depicted in the images, employed for image categorization. The Enron dataset has a size of 1.3 GB, while the COCO dataset has a significantly larger size of 25 GB. In our scheme, the bit length corresponds to the number of files, and each keyword necessitates a bitmap index for search result retrieval. Consequently, for the Enron mail dataset, the server requires 772.875 MB of storage capacity to store the necessary information for search purposes. As this storage requirement accounts for 50% of the dataset volume, the storage overhead may initially appear substantial. It is worth noting that the Enron dataset is predominantly text-based, with a considerable number of keywords contained within a single file. However, when our scheme is applied to the other formats, such as images, the ratio of dataset volume to the storage needed for the encrypted index diminishes significantly. In the case of the

TABLE 4. A comparison of storage costs among different schemes.

Scheme	Client-side Storage	Sever-side Storage
Aura [17]	$\mathcal{O}(Wd)$	$\mathcal{O}(W \log D)$
FB-DSSE [12]	$\mathcal{O}(W \log D)$	$\mathcal{O}(W \log D)$
Orion [14]	$\mathcal{O}(1)$	$\mathcal{O}(N)$
Proposed	$\mathcal{O}(W \log D)$	$\mathcal{O}(W \log D)$

*W: the total number of distinct keywords; D: the total number of documents; N: the total number of updates; d: the size of search result matching w

COCO dataset, which encompasses 91 object categories for image classification, our scheme necessitates only 1.87 MB of storage on the server. This represents less than 0.0001% of the dataset volume, indicating a considerably reduced storage overhead [25], [26].

To evaluate the storage overhead on the client-side for CT_{cache}, we conduct a simulation using a distribution model for data insertion, deletion, and keyword search operations with the Enron and COCO datasets. In our evaluation, we assume that data insertion follows a Poisson distribution with a mean duration denoted as $\tilde{\lambda}$. Similarly, deletion follows an exponential distribution with a mean duration of $\frac{1}{\mu}$, while search requests follow an exponential distribution with a mean duration of $\frac{1}{\mu'}$. Additionally, we assume that the keyword frequency of each dataset follows a Zipf distribution. By conducting this simulation, we can quantitatively analyze the storage overhead on the client-side when utilizing CT_{cache}. Figure 4 illustrates the results obtained from a simulation conducted over a duration of 100 hours, using parameter values of $\tilde{\lambda} = 3$, $\frac{1}{\mu} = 40$, and $\frac{1}{\mu'} = 1$. The horizontal axis of the figure represents time in hours, while the vertical axis represents the storage consumption of CT_{cache}. As indicated in Table 4, the proposed scheme exhibits higher storage overhead compared to Orion [14], which provides a similar level of backward privacy. Orion needs 0.098 MB for Enron Mail dataset and 0.0007 MB for COCO dataset on the client-side. However, based on the simulation results, the storage overheads of CT_{cache} are limited to a maximum of 0.187 MB for the Enron Mail dataset and 0.006 MB for the COCO dataset. It is important to note that the storage overhead varies depending on the type of operations performed, as the values stored in the cache are removed during update operations. Consequently, the additional storage overhead on the client-side can be deemed an acceptable trade-off considering the improved efficiency in search latency.

C. PERFORMANCE COMPARISON WITH FB-DSSE

We now compare our scheme with FB-DSSE. Similar to the previous experiment, we perform the update operation for each keyword and search operation for the keyword. The time for update and search operations of our scheme and FB-DSSE with different bit lengths are given in Fig. 5 and Fig. 6, respectively. The bit length means the maximum number of files supported by the scheme.

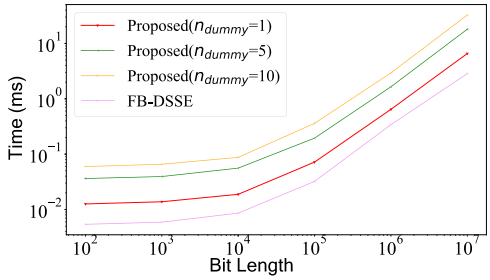


FIGURE 5. Comparison of update time with FB-DSSE.

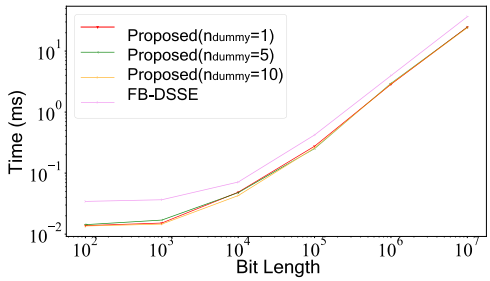


FIGURE 6. Comparison of search time with FB-DSSE.

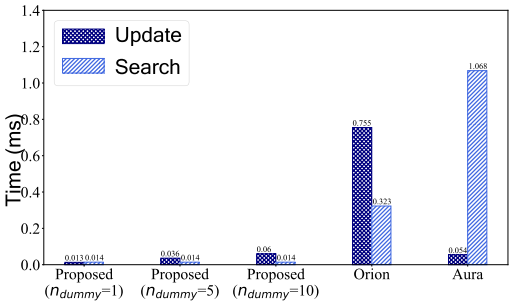


FIGURE 7. Comparison of schemes that are resilient to *backward privacy attack*.

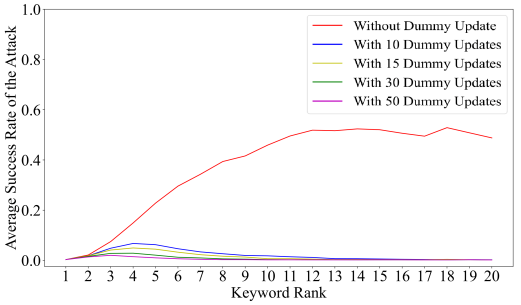


FIGURE 8. Average of the attack success rate.

1) UPDATE TIME

Fig. 5 shows the average update time for 20 update operations. Compared to FB-DSSE, our scheme executes additional operations related to the dummy update, resulting in extra computational overhead. That is, the update time of our scheme with n_{dummy} is approximately $n_{dummy} + 1$ times slower than the scheme FB-DSSE. However, note that FB-DSSE has lower backward privacy level as it leaks the update history, leading to vulnerability to our attack.

2) SEARCH TIME

Fig. 6 illustrates the comparison of search time. In the search protocol of our scheme, the server is able to directly access one address to get the update token, while FB-DSSE needs to access multiple addresses according to the update history. On the server side, the search time of our scheme is independent of the total number of update operations. Nonetheless, since the client locally stores the information of the dummy update, the client needs to operate additional modular additions for each dummy counter to calculate accurate results. Despite the need for additional computation on the client side, the entire search time of our scheme is 1.5× to 2.5× faster than FB-DSSE under the cases of different bit lengths.

D. PERFORMANCE COMPARISON WITH ATTACK RESILIENT SCHEMES

Since the main goal of our scheme is to make it resilient to *Backward Privacy Attack*, we also compared our scheme to the previous schemes that are resilient to the attack. For the comparison, we set the maximum number of files supported by the scheme as 100. In each scheme, we run insertion operations (update operation for our scheme) 20 times for each keyword, and then perform the search operation for the keyword. The comparison result is given in Fig. 7. The results illustrate that our scheme considerably outperforms the other schemes during both the search and the update operations.

1) COMPARISON WITH ORION

The search protocol of Orion [14] conceals the update history by utilizing ORAM. The update time of Orion is about 12× to 58× slower, and its search time is 23× slower than our scheme due to the computational overhead of ORAM.

2) COMPARISON WITH AURA

Aura [17] makes deletion operation oblivious to the server. As shown in the figure, the update time of Aura is close to our scheme with 10 dummy updates. However, Aura shows the largest search latency among all of the schemes, because the search protocol of Aura needs to decrypt tokens with the key which is found at GGM tree [27]. When it comes to the update time, Aura is 1.1× faster than our scheme when our scheme supports 10 dummy updates, and our scheme is about 4× faster than Aura when our scheme supports one dummy update. Moreover, the search time of our scheme is 76× faster than that of Aura regardless of the number of dummy updates.

E. EFFECTIVENESS OF RESILIENCE

We evaluate the resilience of the proposed framework to our backward privacy attack by conducting a simulation in

diverse distribution models of keyword insertion, deletion, and keyword search in the cloud storage.

According to the distribution models in [28] for file transfer and in [29] for the search, we designed a simulation by assuming that an update operation to the pair (keyword, file identifier) follows a Poisson distribution with the rate $\tilde{\lambda}$. This pair's lifespan follows an exponential distribution with a mean duration $\frac{1}{\mu}$, while the search request on the identical keyword follows an exponential distribution with a mean duration $\frac{1}{\mu'}$. We also assume that the keyword frequency follows a Zipf distribution.

Based on the distribution model, we evaluate the resilience of the proposed scheme by measuring the probability of the successful guess of the correct deletion time for each identifier by the attacker. We simulate the success rate for the top-20 most frequent keywords, when $\tilde{\lambda} = 3$, $\frac{1}{\mu} = 30$ and $\frac{1}{\mu'} = 20$. For each keyword, we measured the average success rate for each identifier during the update operations over 10,000h timespan and repeated this simulation 10 times. Fig. 8 shows the attack success rate on average in several cases. In the figure, we can see that the larger the keyword rank, i.e., a low frequency, the greater attack's success rate when dummy updates are not used. We also can observe that the attack's success rate is dramatically decreased even though deploying only 10 dummy updates.

VII. RELATED WORK

Song et al. [1] first presented an SSE scheme construction with linear-time search capability. Since then, Curtmola et al. [6] have provided a formal security definition of SSE and a sublinear-time construction. Kamara et al. [9], and Kamara and Papamanthou [8] proposed SSE schemes for sublinear-time updates in the encrypted databases.

Since the concept of forward privacy was introduced in [7], forward private DSSE schemes have been widely researched, and various works have been proposed to improve its security and efficiency [2], [10], [11]. Backward privacy was first mentioned in [30], and Bost et al. [11] have formalized the definition of backward privacy and classified it into Type-I, II, and III categories based on the amount of the leakage information. Chamani et al. [14] proposed three forward and backward private schemes, called Mitra (Type-II), Orion (Type-I), and Horus (Type-III). Demertzis et al. [13] presented three DSSE schemes with constant permanent client storage and efficient search. Zuo et al. [12] designed a scheme called FB-DSSE, which satisfies both forward privacy and Type-I$^-$ backward privacy. They use the homomorphic addition and bit-string representation in their scheme to achieve Type-I$^-$ backward privacy. Sun et al. [17] introduced a new cryptographic primitive named Symmetric Revocable Encryption, and also presented a practical and non-interactive backward privacy Type-II DSSE scheme without using the trusted environment and oblivious component.

There are other works that have been introduced recently in the DSSE literature that have considered different client-server models or different types of query from that of ours. Liu et al. [31] introduced Eurus that leverages ORAM as a core building block. Eurus enhances the efficiency and security by employing a multi-server model for constructing an ORAM-friendly protocol. Wu and Li [32] also introduced a multi-path ORAM for forward and backward private DSSE schemes, aiming to support conjunctive query. Chamani et al. [33] introduced a DSSE scheme for the multiple user setting, and provided a security definition for multi-user DSSE settings in the presence of the corrupted users. Guo et al. [34] introduced a forward private verifiable DSSE scheme for conjunctive query. Specifically, they provided a new design of verification tag and applied it to DSSE to support backward privacy.

VIII. CONCLUSION AND FUTURE WORK

In this study, we proposed a novel backward privacy attack on Dynamic Symmetric Searchable Encryption (DSSE). Our attack leverages only the data originally permissible in the backward private schemes to disclose deletion records, which are protected in the schemes that offer greater protection than Type-III backward privacy. Under our backward privacy attack, the leakage information from even the backward privacy Type-I$^-$ scheme could be exploited to weaken its security level. We also presented a framework to hide update history by utilizing dummy updates without sacrificing search correctness. Finally, we constructed a novel scheme by applying the update history hiding technique to FB-DSSE. To the best of our knowledge, we proposed the first forward and Type-I backward privacy scheme without using the oblivious component.

However, our attack only aims to stateful DSSE schemes, especially counter-based ones. Applying our method to stateless schemes is not so trivial issue. For instance, if the schemes use revocation mechanisms, our caching mechanism can hardly be applied, because the client cannot match the newly generated token with cached values. Therefore, how to extend our method into stateless DSSE schemes is one of the important future works.

ACKNOWLEDGMENT

(Hyundo Yoon and Muncheon Yu contributed equally to this work.)

REFERENCES

[1] D. Xiaoding Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. IEEE Symp. Secur. Privacy.*, May 2000, pp. 44–55.

[2] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: Efficient oblivious ram in two rounds with applications to searchable encryption," in *Proc. Annu. Int. Cryptol. Conf.* Cham, Switzerland: Springer, 2016, pp. 563–592.

[3] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," IACR Cryptol. ePrint Arch., Tech. Rep., 2012, p. 144, vol. 2012.

[4] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 1–51, Jan. 2015.
[5] M. Naveed, "The fallacy of composition of oblivious ram and searchable encryption," IACR Cryptol. ePrint Arch., Tech. Rep., 2015, p. 668, vol. 2015.
[6] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," *J. Comput. Secur.*, vol. 19, no. 5, pp. 895–934, Nov. 2011.
[7] Y.-C. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Proc. Int. Conf. Appl. Cryptography Netw. Secur.* Cham, Switzerland: Springer, 2005, pp. 442–455.
[8] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Proc. Int. Conf. Financial Cryptography Data Secur.* Cham, Switzerland: Springer, 2013, pp. 258–274.
[9] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. ACM Conf. Comput. Commun. Secur.*, Oct. 2012, pp. 965–976.
[10] R. Bost, "$\Sigma o \varphi o \varsigma$: Forward secure searchable encryption," in *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1143–1154.
[11] 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.
[12] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *Proc. Eur. Symp. Res. Comput. Secur.* Cham, Switzerland: Springer, 2019, pp. 283–303.
[13] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou, "Dynamic searchable encryption with small client storage," IACR Cryptol. ePrint Arch., Tech. Rep., 2019, p. 1227, vol. 2019.
[14] J. Ghareh 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.
[15] V. Sharma, "Bitmap index vs. B-tree index: Which and when," *Oracle Technol. Netw.*, 2005.
[16] C. Castelluccia, E. Mykletun, and G. Tsudik, "Efficient aggregation of encrypted data in wireless sensor networks," in *Proc. 2nd Annu. Int. Conf. Mobile Ubiquitous Syst., Netw. Services*, 2005, pp. 109–117.
[17] S.-F. Sun, R. Steinfeld, S. Lai, X. Yuan, A. Sakzad, J. Liu S. Nepal, and D. Gu, "Practical non-interactive searchable encryption with forward and backward privacy," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.
[18] 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.
[19] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," IACR Cryptol. ePrint Arch., Tech. Rep., 2014, p. 853, vol. 2014.
[20] D. Micciancio, "Oblivious data structures," in *Proc. 29th Annu. ACM Symp. Theory Comput.*, 1997, pp. 215–226.
[21] E. Stefanov, M. Van Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious ram protocol," *J. ACM JACM*, vol. 65, no. 4, pp. 1–26, 2018.
[22] (2020). *Ghareh Chamani*. [Online]. Available: https://github.com/jgharehchamani/SSE
[23] (2018). *MonashCybersecurityLab*. [Online]. Available: https://github.com/MonashCybersecurityLab/Aura
[24] E. A. Young, T. J. Hudson, and R. Engelschall, "OpenSSL: The open source toolkit for SSL/TLS," OpenSSL, Tech. Rep., 2011.
[25] B. Klimt and Y. Yang, "Introducing the Enron corpus," in *Proc. CEAS*, 2004, pp. 92–96.
[26] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common objects in context," in *Computer Vision—ECCV*. Zurich, Switzerland: Springer, 2014, pp. 740–755.
[27] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *J. ACM*, vol. 33, no. 4, pp. 792–807, Aug. 1986.
[28] G. Carofiglio, M. Gallo, L. Muscariello, and D. Perino, "Modeling data transfer in content-centric networking," in *Proc. 23rd Int. Teletraffic Congr. (ITC)*, Sep. 2011, pp. 111–118.
[29] T. Fenner, M. Levene, and G. Loizou, "A stochastic evolutionary model generating a mixture of exponential distributions," *Eur. Phys. J. B*, vol. 89, no. 2, pp. 1–7, Feb. 2016.
[30] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," IACR Cryptol. ePrint Arch., Tech. Rep., 2013, p. 832, vol. 2013.
[31] Z. Liu, Y. Huang, X. Song, B. Li, J. Li, Y. Yuan, and C. Dong, "Eurus: Towards an efficient searchable symmetric encryption with size pattern protection," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 3, pp. 2023–2037, May 2022.
[32] Z. Wu and R. Li, "OBI: A multi-path oblivious RAM for forward-and-backward-secure searchable encryption," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–16.
[33] J. G. Chamani, Y. Wang, D. Papadopoulos, M. Zhang, and R. Jalili, "Multi-user dynamic searchable symmetric encryption with corrupted participants," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 1, pp. 114–130, Jan. 2023.
[34] C. Guo, W. Li, X. Tang, K. R. Choo, and Y. Liu, "Forward private verifiable dynamic searchable symmetric encryption with efficient conjunctive query," *IEEE Trans. Dependable Secure Comput.*, early access, 2023.

HYUNDO YOON received the B.S. degree in computer science from Korea University, Seoul, South Korea, in 2019. He is currently with Korea University, in computer science (combined course). His research interests include information security, cloud computing security, and applied cryptography.

MUNCHEON YU received the B.S. degree from Kukmin University, Seoul, in 2021, and the M.S. degree from the Department of Computer Science and Engineering, College of Informatics, Korea University, South Korea, in 2023. His research interest includes applied cryptography.

CHAEWON KWAK is currently pursuing the B.S. degree with the Department of Computer Science, Dongduk Women's University, South Korea. Her research interest includes applied cryptography.

CHANGHEE HAHN received the B.S. and M.S. degrees in computer science from Chung-Ang University, Seoul, South Korea, in 2014 and 2016, respectively, and the Ph.D. degree from the Department of Computer Science and Engineering, College of Informatics, Korea University, South Korea, in 2020. He was with Korea University as a Postdoctoral Researcher, from 2020 to 2021. He is currently an Assistant Professor with the Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul. His research interests include information security and cloud computing security.

DONGYOUNG KOO received the B.S. degree in computer science from Yonsei University, Seoul, South Korea, in 2009, and the M.S. and Ph.D. degrees in computer science from KAIST, in 2012 and 2016, respectively. He is currently an Associate Professor with the Department of Convergence Security, Hansung University, Seoul. His research interests include information security, secure cloud computing, and cryptography.

JUNBEOM HUR (Member, IEEE) received the B.S. degree in computer science from Korea University, Seoul, South Korea, in 2001, and the M.S. and Ph.D. degrees in computer science from KAIST, in 2005 and 2009, respectively. He was with the University of Illinois at Urbana–Champaign as a Postdoctoral Researcher, from 2009 to 2011. He was with the School of Computer Science and Engineering, Chung-Ang University, South Korea, as an Assistant Professor, from 2011 to 2015. He is currently a Professor with the Department of Computer Science and Engineering, Korea University. His research interests include information security, cloud computing security, mobile security, and applied cryptography.