applied sciences

MDPI

Article

Exploiting Hidden Information Leakages in Backward Privacy for Dynamic Searchable Symmetric Encryption

Hyundo Yoon [1], Muncheon Yu [1], Changhee Hahn [2,*], Dongyoung Koo [3,*] and Junbeom Hur [1]

[1] Department of Computer Science and Engineering, Korea University, Seoul 02841, Republic of Korea; hdyoon@isslab.korea.ac.kr (H.Y.); mcyu@isslab.korea.ac.kr (M.Y.); jbhur@korea.ac.kr (J.H.)
[2] Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul 01811, Republic of Korea
[3] Department of Convergence Security, Hansung University, Seoul 02876, Republic of Korea
* Correspondence: chahn@seoultech.ac.kr (C.H.); dykoo@hansung.ac.kr (D.K.)

Abstract: Dynamic searchable symmetric encryption (DSSE) enables searches over encrypted data as well as data dynamics such as flexible data addition and deletion operations. A major security concern in DSSE is how to preserve forward and backward privacy, which are typically achieved by removing the linkability between the newly added data and previous queries, and between the deleted data and future queries, respectively. After information leakage types were formally defined for different levels of backward privacy (i.e., Type-I, II, III), many backward private DSSE schemes have been constructed under the definitions. However, we observed that the backward privacy can be violated by leveraging additional secondary leakage, which is typically leaked in specific constructions of schemes in spite of their theoretical guarantees. In this paper, in order to understand the security gap between the theoretical definitions and practical constructions, we conduct an in-depth analysis of the root cause for the secondary leakage, and demonstrate how it can be abused to violate Type-II backward privacy (e.g., the exposure of the deletion history) of DSSE constructions in practice. We then propose a novel Type-II backward private DSSE scheme based on Intel SGX, which is resilient to the secondary leakage abuse attack. According to the comparative analysis of our scheme with the state-of-the-art SGX-based DSSE schemes, Bunker-B (EuroSec'19) and SGX-SE1 (ACNS'20), our scheme shows higher efficiency in terms of the search latency with a negligible utility loss under the same security level (cf. Bunker-B) while showing similar efficiency with a higher security level (cf. SGX-SE1). Finally, we formally prove that our scheme guarantees Type-II backward privacy.

Keywords: dynamic searchable encryption; information leakages; forward security; backward security

Citation: Yoon, H.; Yu, M.; Hahn, C.; Koo, D.; Hur, J. Exploiting Hidden Information Leakages in Backward Privacy for Dynamic Searchable Symmetric Encryption. *Appl. Sci.* **2024**, *14*, 2287. https://doi.org/10.3390/app14062287

Academic Editors: Pengpeng Chen, Liangyin Chen and Yanru Chen

Received: 30 January 2024
Revised: 4 March 2024
Accepted: 6 March 2024
Published: 8 March 2024

1. Introduction

Dynamic searchable symmetric encryption (DSSE) is a kind of searchable symmetric encryption (SSE) specifically designed to support data dynamics such as addition and deletion operations in an encrypted database (EDB) [1,2]. Although DSSE schemes benefit from the flexible operations without decryption, they are likely to leak sensitive information. For example, an adversary can observe added or deleted documents that are accessed by users by exploiting the access pattern leakage [3], or identify the underlying keyword of queries by exploiting the search pattern leakage [3].

To formally address this information leakage problem, Bost et al. [4,5] introduced the notions of forward and three different types (i.e., Type-I, II, III) of backward privacy in DSSE, mainly focusing on addressing security concerns regarding the linkability between queries (e.g., update and search) and updated data. Although leaking less information implies higher security, it inevitably incurs higher computational overhead. For example, Type-I backward privacy has been only achieved by adopting cryptographically heavy operations such as oblivious RAM (ORAM) [6].

Recently, a trusted execution environment (TEE) such as Intel SGX [7] has been considered for DSSE to mitigate efficiency problems caused in favor of the forward/backward privacy preservation [8–11]. Amjad et al. [8] proposed several forward and various types of backward private DSSE schemes using Intel SGX. Their first scheme, called Fort, achieves the highest security guarantee (i.e., Type-I), but suffers from high communication overhead due to the usage of ORAM [6]. To balance security and efficiency, they proposed another scheme, called Bunker-B (Type-II), aiming to reduce communication costs. However, it still suffers from scalability degradation.

To resolve this problem, Vo et al. proposed SGX-SE1, providing Type-II backward privacy [11], and Maiden providing Type-I backward privacy [12], both leveraging the server-side SGX enclave as a proxy to reduce the communication cost and enhance the scalability of the scheme. Unfortunately, despite its efficiency improvement, we observed that SGX-SE1 does not fully guarantee Type-II backward privacy as expected. SGX-SE1 has additional information leakage related to deletion history, which is information leakage only allowed in Type-III backward privacy, by exploiting secondary leakage, which is naturally allowed in the protocol construction. Thus, in this paper, we seek to answer the following questions: *What is the root cause or conditions for the leakage, and how can it be exploited to break the backward privacy of DSSE schemes?*

In search of the answer, we first conduct an in-depth analysis of the information leakages in existing Type-II backward private DSSE schemes [8,11,13,14] in terms of both theoretical definitions and scheme constructions. Consequently, we found that there exist information leakages in several schemes [11,14], which can be utilized to extract a deletion history of the encrypted data, leading to violation of Type-II backward privacy. Then, we examine the condition allowing the vulnerability, and demonstrate how it can affect the Type-II backward privacy in practice by exploiting the secondary information leakage on Vo et al.'s [11] and Sun et al.'s [14] DSSE schemes.

Next, we propose a novel forward and Type-II backward private DSSE scheme based on SGX. To this end, we design an obfuscation technique to hide the access and the search patterns in order to prevent the information leakages related to update operations. To minimize extra communication and computation costs caused by the obfuscation technique, we selectively cache the top k-frequently accessed documents inside the SGX enclave for fast data retrieval and obfuscation.

According to our comparative analysis results with the state-of-the-art SGX-based DSSE schemes, Bunker-B [8] and SGX-SE1 [11], the search time of the proposed scheme is approximately 27× faster than that of Bunker-B, while providing the same security level. Compared with SGX-SE1, the proposed scheme achieves a higher level of security while minimizing performance degradation.

Contributions: Our contributions are summarized as follows:

- We conduct a comprehensive analysis of the existing Type-II backward private schemes, and discover information leakage that falls outside the purview of the existing backward privacy notions. We then demonstrate how those leakages are exploited to extract the deletion history in Type-II schemes.
- We demonstrate how our findings on information leakages exacerbate the security of known Type-II backward private schemes by exploiting the Vo et al. DSSE scheme [11] and Sun et al. DSSE scheme [14].
- We design a novel forward and Type-II backward private DSSE scheme based on SGX, which hides the information leakage of deletion history with high efficiency.
- We conduct a comparative analysis of our scheme with the state-of-the-art SGX-based DSSE schemes, Bunker-B [8] and SGX-SE1 [11], in both synthetic and real-world Enron [15] datasets. According to the analysis, our scheme shows higher efficiency in search latency with negligible utility loss under the same security level (cf. Bunker-B) while showing similar efficiency with a higher security level (cf. SGX-SE1).

2. Preliminaries

In this section, we introduce the basic background of Intel SGX, the preliminaries, and the cryptographic background of forward/backward privacy in DSSE. λ denotes the security parameter, and $negl(\lambda)$ denotes a negligible function in the security parameter. A database DB denotes a list of file identifier and keyword set pairs.

2.1. Intel SGX

A trusted execution environment (TEE) allows an arbitrary code to be executed in an isolated environment, providing integrity and confidentiality protection on the executed codes, stored data, and runtime states, such as sensitive I/O, CPU registers, and memory. Intel SGX, one of the main prevailing and representative TEEs, is a set of extended Intel's x86 instructions providing isolated execution environments, called enclaves [7]. The enclave is the trusted component located in a dedicated memory portion of the physical RAM, called the enclave page cache (EPC). During system boot-up, a total of 128 MB is typically reserved for the Intel SGX, out of which 96 MB is allocated to the EPC; this EPC is shared among all the running enclaves on the system. The untrusted part is executed as an ordinary process, and can trigger the enclave under a securely defined process. When the data are loaded inside the enclave, the SGX-enabled processors protect their integrity and confidentiality by isolating them from the outside untrusted environment, including the operating systems and hypervisor. The enclave can access the entire virtual memory of its untrusted host process, but the untrusted host or any other enclave cannot directly access any code or data in the enclave.

When an enclave is required to communicate with other instances, an attestation mechanism is executed for authentication before exchanging the data. There are two forms of attestation: local attestation and remote attestation. Local attestation is used for authenticating two enclaves running on the same physical machine, and is conducted using the EREPORT and EGETKEY instructions, which generate signed reports for verification. Remote attestation is used when an enclave attests its identity (usually represented by its initial code and data) to another enclave running on a remote physical machine. The remote enclave uses a quote generated by the quoting enclave (QE), and verifies the signature by sending it to the Intel attestation server. Through the attestation service, a secure channel is established between them.

2.2. Definition of DSSE

A DSSE scheme consists of three algorithms: **Setup**, **Update**, and **Search**. Let λ denote the security parameter, and it is implicit input for all algorithms. DB $= \{(ind_i, w_i)\}$ refers to the database, where *ind* denotes file identifier and w denotes the keyword. EDB denotes an encrypted database, and σ denotes the state of the client. op denotes the type of update operation (addition or deletion). Each algorithm is defined as follows:

- (EDB, σ) $\leftarrow$ **Setup**(1^λ, DB): **Setup** takes as input the security parameter λ and the initial database DB.
- (σ', EDB′) $\leftarrow$ **Update**(σ, op, ind; EDB): **Update** is a client-server protocol, where the client takes as input the state σ, a file identifier ind, and type of operation op $= \{add, del\}$; and the server takes as input the encrypted database EDB. The client outputs updated state σ', and the server outputs an updated encrypted database EDB′ as a result.
- (σ', DB(w); EDB′) $\leftarrow$ **Search**(σ, w; EDB): **Search** is also a client-server protocol, where the client takes as input the state σ and search keyword w; the server takes as input the encrypted database EDB. Then, the client outputs σ' and DB(w) of the file identifiers matching keyword w; the server outputs the updated encrypted database EDB′.

The adversary is only allowed to learn the information explicitly captured by the leakage function $\mathcal{L} = (\mathcal{L}^{Setup}, \mathcal{L}^{Srch}, \mathcal{L}^{Updt})$.

2.3. Forward and Backward Privacy of DSSE

2.3.1. Forward Privacy

Forward privacy [16] ensures newly updated data entries cannot be related to the previous search queries. The forward-private DSSE scheme hides the linkability between an update operation and the past search queries [2]. Thus, if a client performs any update operations (i.e., document/keyword add or delete) on a keyword w, the adversary, including the data server, cannot learn any information whether searches on keyword w have been made beforehand, mitigating the file injection attack [17] in DSSE schemes.

2.3.2. Backward Privacy

When searching on keyword w is made, backward privacy [2,5] guarantees that the deleted document identifiers containing w are not revealed.

Now, we introduce a leakage function defined in backward privacy. The record information revealed in a list of queries issued so far is represented as the above leakage function $\mathcal{L}$. The list of queries is denoted as Q, where

$$Q = \{(u, \mathsf{op}, \mathsf{in}) \; or \; (u, w)\}.$$

There are two types of query: search query and update query. (u, w) is a search query, where u refers to the timestamp and w refers to the search keyword. Note that the timestamp u starts from 0 and increases with the incoming queries. $(u, \mathsf{op}, \mathsf{in})$ is an update query, where $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ and in refers to the input (w, ind).

The search pattern [3], $sp(w)$, is one of the common leakage function, composed of timestamps of all search queries on keyword w, which allows an adversary to identify identical search queries. The formal definition of $sp(w)$ is defined as follows:

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

$TimeDB(w)$, for a keyword w, leaks the list of all documents matching a search keyword w and the timestamp of their insertion, except for documents that are deleted. The formal definition is as follows:

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

Note that u' is the timestamp that comes after the u. $Updates(w)$ refers to the list of timestamps of all updates on keyword w. Formally, $Updates(w)$ is defined as :

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

Lastly, the leakage function $DelHist(w)$ is defined as follows:

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

Bost et al. [5] introduced the formal definitions for three different types of backward privacy ordered from the most to least secure notions, which are defined as follows:

- Type-I: *It leaks the document identifiers matching the keyword w, timestamps of their insertion $(TimeDB(w))$, and total number of updates on w.*
- Type-II: *It leaks the document identifiers matching the keyword w, $TimeDB(w)$, and the list of timestamps of the entire updates on keyword w $(Updates(w))$.*
- Type-III: *It leaks the document identifiers matching the keyword w, $TimeDB(w)$, $Updates(w)$, and deletion history $(DelHist(w))$ that reveals which deletion update canceled which insertion update.*

3. Leakage Abuse Attack

In this section, we introduce how leakage information we found can be exploited by an adversary to obtain the deletion history of the state-of-the-art Type-II backward privacy schemes. Table 1 shows state-of-the-art Type-II backward private schemes that are prone to our attack. In order to demonstrate its feasibility, we show how our attack can be conducted on SGX-SE1 [11] and Aura [14] as representative SGX-based and non-SGX-based Type-II backward privacy schemes, respectively. Finally, we discuss the root cause of the vulnerability of the scheme constructions.

Table 1. Classification of existing Type-II DSSE schemes.

Scheme	BP-Type	Prone to Attack	SGX
Fides	Type-II	X	-
Mitra	Type-II	X	-
Bunker-B	Type-II	X	O
SGX-SE1	Type-II	O	O
SGX-SE2	Type-II	O	O
Aura	Type-II	O	-

3.1. Threat Model

Similar to the previous DSSE schemes [8,11], we consider a semi-honest adversary at the server side, such that the adversary can observe the interaction of the enclave with the other resources located outside the enclave, and has the privilege of gaining full access over software stack outside the enclave, as well as the operating system and hypervisor. Additionally, the adversary can learn information about the access patterns by observing memory addresses and encrypted data in the encrypted database. Finally, the adversary can log the timestamps of every memory manipulation during the entire protocol, aiming to extract the deletion history.

3.1.1. Trust Assumptions on Intel SGX

We assume that the SGX enclave behaves normally without hardware bugs or backdoors. The preset code and data inside the enclave are securely protected, and cryptographic primitives provided by SGX are trusted [7]. Furthermore, the communications and data transfer between different clients or servers are protected by the secure channels established by the SGX attestation service. The enclave can be invoked whenever clients need. We do not consider the side-channel and denial-of-service (DoS) attacks against the SGX as in many other SGX-based applications [13,18–21].

3.2. Extraction of Deletion History

As described in Section 2.3.2, Type-II backward privacy only leaks document identifiers matching the searched keyword w, $TimeDB(w)$, and $Updates(w)$. If a scheme leaks $DelHist(w)$ along with the other Type-II leakages, the adversary is allowed to know which deletion update cancels which add update that previously occurred, downgrading it into Type-III backward privacy. Thus, if a part of deletion history is revealed to an adversary, the privacy level of the scheme may be weakened than the original Type-II backward privacy level.

According to our investigation of the existing Type-II backward private schemes, we observed that they may leak $DelHist(w)$ when an adversary can exploit the access and search pattern leakages together with information leakages allowed in Type-II backward privacy (especially when they are constructed with static data structures or static values for identifying documents matching a specific keyword w). In order to obtain it, specifically, the adversary should be able to (1) distinguish whether the update operation is addition or deletion, and (2) distinguish whether two separate queries are on the same keyword w

(search pattern leakage). When an addition operation is executed, the adversary is able to observe the added document identifier id from the update query (but not the keyword w). Thus, the adversary learns $(u, add, (w, id))$, where u refers to the timestamp of the operation. According to the definition of Type-II backward privacy, $DelHist(w)$ should be hidden from the view of the server.

Next, during the search protocol for a specific keyword, the server is able to observe the query tokens and memory access of the EDB (i.e., the access pattern). Note that each query token refers to a single document. If the query token or the search index is encrypted, the adversary may observe the following search query consisting of the encrypted i query tokens $Q = \{qt_1^w, qt_2^w, ..., qt_i^w\}$, where qt_j^w refers to the j-th query token for keyword w for $1 \leq j \leq i$. The adversary then observes and learns the search result $Res = \{id_1, id_2, ..., id_i\}$, directly indicating the access pattern of the EDB for the documents in Res.

When a client sends a search query Q' consisting of k query tokens later, the adversary would observe $Q' = \{qt_1^{w'}, qt_2^{w'}, ..., qt_k^{w'}\}$, and learn $Res' = \{id'_1, id'_2, ..., id'_k\}$, for the search result of Q'. If the value of query token qt is encrypted in a deterministic manner and is bound to a specific keyword, then the adversary can identify whether two different search queries are linked to the same keyword w by simply comparing the values of the query tokens (When the query token is generated in a non-deterministic manner, such as re-encryption, as in Bunker-B [8], for example, our extraction of deletion history might not work due to the indistinguishability in the adversary's view.).

When observing $i > k$, the adversary may learn the deletion updates are performed between two searches of Q and Q' on the same keyword w. Further, by comparing the search results Res and Res', the adversary can learn which document id was deleted. Consequently, the probability that the adversary can determine which delete updates are related to w is $P_{Del(w)} = \frac{i-k}{N_{del}}$, where N_{del} is the number of possible deletion operations performed in the period. According to the definition of Type-II backward privacy, any information on $DelHist(w)$ should not be leaked. However, by exploiting the query token, search result, and access pattern of the EDB, the adversary can learn which delete update cancels which add update, leaking $DelHist(w)$ information with a probability of $P_{Del(w)}$. In addition, as mentioned in Section 2.2, if there exists an index such that the timestamp of its insertion and deletion is revealed to the adversary, the scheme leaks $DelHist(w)$. Thus, if a deletion history can be successfully extracted, the claimed Type-II backward privacy is violated. It is important to note that even though queries are encrypted using a randomized encryption algorithm, the extraction of deletion history can still be applied as long as the search pattern and the other information required for the extraction are leaked. In Section 3.4, we demonstrate the efficacy of the extraction by showing that $P_{Del(w)}$ can be non-negligible in the real-world scenario.

3.3. Attacks on Prior Works

We show how the deletion history can be extracted in Vo et al.'s SGX-SE1 [11] and Sun et al.'s Aura [14]. For better understanding, we briefly explain the overview of each scheme's construction first. Then, we demonstrate how the deletion history can be extracted from each protocol.

3.3.1. Vo et al. Scheme

Vo et al. [11] proposed SGX-based forward and Type-II backward private dynamic searchable encryption schemes, named SGX-SE1 and SGX-SE2. In this paper, we focus only on SGX-SE1 because it is the baseline scheme upon which SGX-SE2 is built.

When a client uploads a new document in SGX-SE1, the document is sent to the SGX enclave in the server through the addition operation in the update protocol. The enclave then parses all the keywords within the document, and uses the latest state $ST[w]$, which infers the number of documents containing w. Specifically, for each keyword w, $ST[w]$ increases by 1 whenever a document containing w is added. The encrypted index u is a deterministic value generated by $H(k_w, c)$, where H refers to a hash function that

hashes k_w, which is a key value bound to w, and the count value c of $ST[w]$. A map of the encrypted index $M_I[u]$ stores the encrypted document identifier.

When deleting a document, the client transfers the corresponding document identifier id to the enclave. On receipt of it, the enclave stores the id within a deleted document list d, and the actual deletion of it from the EDB is conducted during the search protocol.

In the search protocol, when the enclave receives a keyword w from the client, it first fetches the document identifiers from the deleted document list d. Next, the enclave loads the corresponding documents and checks whether the keyword w exists within each document. The enclave retrieves $ST[w]$ of the deleted documents, which were used when they were added, and excludes these values from $\{0, ..., ST[w]\}$. The remaining values in the set are then used to generate query token u for the non-deleted documents. The list of query tokens u is transferred to the server. Finally, the server computes the id by decrypting $M_I[u]$, and returns the corresponding encrypted documents to the client.

Extraction Scenario. Table 2 shows the example flow of SGX-SE1 protocol [11], along with the leakage information and its type. When searching on keyword w in SGX-SE1, the enclave sends a search query Q_w containing a list of (u, k_{id}) pairs to the server, where k_{id} refers to the key value bound to the document identifier. The adversary can observe and trace from Q_w the deterministic values of u's, which are bound to the specific keyword. Therefore, by comparing the values of u from the past search, the adversary learns if two different queries are on the same keyword, leading to the search pattern leakage. Because the document identifiers and encrypted documents are retrieved in the untrusted area, the adversary can observe the access pattern of the matching result as well as the accessed document identifiers. The adversary compares the matching results of the two queries and learns that deletion update on id_3 occurred in the period between the two searches (as shown in Table 2). Among the three delete updates that occurred between the two searches, one of them must correspond to the deletion of id_3, thus the probability of making a correct guess is $\frac{1}{3}$. Since the adversary has the knowledge of the timestamps of all updates and when each document is added with their identifier values, the aforementioned leakage information allows the adversary to learn $DelHist(w)$.

Table 2. Example of SGX-SE1 protocol flow.

Algorithm	Leakage Information (Attacker's View)	Leakage Type
$Setup(1^\lambda)$	-	-
$Update(add, (doc_1, id_1))$	(add, id_1)	Update Pattern
...	...	...
$Search(w)$	$Q_w \rightarrow \{(u_1, k_{id_1}), (u_2, k_{id_3}), (u_3, k_{id_4})\}$	Search Pattern
	$Res \rightarrow \{id_1, id_3, id_4\}$	Access Pattern
$Update(del, id_{12})$	-	-
$Update(del, id_3)$	-	-
$Update(del, id_9)$	-	-
$Search(w)$	$Q_w \rightarrow \{(u_1, k_{id_1}), (u_3, k_{id_4})\}$	Search Pattern
	$Res \rightarrow \{id_1, id_4\}$	Access Pattern

Q_w: search query for keyword w, Res: search result, id: document identifiers, u: encrypted index, k_{id}: key value for id.

3.3.2. Sun et al. Scheme

Sun et al. [14] proposed a non-SGX-based Type-II backward private scheme called Aura. Aura requires the client to revoke the encryption key after each search. The search result does not need to be re-encrypted because the previous search result is cached.

When the client adds a keyword and document identifier ind to the database, the client retrieves the most recently used encryption key msk, and computes a ciphertext with ind and a tag $t = F_{K_t}(w, \text{ind})$, where F is a pseudo-random function and K_t is secret key for t. Then, the computed ciphertext is inserted into the database EDB_{add}. For deletion, the client inserts tag t corresponding to the deleting entry (w, ind) into the deletion list D.

For searching on keyword w, the client retrieves the number of searches on keyword w, denoted as i, the current secret key sk, and the deletion list D. Then, the client computes the revoked secret key sk_R and query token tkn, sends them to the server, and refreshes msk. The server retrieves encrypted indices matching w and decrypts the non-deleted indices with sk_R. The non-deleted indices are added to a list NewInd; the deleted tags are added to a list DelInd. Then, the server retrieves indices stored in $\text{EDB}_{cache}[tkn]$ and excludes entries that are in DelInd, where EDB_{cache} stores the previous search result. Finally, NewInd together with non-deleted indices stored in cache $\text{EDB}_{cache}[tkn]$ are returned to the client.

Extraction Scenario. As explained above, during the search protocol, the server accesses $\text{EDB}_{cache}[tkn]$ for retrieving cached search results. By comparing the accessed $\text{EDB}_{cache}[tkn]$, the adversary can learn which previous search query is on the same keyword w. The timestamps of addition update queries on w (i.e., $TimeDB(w)$) that occurred between two search queries on w are revealed to the adversary. Note that remaining update queries between two search queries can be possibly deletion updates on w. During the execution of the second search query, the adversary can obtain additional information on how many deletion updates on w occurred by observing the number of entries excluded from EDB_{cache}. Since the entries stored in EDB_{cache} are in the form of (ind, t), the deleted indices are also revealed while excluding deleted entries from the cache.

Because the insertion timestamps of each (w, ind) are revealed, the adversary, similar to the case of Vo et al.'s scheme [11], is able to extract $DelHist(w)$ using the timestamps of possible deletion updates on w and deleted ind. However, the attack against Aura has a lower success rate than that of SGX-SE. The reason is that Aura does not follow the traditional definition of Type-II backward privacy and follows a somewhat different definition for Type-II backward privacy.

3.3.3. Discussion

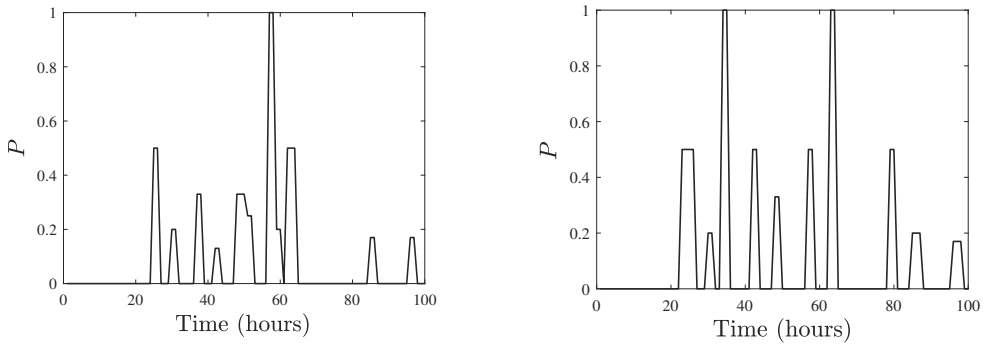
As shown in the above attack scenarios, in addition to the information leakages defined in Type-II backward privacy, there exists extra information that an adversary may learn from the protocols and exploit for extracting a deletion history. A root cause for allowing the additional leakage is the static data structures and static values used for updating and searching a keyword. During the search protocol, for instance, SGX-SE1 [11] and Aura [14] access the same locations of data structures when searching on the same keyword. Thus, the adversary is able to specify whether a certain document is added or deleted when compared with the same information learned from the previous search query on the same keyword. Bunker-B [8], on the other hand, accesses different locations of data structure for every search query on the same keyword, due to the re-encryption of the entries after every search. However, the re-encryption of entries incurs high computational overhead, degrading the practicality of the scheme. It is thus a challenging problem to minimize extra information leakages while achieving high efficiency.

3.4. Feasibility of Deletion History Extraction

We evaluate the feasibility of deletion history extraction by conducting a simulation on its probability in diverse distribution models of data upload, delete, and keyword search in the cloud storage. In the simulation, we randomly generate queries and analyze the probability of deletion history extraction.

According to the distribution models for file transfer [22] and search query [23], we assume that the document upload follows a Poisson distribution with rate $\tilde{\lambda}$, the data lifespan follows an exponential distribution with a mean duration $\frac{1}{\mu}$, and the search request on the same keyword follows an exponential distribution with a mean duration $\frac{1}{\mu'}$. Also, the keyword frequency of the documents follows a Zipf distribution.

Based on the aforementioned distribution models, we evaluate the feasibility of deletion history extraction by measuring its probability under various simulation settings. Figure 1 shows the evaluation results in each case. In the figure, the horizontal axis represents the time in hours, and the vertical axis represents the probability of successful extraction $P_{Del(w)}$, which is the probability of identifying which document is deleted. When simulating for 100 h with $\tilde{\lambda} = 3$, $\frac{1}{\mu} = 40$, and $\frac{1}{\mu'} = 1$, we have observed a total of 11 time instances that enable our extraction with non-negligible probabilities. As shown in Figure 1a, for example, there exists a time instance when the extraction succeeds with 100% (between 56 and 57 h), leading to the violation of Type-II backward privacy. As another example, one can observe that $P_{Del(w)}$ reaches 0.5 two times between 20 and 70 h, meaning that the deleted document can be identified with probability $\frac{1}{2}$, violating Type-II backward privacy in a probabilistic (but still pragmatically meaningful) way in those periods. Figure 1b shows the evaluation results when search requests are performed on average four times more frequently than in Figure 1a. As a result, we could observe that there are two time instances showing $P_{Del(w)} = 1$ (between 33 and 34; 61 and 62 h) and five time instances showing $P_{Del(w)} = 0.5$.



(**a**) (**b**)

Figure 1. Extraction success rate. (**a**) $\tilde{\lambda} = 3$, $\frac{1}{\mu} = 40$, $\frac{1}{\mu'} = 1$. (**b**) $\tilde{\lambda} = 3$, $\frac{1}{\mu} = 40$, $\frac{1}{\mu'} = \frac{1}{4}$.

In Figure 1a,b, on average, 53 update queries (consisting of 41 addition and 10 deletion queries) were generated for the target keyword. When considering all of the update queries in the above simulations, the adversary could extract 1.6% and 3.2% of actual deletion history for the target keyword, respectively. For deletion queries for the target keyword, the adversary could extract 10% and 20% of deletion queries. When considering the attack success rate above $P_{Del(w)} = 0.5$, the probability of possible deletion history extraction non-negligibly increases.

The aforementioned simulations indicate that the extraction probability increases as the number of search requests on the same keyword increases (compare Figure 1a,b) When calculating $P_{Del(w)}$, the total number of deletions that occurred between two searches on the same keyword, N_{del}, affects the probability. It is clear that if the search requests are sent more frequently, N_{del} would be smaller with high probability, leading to an increase in $P_{Del(w)}$.

4. Our Construction

In this section, we propose a novel forward and Type-II backward private DSSE scheme based on SGX.

4.1. System Overview

Figure 2 shows the overview of the proposed scheme. There are three entities in the system: the client, the server, and the SGX enclave within the server. The client is the data owner, which is assumed to be trusted. We assume that all the entities above act normally and do not consider incorrect queries or wrong operations. As described in Section 3.1.1, although the enclave is located within the untrusted server, the server cannot observe the code or data inside the enclave. Also, we assume that the enclave is fully trusted and resilient to side-channel attacks.

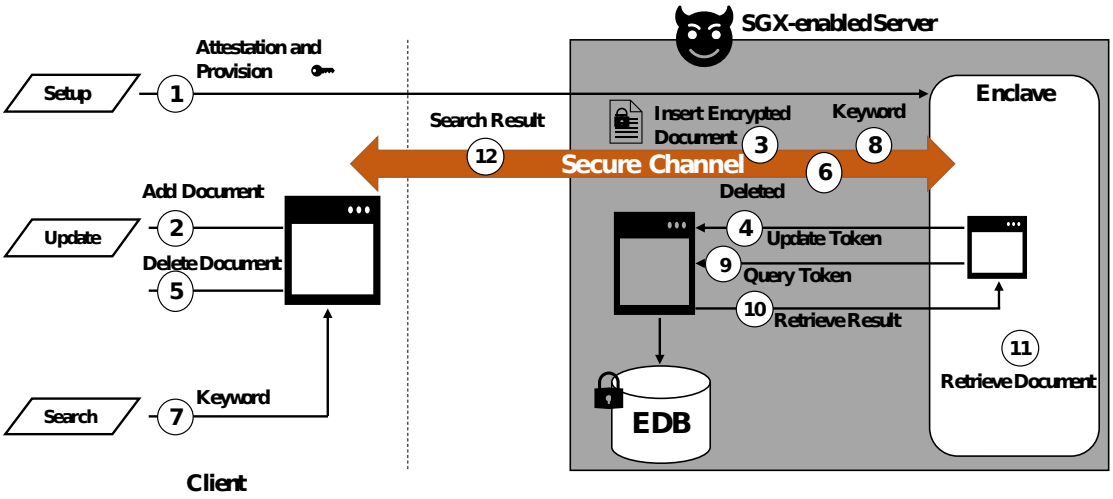


Figure 2. Overview of the proposed scheme.

In the proposed scheme, the client establishes a secure channel with the authenticated enclave using the SGX remote attestation service. The client provisions a secret key K in the enclave through the secure channel (step ①). During this procedure, the client does not deploy any encrypted database (EDB) to the server as in the other DSSE schemes [5,8].

The update procedure in the proposed scheme consists of two different operations: addition and deletion. When adding a document to the cloud storage, the client assigns a unique identifier *id* to the document, and encrypts the document with a key K (Step ②). Then, the encrypted document and its *id* are sent to the enclave through a secure channel (Step ③). With the received *id* and encrypted document, the enclave extracts all the keywords within the document, and performs cryptographic operations to generate an update token with the encrypted index and state. The update token and encrypted document are then transferred to the server (Step ④). The map of the encrypted index M_I is separately managed in the untrusted area. When deleting a document, the client simply sends the document identifier *id* to the enclave (Step ⑤ & Step ⑥). (The actual deletion process of the document will be further explained in Section 4.3).

When retrieving documents from the outsourced EDB, the client sends the search keyword w to the enclave through the secure channel (Step⑦& Step⑧). The enclave then computes the query tokens, and sends the final search query to the server except the tokens for the deleted documents (Step⑨). The server then searches over the encrypted index, and returns back the document identifier list to the enclave (Step⑩). With the identifier list, the enclave searches over the cached document list; and returns the identifiers not matching any of the cached documents to the server to retrieve the encrypted documents (Step⑪). Finally, the enclave sends the matching documents to the client through the secure channel (Step⑫).

4.2. Design Goal

As analyzed in Section 3, Vo et al. [11] and Sun et al. [14] schemes are vulnerable to extraction of deletion history, related to leakage profile of Type-III, although their initial security aimed at Type-II backward privacy. Our principal objective is to design a *'solid'*

Type-II backward private scheme, which only leaks information that is allowed in Type-II backward privacy.

4.3. Scheme Construction

To facilitate searches, our scheme lets the enclave store the following information: (1) the map ST storing the pairs of a keyword and its latest state, (2) the deleted document list d, (3) the map D storing the pairs of a keyword and a deleted document's id in d associated with the keyword, (4) the map SC storing the pairs of a document id and the number of searches over the document, and (5) the map FD storing the pairs of a document id and its encrypted data in the EDB. Each notation is defined in Table 3.

Table 3. Notations of data structures.

Notation	Definition
K	Secret key
ST	Latest state of keywords
d	List of deleted documents
D	A map for keyword and deleted documents
SC	Number of times each document searched
FD	k-frequent documents
R	Repository (EDB)
M_I	Encrypted index
M_c	Encrypted state

The proposed scheme consists of three algorithms: $Setup$ (Algorithm 1), $Update$ (Algorithm 2), and $Search$ (Algorithm 3). In the algorithms, H_1, H_2, and H_3 denote the cryptographic hash functions, and $\mathsf{Enc}(a, b)$ is the encryption of b under a key a.

Algorithm 1 $Setup(1^\lambda)$

Client:
$k_\Sigma, k_f \leftarrow\$ \{0,1\}^\lambda$
Launch Remote Attestation ▷ Establish secure channel
Send $K = (k_\Sigma, k_f)$ to Enclave
Enclave:
Initialize maps ST, D, SC, FD; tuples T_1, T_2; a list d
Receive $K = (k_\Sigma, k_f)$
Server:
Initialize maps M_I and M_c
Initialize Repository R

In the $Setup$ algorithm, the client communicates with the enclave and provisions a secret key K containing a key pair (k_Σ, k_f) through the secure channel established by the attestation service provided by Intel [7]. The enclave uses k_Σ to generate update and query tokens and a symmetric key k_f to encrypt/decrypt a document. The enclave initializes the maps ST and D and the list d. The server initializes the maps M_I and M_c, where M_I and M_c refer to the encrypted index and the encrypted state, respectively. Also, the repository R stores the encrypted document with its document identifier id.

Algorithm 2 $Update(\text{op}, \text{in})$

Client:
if op $=$ *add* **then**
  $f \leftarrow \mathsf{Enc}(k_f, \text{doc})$
  send (op, id, f) to Enclave
else ▷ when op $=$ *del*
  send (op, id) to Enclave
end if
Enclave:
if op = *add* **then**
  send (id, f) to Server
  $\{(w, id)\} \leftarrow Parse(\mathsf{Dec}(k_f, f))$
  foreach (w, id) **do**
    $k_w \| k_c \leftarrow F(k_\Sigma, w)$
    $c \leftarrow ST[w]$
    if $c = \perp$ **then** $c = -1$
    end if
    $c \leftarrow c + 1$
    $k_{id} \leftarrow H_1(k_w, c)$
    $(u, v) \leftarrow (H_2(k_w, c), \mathsf{Enc}(k_{id}, id))$
    add (u, v) to T_1
    $(u', v') \leftarrow (H_3(k_w, id), \mathsf{Enc}(k_c, c))$
    add (u', v') to T_2
  end for
  send (T_1, T_2) to Server
else ▷ when op $=$ *del*
  add *id* to *d*
end if
Server: ▷ when op $=$ *add*
$R[id] \leftarrow f$
receive (T_1, T_2) from Enclave
foreach (u, v) **in** T_1 **do**
  $M_I[u] = v$
end for
foreach (u', v') **in** T_2 **do**
  $M_c[u'] = v'$
end for

The *Update* algorithm takes as an input (op,in), where op can be either *add* or *del*, and *in* would be (doc, *id*) if op = *add*, or *id* if op = *del*. When an add update is invoked (i.e., op = *add*) in the algorithm, the client generates an encrypted document f by encrypting the doc under a key k_f. The enclave receives (op, f, id) from the client. The received (id, f) is stored in repository R, located in the server area. Then, the enclave parses the received doc to retrieve a list of $\{(w, id)\}$. For each keyword w, the enclave uses k_Σ to generate k_w and k_c, and retrieves the latest state c. Then, k_{id} is generated from c by executing the hash function as $k_{id} = H_1(k_w, c)$; and two encrypted entries (u, v) and (u', v') are generated using k_w, k_c, and k_{id} as follows:

$$(u, v) = (H_2(k_w, c), \mathsf{Enc}(k_{id}, id)),$$

where the values of (u, v) contain the mapping information between c and id that is retrieved based on u and k_{id}.

$$(u', v') = (H_3(k_w, id), \mathsf{Enc}(k_c, c)),$$

where (u', v') holds the encrypted state of the document with identifier id, and allows the client to check the states of deleted documents by sending u' in the *Search* algorithm. The enclave adds each (u, v) and (u', v') to tuples T_1 and T_2, respectively, and sends (T_1, T_2) to the server for updating M_I and M_c. When a deletion update is invoked (i.e., op $= del$), the client simply sends id to the enclave. On receipt of it, the enclave updates the list d by adding the received d to the list, and does not require further computations or communications with the server.

Algorithm 3 *Search*(w)

Client:
Send w to Enclave
Enclave:
$st_{w_c} \leftarrow \{\emptyset\}$, $Q_w \leftarrow \{\emptyset\}$
$k_w \| k_c \leftarrow F(k_\Sigma, w)$
foreach id_i **in** d **do**
    $f_i \leftarrow R[id_i]$
    $doc_i \leftarrow \mathsf{Dec}(k_f, f_i)$
    if w **in** doc_i **then**
        $D[w] \leftarrow id_i \cup D[w]$
    end if
end for
foreach id **in** $D[w]$ **do**
    $u' \leftarrow H_3(k_w, id)$
    $v' \leftarrow M_c[u']$
    $c \leftarrow \mathsf{Dec}(k_c, v')$
    $st_{w_c}^{del} \leftarrow \{c\} \cup st_{w_c}^{del}$
end for
$st_{w_c} \leftarrow \{0, ..., ST[w]\} \setminus st_{w_c}^{del}$
foreach c **in** st_{w_c} **do**
    $u \leftarrow H_2(k_w, c)$
    $k_{id} \leftarrow H_1(k_w, c)$
    with prob. p, $Q_w \leftarrow \{(u, k_{id})\} \cup Q_w$
end for
send Q_w to Server
receive Q_{docId} from Server
foreach id_i **in** Q_{docId} **do**
    if $sc = \perp$ **then** $sc = 1$
    else
        $sc \leftarrow SC[id_i]$
        $SC[id_i] \leftarrow sc + 1$
    end if
    if id **in** FD **then**
        $doc_i \leftarrow FD[id_i]$
    else
        $doc_i \leftarrow R[id_i]$
    end if
    add doc_i to Res
end for
Send Res to Client
Update FD
Server:
receive Q_w from Enclave
foreach (u_i, k_{id_i}) **in** Q_w **do**
    $id_i \leftarrow \mathsf{Dec}(k_{id_i}, M_I[u_i])$
    $Q_{docId} \leftarrow \{id_i\} \cup Q_{docId}$
end for
send Q_{docId} to Enclave

In the *Search* algorithm, when the client sends a query containing a keyword w to the enclave through a secure channel, the enclave begins with checking deleted documents in the list d. Specifically, all of the documents in d are loaded into the enclave. The enclave then decrypts them using k_f, and checks if the queried keyword w is included in any of them. If w exists in some documents, their document identifiers are updated to $D[w]$. After that, each id in $D[w]$ allows retrieval of the state list $st_{w_c}^{del} \supset \{c_{id}\}$, where c_{id} refers to the state used when the document id is added for keyword w. By excluding the states in $st_{w_c}^{del}$ from the set of $\{0, ..., ST[w]\}$, the enclave obtains a state list st_{w_c} for the non-deleted documents. The enclave computes $|st_{w_c}|$ number of encrypted index pairs (u, k_{id})s to form search query Q_w. Note that each pair (u, k_{id}) is added to Q_w with probability p (typically, p will be close to 1), aiming to retrieve matching documents with false negatives of a small probability $(1 - p)$. Therefore, it holds

$$Q_w[i] \sim \textbf{Bern}(p), \ \textit{where } i \leq |st_{w_c}|. \tag{1}$$

The enclave sends Q_w to the server. The server decrypts $M_I[u_i]$ with k_{id_i}, stores obtained document identifiers in Q_{docId}, and sends Q_{docId} to the enclave. After that, the enclave increments the search counter $SC[id_i]$ of each matching document id_i. When retrieving them, the enclave adds the corresponding encrypted documents to the search result Res without accessing the untrusted area if they are stored in the frequently matched document map FD to minimize the possibility of access pattern leakage to the server, or they are retrieved from the repository R located in the untrusted area, otherwise. Finally, the enclave sends Res to the client and updates FD based on the updated SC.

5. Security Analysis

In this section, we analyze the security of the proposed scheme, and prove that it guarantees forward and Type-II backward privacy.

$Setup$ leaks nothing to the server by leveraging a secure channel established by remote attestation. However, because the adversary can observe the interactions between its memory and the enclave during the search and update procedures, we consider the communications between them as information leakage in the proposed scheme. For addition, $Update$ leaks timestamp and memory access patterns when inserting new entries to the data structures M_I, M_c, and R. For deletion, $Update$ does not reveal any information to the server because there is no interaction between the enclave and the server. In *Search*, the access patterns on M_I, M_c, and R are revealed to the server.

Definition 1 (Obfuscated Search Pattern)**.** *The obfuscated search pattern $\tilde{\Phi}_{\overrightarrow{w}}$ is the vector characterized by Equation (2)*

$$\tilde{\Phi}_{\overrightarrow{w}} = (u_1, ..., u_{|i|}), \text{where } i \leq |st_w|, \tag{2}$$

where u refers to the encrypted index or query token generated during the search.

Definition 2 (Obfuscated Access Pattern)**.** *The obfuscated access pattern $\tilde{\Pi}_{\overrightarrow{w}}$ is the vector*

$$\tilde{\Pi}_{\overrightarrow{w}} = (id_1, id_2, ..., id_{|i-|fd||}), \tag{3}$$

where $|fd|$ refers to the number of documents retrieved from cached map FD.

We formulate the leakage function, and define $\textbf{Real}_{\mathcal{A}}(\lambda)$ and $\textbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ games for an adaptive adversary $\mathcal{A}$ and a polynomial time simulator $\mathcal{S}$. $\mathcal{D}$ denotes the proposed scheme, and the leakage function of $\mathcal{D}$ is

$$\mathcal{L} = (\mathcal{L}^{Stp}, \mathcal{L}^{Updt}, \mathcal{L}^{Srch}, \mathcal{L}^{hw}).$$

Note that the first three leakage functions define the information exposed in $Setup$, $Update$, and $Search$, respectively. $\mathcal{L}^{hw}$ refers to the inherent leakage of the enclave exposed during the interaction between the enclave and the server.

$Setup$ leaks nothing to the server, but the data structures of M_I, M_c, and R are revealed to the server during the initialization. $Update$ leaks information to the server only when op $= add$. The access pattern of encrypted entries is observed by the server when they are inserted to M_I, M_c, and R. When $op = del$, $\mathcal{D}$ leaks nothing to the server because there is no interaction between the enclave and the server. Therefore,

$$\mathcal{L}^{Updt}(\text{op}, \text{in}) = (T_1, T_2, R[id_i]).$$

T_1 is a tuple of $\{(u, v)\}$ that is inserted to M_I (i.e., encrypted index), and T_2 is a tuple of $\{(u', v')\}$ that is inserted to M_c (i.e., encrypted map of keyword states). $R[id_i]$ refers to the encrypted document inserted with document identifier id_i.

$Search$ leaks obfuscated search pattern $\tilde{\Phi}_{\vec{w}}$ when the enclave sends query tokens to the server, obfuscated access pattern $\tilde{\Pi}_{\vec{w}}$ when it retrieves the encrypted documents from the server, and the patterns on the deleted documents' list d_w. $\mathcal{L}^{Srch}$ is defined as

$$\mathcal{L}^{Srch} = (\tilde{\Phi}_w, \tilde{\Pi}_w, d_w).$$

$\mathcal{L}^{hw}$ includes hardware leakages observed during $Update$ and $Search$, such as memory access patterns, locations, timestamps, and manipulated memory areas for M_I, M_c, and R. $\mathcal{L}^{hw}$ is defined as

$$\mathcal{L}^{hw} = ((M_I, M_c, R)^{Updt}, (M_I, M_c, R)^{Srch}).$$

$\mathbf{Real}_{\mathcal{A}}(\lambda)$: As presented in Algorithm 1, the challenger performs $Setup(1^{\lambda})$ to initialize data structures used by the client, the server, and the enclave. $\mathcal{A}$ selects a database $DB = \{doc_i\}_{i \in \mathbb{N}}$ and performs a polynomial number of updates, where $\mathbb{N}$ is a natural number of documents. When the challenger runs $Update(\text{op}, in)$, $(M_I, M_c, R)^{Updt}$ is returned to $\mathcal{A}$. After that, $\mathcal{A}$ selects a keyword w and performs $Search$. As a result of $Search(w)$, the challenger returns the transcript of each operation and outputs $(M_I, M_c, R)^{Srch}$ to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit b.

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$: $\mathcal{A}$ selects a $DB = \{doc_i\}_{i \in \mathbb{N}}$. $\mathcal{S}$ generates a tuple of (M_I, M_c, R) by using $\mathcal{L}^{Updt}$ and $(M_I, M_c, R)^{Updt}$. Then, $\mathcal{S}$ sends the generated tuple to $\mathcal{A}$. $\mathcal{A}$ adaptively selects the keyword w to search. The transcript obtained by $\mathcal{S}(\mathcal{L}^{Srch}(w))$ is returned by the challenger. Finally, $\mathcal{A}$ returns a bit b.

For all probabilistic polynomial time algorithms $\mathcal{A}$, if there exists a PPT simulator such that

$$|\Pr[\mathsf{Real}_{\mathcal{A}}(\lambda) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda),$$

then $\mathcal{D}$ is $\mathcal{L}$-secure against adaptive chosen-keyword attacks.

Definition 3 ($\mathcal{L}$-security)**.** *Scheme $\mathcal{D}$ consists of three protocols: $Setup$, $Update$, and $Search$. $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ and $\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ are probabilistic experiments, where $\mathcal{A}$ is a stateful adversary and $\mathcal{S}$ is a stateful simulator that gets the leakage function $\mathcal{L}$. $\mathcal{D}$ is $\mathcal{L}$-secure if $\mathcal{A}$ can distinguish $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ and $\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ with negligible probability.*

The scheme $\mathcal{D}$ is secure if it achieves both forward and Type-II backward privacy. Since the client issues a query on keyword w to the enclave through the secure channel, $\mathcal{A}$ has to generate a query token by itself in the game presented in Definition 3. Forward privacy is guaranteed because state $ST[w]$ increases when a new document containing w is inserted. The increase in state value constrains $\mathcal{A}$ to generate a query token to retrieve newly added documents. Regarding backward privacy, $\mathcal{A}$ can learn the timestamps indicating when the deleted states of w were added in M_c, when the enclave requests the server to access M_c during $Search$, but $\mathcal{A}$ cannot know when they were actually requested for deletion. Scheme $\mathcal{D}$ caches deletion requests in the enclave and only accesses them during $Search$. Due to

the obfuscation technique in the scheme, during the search on w, the enclave generates the query tokens and includes them with probability p. In other words, a false negative occurs with probability $1 - p$, which probabilistically omits the query tokens in the result. Therefore, $\mathcal{A}$ is unable to identify whether the tokens are omitted due to deletion operations or false negatives. In addition, the enclave stores the most frequently retrieved document. Thus, if the matching document exists within the map FD, the enclave does not request the server to access R for those documents. As a result, $\mathcal{A}$ does not know which delete updates are conducted for specific document identifiers.

Theorem 1. *Scheme $\mathcal{D}$ is $\mathcal{L}$-secure according to Definition 3.*

Proof. We now prove Theorem 1 by illustrating a PPT simulator $\mathcal{S}$ for which $\mathcal{A}$, a PPT adversary, can distinguish $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ with negligible probability.

- *Setup* : $\mathcal{S}$ performs a random key generation $\tilde{K} = (\tilde{k}_\sigma, \tilde{k}_f)$ to simulate the key components provisioned inside the enclave.
- *Simulate* : $\mathcal{S}$ executes $Update$ on a random keyword w and obtains a query token q. Then, $\mathcal{S}$ performs an addition update for w based on $\tilde{K}$ and $\mathcal{L}^{hw}(M_I, M_c, R)$, and passes them to the enclave to receive the new update of (M_I, M_c, R).
  However, $\mathcal{A}$ cannot classify which update tokens match q because the enclave keeps increasing the state $ST[w]$. Thus, $\mathcal{S}$ is unable to distinguish between the output of $\mathbf{Real}_{\mathcal{A}}(\lambda)$ and the simulated output in $Update$ and *Search*, guaranteeing forward privacy.
- *Simulate* : $\mathcal{S}$ executes *Search* on a random keyword w. The encrypted documents in the deleted document list d stored in the enclave are only requested during *Search*. Thus, $\mathcal{S}$ is unable to specify the exact timestamp for deletion updates. When generating query token during *Search*, false negative occurs with rate $1 - p$, causing probabilistic omission of query token via $\tilde{\Phi}_{\vec{w}}$. In addition, for generated query tokens, if the matching encrypted documents are cached within the enclave, those tokens are omitted from the result sent to $\mathcal{S}$ via $\tilde{\Pi}_{\vec{w}}$.
  When $\mathcal{A}$ compares the matching document identifier lists of the same query, $\tilde{\Phi}_{\vec{w}}$ and $\tilde{\Pi}_{\vec{w}}$ prevents $\mathcal{A}$ from learning information required for reconstructing the deletion history. However, $\mathcal{A}$ is able to learn the timestamps of the inserted entries related to specific *id* via $\mathcal{L}^{hw}$. Hence, the scheme $\mathcal{D}$ guarantees Type-II backward privacy.

□

6. Implementation and Evaluation

In this section, we evaluate our scheme in comparison to those of Amjad et al. (Fort, Bunker-A, and Bunker-B) [8] and Vo et al. (SGX-SE1, SGX-SE2, Maiden) [11,12], which are state-of-the-art SGX-based DSSE schemes. The comparison results related to performance and security are shown in Tables 4 and 5.

6.1. Implementation Setup

We implemented the prototype of our scheme in C++ using Intel SGX SDK 2.15.1 in a system with SGX-enabled Intel i5-8500 3.0 GHz, 16 GB RAM, and Ubuntu 18.04.5 LTS. Also, we used a synthetic dataset [11] consisting of 100,000 documents with a keyword frequency following the Zipf distribution.

The prototype utilizes cryptographic primitives in the SGX SDK for supporting cryptographic operations. We also used APIs provided by SDK to create, manage, and access the enclave. Note that the enclave is only allocated 96 MB memory [7], thus using memory more than 96 MB requires the paging mechanism [7]. As paging triggers extra overhead to the system, we leveraged batch processing during the search. In the experiment, we set the batch size to 1×10^4 for all schemes. For large datasets, the size of the query token may be large; thus, by dividing the size of the search query to the multiple batches, we minimize the chances of a paging mechanism.

Table 4. Comparison of SGX-based DSSE schemes.

Schemes	Computation		Communication		Backward Privacy Type
	Search	Update	Search	Update	
Fort	$\mathcal{O}(n_w + \Sigma_w d_w)$	$\mathcal{O}\left(\log^2 N\right)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	I
Maiden	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	I
Bunker-B	$\mathcal{O}(a_w)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	II
SGX-SE1	$\mathcal{O}(n_w + d)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	II
SGX-SE2	$\mathcal{O}(n_w + v_d)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	II
Proposed	$\mathcal{O}(n_w + d)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	II
Bunker-A	$\mathcal{O}(a_w)$	$\mathcal{O}(1)$	$\mathcal{O}(n_w)$	$\mathcal{O}(1)$	III

N: the total number of keyword/document pairs; W: the total number of keywords; a_w, n_w, d: the total number of entries (including all updates) performed on w, the number of non-deleted documents containing w, and the number of deleted documents, respectively; v_d: the vector of a bloom filter to check the membership of #d documents.

Table 5. Storage comparison.

Schemes	Storage	
	Client	Enclave
Fort	$\mathcal{O}(W \log D)$	-
Maiden	$\mathcal{O}(W \log D + a_w + N)$	-
Bunker-B	$\mathcal{O}(W \log D)$	-
SGX-SE1	-	$\mathcal{O}(W \log D + d)$
SGX-SE2	-	$\mathcal{O}\left(W \log D + \lvert\vec{bf}\rvert\right)$
Proposed	-	$\mathcal{O}(W \log D + d + \lvert K\rvert)$
Bunker-A	$\mathcal{O}(W \log D)$	-

N: the total number of keyword/document pairs; D: the total number of documents; W: the total number of keywords; a_w: the total number of entries (including all updates) performed on w; d: the number of deleted documents; $\vec{bf}$: the configurable bloom filter vector; K: the number of frequently searched documents cached in FD.

6.2. Performance

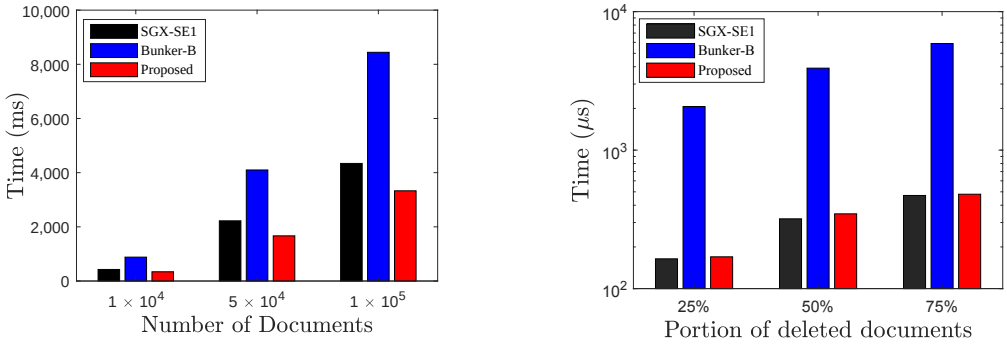
We now evaluate the performance of each scheme. In the evaluation, we compare our scheme mainly with SGX-SE1 and Bunker-B, because they are both designed using Intel SGX with the same assumptions and system environment. Bunker-B provides the same security level as the proposed scheme. Even though SGX-SE1 was originally designed to provide Type-II backward privacy with high efficiency, we prove it guarantees Type-II privacy partially. Although SGX-SE2 also provides the same level of backward privacy as SGX-SE1, it is a simple extension from SGX-SE1 adopting a bloom filter. Thus, SGX-SE2 is not considered in the performance evaluation.

6.2.1. Insertion and Deletion Time

We evaluate the time required for insertion and deletion updates. As shown in Table 4, both the computation and communication costs for update (i.e., addition and deletion) are $\mathcal{O}(1)$ in time complexity. However, when evaluated with real-world implementation using a synthetic dataset, there is a noticeable difference in efficiency. In the experiment, we measured the runtime for adding 1×10^4, 5×10^4, and 1×10^5 documents into the EDB in each scheme. As shown in Figure 3a, Bunker-B takes 881 ms, 4099 ms, and 8443 ms to insert 1×10^4, 5×10^4, 1×10^5 documents, respectively. SGX-SE1 takes 426 ms, 2221 ms, and 4340 ms, and the proposed scheme takes 341 ms, 1667 ms, and 3329 ms for each respective

case, demonstrating the proposed scheme outperforms SGX-SE1 slightly. Compared to Bunker-B, both the proposed scheme and SGX-SE1 show significantly higher efficiency as the number of inserted documents increases.

For deletion, we evaluate the runtime by varying the portion of deletion from 1×10^5 documents: 25%, 50%, and 75%. As shown in Figure 3b, Bunker-B takes 2065 ms, 3907 ms, and 5885 ms to delete 25%, 50%, and 75% of documents, respectively. SGX-SE1 takes 164 ms, 319 ms, and 470 ms, and the proposed scheme takes 169 ms, 346 ms, and 480 ms for each respective case. It shows the proposed scheme and SGX-SE1 have almost the same computational overhead while significantly outperforming Bunker-B. This is mainly because the proposed scheme and SGX-SE1 implement the deletion process by simply inserting the *id*s of the documents to be deleted into a list, but Bunker-B actually deletes the documents from the EDB as in the insertion process.



(**a**) (**b**)

Figure 3. Total time for insertion/deletion. (**a**) Insertion. (**b**) Deletion.

6.2.2. Search Time

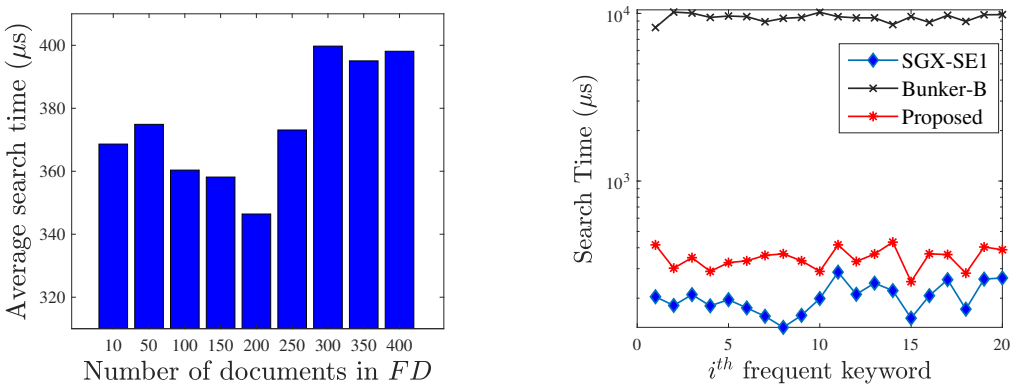
Now, we evaluate the search time or query delay of the proposed scheme. Because the proposed scheme caches frequently searched documents FD within the enclave for obfuscation and faster retrieval, it is important to find the optimal size of FD. Figure 4a shows the search time of the proposed scheme with various sizes of FD. The search time decreases as the number of documents in FD increases up to 200; but, when more than 250 documents are stored, the search time rather increases. We observed that this is mainly due to the limitation of the enclave memory, which triggers the paging mechanism more frequently to process the search query when it caches more than 250 documents. With the synthetic data with 1×10^5 documents, we set the optimal size of FD as 200 which shows the average search time of 346 µs.

Next, with the optimal size of $|FD| = 200$, we compare the search time of the proposed scheme with those of Bunker-B and SGX-SE1. Figure 4b illustrates the search time when inserting 1×10^5 documents and deleting 50% of the documents. In the experiment, we select the top 20 frequent keywords to perform the *Search* algorithm. For the top-20 frequent keywords, Bunker-B takes on average 9434 µs to query, SGX-SE1 takes 203 µs, and the proposed scheme takes 348 µs. Because Bunker-B requires documents to be re-encrypted, it has a larger computational overhead of which computation complexity is $\mathcal{O}(a_w)$. The proposed scheme requires on average 9086 µs less search time (27× faster) than Bunker-B, while guaranteeing a similar level of security, on average 140 µs more search time (1.7× slower) than SGX-SE1 due to the extra computations required to obfuscate access patterns on M_I and R while providing higher security.

6.2.3. Storage Overhead

We can consider three storage types: client, server, and enclave. Among them, the server storage is used to store all of the data for data retrieval such as EDB, encrypted indices, and so on. Because it is a commonly required storage overhead in all of the schemes,

we exclude the server storage when analyzing the storage overhead, and consider only the client and enclave storage, as shown in Table 5.



(**a**) (**b**)

Figure 4. Search latency. (**a**) Search latency depending on $|FD|$. (**b**) Search latency comparison of different schemes.

Among the three schemes, Bunker-B only requires $\mathcal{O}(W \log D)$ storage overhead on the client, without requiring enclave storage. In contrast, the proposed scheme and SGX-SE1 maintain persistent data structures for tracing the deleted documents and their mapping information with keywords within the enclave without incurring any storage overhead on the client. Specifically, SGX-SE1 requires $\mathcal{O}(W \log D + d)$ storage overhead on the enclave for storing the deleted document list d, and the mapping information D between the keyword and the deleted documents in d. The proposed scheme also requires the same enclave storage overhead as SGX-SE1, but requires additional storage for the frequently retrieved K-documents, FD, on the enclave.

6.2.4. Utility Loss

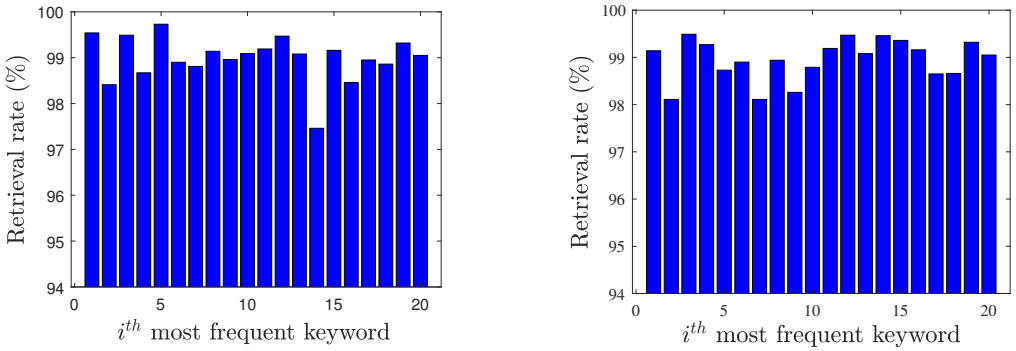
We evaluate the retrieval rate of our scheme with synthetic and real-world datasets.

Synthetic dataset. When generating a search token, we generate it with a probability (true positive rate) p close to 1. Hence, there should be the possibility of false negatives, leading to utility loss. In order to evaluate the utility loss, we insert 1×10^5 documents, delete 20% of the documents, and query the top 20 most frequent keywords by setting $p = 0.9999$. We then repeat the above experiment 20 times and calculate the average rate of document retrieval. Figure 5a illustrates the retrieval rate when querying the ith most frequent keywords. In the experiment, the 14th and 1st most frequent keywords show the lowest and highest rates of 97.46% and 99.54%, respectively. On average, the proposed scheme retrieves 98.98% of the matching documents, showing a 1.02% utility loss.

Real-world dataset. We further evaluate the utility loss of the proposed scheme with a real-world Enron [15] email dataset. For the evaluation, we insert 3.14×10^5 documents, and delete 10% of them. Identical to the previous evaluation with the synthetic dataset, we query the top 20 most frequent keywords and calculate the average rate of document retrieval after repeating the experiment 20 times. Figure 5b shows the retrieval rate when querying the ith most frequent keywords. In the experiment, the 3rd and 2nd most frequent keywords show the highest and the lowest retrieval rates of 99.49% and 98.11%, respectively. On average, the proposed scheme retrieves 98.96% of the matching documents, showing 1.04% utility loss, which is almost the same as the experimental result with the synthetic dataset.

Discussion. The utility loss is an inevitable trade-off for providing higher security without degrading the efficiency of the proposed scheme, as many of the other schemes adopt a differential privacy technique to obfuscate access and search patterns [24,25]. Even if the utility loss rate is very low (which is approximately 1% in the proposed scheme), the possibility of utility loss should affect the number of documents retrievable by the client. A simple solution to mitigate this problem is to add redundancy in the stored data

or to send redundant queries for a single search. For example, Chen et al. scheme [24] adopted erasure code to add redundancy to the stored data. On retrieval, the encoded message can be reconstructed into its original one even though some parts of the message are lost. Therefore, the erasure coding method may be applied to our design scheme to eliminate utility loss. However, how to minimize such additional overheads further without incurring any utility loss while guaranteeing Type-II backward privacy is a challenging and open problem in the literature.



(**a**) (**b**)

Figure 5. Retrieval rate of the *i*th most frequent keywords. (**a**) Synthetic dataset. (**b**) Enron email dataset.

7. Related Work

Since the first proposal in 2000 [26], a number of searchable symmetric encryption (SSE) schemes have been proposed for formally defining the security model [3], improving efficiency [1,27], and supporting expressive queries [28,29]. Recently, several works have been introduced on how access pattern and search pattern leakages can be exploited by an adversary to break the security of SSE [17,30]. Cash et al. [30] exploited the search pattern leakages to recover the plaintext of the query or reconstruct the client's indexed documents. Zhang et al. [17] introduced a file injection attack on SSE schemes that exploits access patterns on file identifiers to identify what specific file identifiers correspond to the maliciously injected file.

Another important research area in SSE is to support dynamic operations in SSE, which enables a client to perform keyword searches as well as update operations on the encrypted documents, known as dynamic symmetric searchable encryption (DSSE) [1,27]. Stefanov et al. [2] and Bost et al. [4,5] first introduced two fundamental security requirements for DSSE, that is, forward and backward privacy. Recently, Sun et al. [14] introduced a non-interactive forward and backward private DSSE method by constructing revocable encryption. Even though it could reduce the information leakage by making the deletions oblivious to the server, it allows additional information leakage such as the number of deleted documents and candidate timestamps of deletion updates, which can be further exploited by an adversary to violate the backward privacy via obtaining the deletion history. In addition, Vo et al. [11] proposed a Type-II backward private scheme by leveraging Intel SGX enclave as a server-side proxy and constructed a more efficient scheme than Amjad et al. Bunker-B [8]. Despite the improvement of its efficiency, the Vo et al. [11] scheme allows more information leakages than Bunker-B, which can be abused to violate its backward privacy. Therefore, accomplishing high efficiency while minimizing exploitable information leakage is still a challenging and open problem in DSSE literature.

8. Conclusions

In this paper, we first conduct a comprehensive analysis of Type-II backward privacy in the aspects of both theoretic definition and practical constructions. We then demonstrate how information leakage that is not covered by the current notions can be utilized to extract the deletion history from Type-II backward private schemes under specific conditions and violate their Type-II privacy in practice. Finally, we propose a novel forward and Type-II backward private DSSE scheme based on Intel SGX, and formally prove the security. The

proposed scheme outperforms the previous works in either search latency or security level with negligible utility loss. To the best of our knowledge, this is the first work that investigates the importance of secondary information leakages that are not captured in traditional backward privacy notions and discusses their practical implications on the backward privacy schemes.

Author Contributions: Conceptualization, H.Y., C.H., D.K. and J.H.; methodology, H.Y. and M.Y.; validation, H.Y., C.H., D.K. and J.H.; formal analysis, H.Y., C.H., D.K. and J.H.; investigation, H.Y. and M.Y.; writing—original draft preparation, H.Y. and C.H.; writing—review and editing, H.Y., C.H., D.K. and J.H.; supervision, C.H., D.K. and J.H. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported as part of the Military Crypto Research Center (UD210027XD) funded by the Defense Acquisition Program Administration (DAPA) and the Agency for Defense Development (ADD).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available in [MonashCybersecurityLab /SGXSSE] at [https://github.com/MonashCybersecurityLab/SGXSSE], and in [Enron Email Dataset at [https://www.cs.cmu.edu/~enron/], reference number [15].

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Kamara, S.; Papamanthou, C.; Roeder, T. Dynamic searchable symmetric encryption. In Proceedings of the 2012 ACM Conference on Computer and Communications Security, Raleigh, NC, USA, 16–18 October 2012; pp. 965–976.
2. Stefanov, E.; Papamanthou, C.; Shi, E. Practical dynamic searchable encryption with small leakage. *Cryptol. ePrint Arch.* **2013**. https://eprint.iacr.org/2013/832, accessed on 5 March 2024.
3. Curtmola, R.; Garay, J.; Kamara, S.; Ostrovsky, R. Searchable symmetric encryption: Improved definitions and efficient constructions. *J. Comput. Secur.* **2011**, *19*, 895–934. [CrossRef]
4. Bost, R. Forward secure searchable encryption. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 1143–1154.
5. Bost, R.; Minaud, B.; Ohrimenko, O. Forward and backward private searchable encryption from constrained cryptographic primitives. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 1465–1482.
6. Stefanov, E.; Dijk, M.V.; Shi, E.; Chan, T.H.H.; Fletcher, C.; Ren, L.; Yu, X.; Devadas, S. Path ORAM: An extremely simple oblivious RAM protocol. *J. ACM (JACM)* **2018**, *65*, 18. [CrossRef]
7. Costan, V.; Devadas, S. Intel SGX explained. *Cryptol. ePrint Arch.* **2016**. Available online: https://eprint.iacr.org/2016/086 (accessed on 5 March 2024).
8. Amjad, G.; Kamara, S.; Moataz, T. Forward and backward private searchable encryption with SGX. In Proceedings of the 12th European Workshop on Systems Security, Dresden, Germany, 25 March 2019; pp. 1–6.
9. Fuhry, B.; Bahmani, R.; Brasser, F.; Hahn, F.; Kerschbaum, F.; Sadeghi, A.R. HardIDX: Practical and secure index with SGX. In Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy, Philadelphia, PA, USA, 19–21 July 2017; Springer: Cham, Switzerland, 2017; pp. 386–408.
10. Priebe, C.; Vaswani, K.; Costa, M. EnclaveDB: A secure database using SGX. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 264–278.
11. Vo, V.; Lai, S.; Yuan, X.; Sun, S.F.; Nepal, S.; Liu, J.K. Accelerating forward and backward private searchable encryption using trusted execution. In Proceedings of the International Conference on Applied Cryptography and Network Security, Rome, Italy, 19–22 October 2020; Springer: Cham, Switzerland, 2020; pp. 83–103.
12. Vo, V.; Lai, S.; Yuan, X.; Nepal, S.; Liu, J.K. Towards efficient and strong backward private searchable encryption with secure enclaves. In Proceedings of the International Conference on Applied Cryptography and Network Security, Kamakura, Japan, 21–24 June 2021; Springer: Cham, Switzerland, 2021; pp. 50–75.
13. Ghareh Chamani, J.; Papadopoulos, D.; Papamanthou, C.; Jalili, R. New constructions for forward and backward private symmetric searchable encryption. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 1038–1055.
14. Sun, S.F.; Steinfeld, R.; Lai, S.; Yuan, X.; Sakzad, A.; Liu, J.K.; Nepal, S.; Gu, D. Practical Non-Interactive Searchable Encryption with Forward and Backward Privacy. In Proceedings of the NDSS, Online, 21–25 February 2021.
15. Klimt, B.; Yang, Y. Introducing the Enron corpus. In Proceedings of the CEAS, Mountain View, CA, USA, 30–31 July 2004.

16. Chang, Y.C.; Mitzenmacher, M. Privacy preserving keyword searches on remote encrypted data. In Proceedings of the International Conference on Applied Cryptography and Network Security, New York, NY, USA, 7–10 June 2005 ; Springer: Cham, Switzerland, 2005; pp. 442–455.
17. Zhang, Y.; Katz, J.; Papamanthou, C. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In Proceedings of the 25th USENIX Security Symposium (USENIX Security 16), Austin, TX, USA, 10–12 August 2016; pp. 707–720.
18. Mishra, P.; Poddar, R.; Chen, J.; Chiesa, A.; Popa, R.A. Oblix: An efficient oblivious search index. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 279–296.
19. Biondo, A.; Conti, M.; Davi, L.; Frassetto, T.; Sadeghi, A.R. The Guard's Dilemma: Efficient {Code-Reuse} Attacks Against Intel {SGX}. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 14 May 2018; pp. 1213–1227.
20. Weichbrodt, N.; Kurmus, A.; Pietzuch, P.; Kapitza, R. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In Proceedings of the European Symposium on Research in Computer Security, Heraklion, Greece, 26–30 September 2016; Springer: Cham, Switzerland, 2016; pp. 440–457.
21. Murdock, K.; Oswald, D.; Garcia, F.D.; Van Bulck, J.; Gruss, D.; Piessens, F. Plundervolt: Software-based fault injection attacks against Intel SGX. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1466–1482.
22. Carofiglio, G.; Gallo, M.; Muscariello, L.; Perino, D. Modeling data transfer in content-centric networking. In Proceedings of the 2011 23rd International Teletraffic Congress (ITC), San Francisco, CA, USA, 6–9 September 2011; pp. 111–118.
23. Fenner, T.; Levene, M.; Loizou, G. A stochastic evolutionary model generating a mixture of exponential distributions. *Eur. Phys. J. B* **2016**, *89*, 50. [CrossRef]
24. Chen, G.; Lai, T.H.; Reiter, M.K.; Zhang, Y. Differentially private access patterns for searchable symmetric encryption. In Proceedings of the IEEE INFOCOM 2018—IEEE Conference on Computer Communications, Honolulu, HI, USA, 16–19 April 2018; pp. 810–818.
25. Shang, Z.; Oya, S.; Peter, A.; Kerschbaum, F. Obfuscated Access and Search Patterns in Searchable Encryption. In Proceedings of the NDSS, Online, 21–25 February 2021.
26. Song, D.X.; Wagner, D.; Perrig, A. Practical techniques for searches on encrypted data. In Proceedings of the Proceeding 2000 IEEE Symposium on Security and Privacy: S&P 2000, Berkeley, CA, USA, 14–17 May 2000; pp. 44–55.
27. Kamara, S.; Papamanthou, C. Parallel and dynamic searchable symmetric encryption. In Proceedings of the International Conference on Financial Cryptography and Data Security, Okinawa, Japan, 1–5 April 2013; Springer: Berlin/Heidelberg, Germany, 2013; pp. 258–274.
28. Lai, J.; Zhou, X.; Deng, R.H.; Li, Y.; Chen, K. Expressive search on encrypted data. In Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, Hangzhou, China, 8–10 May 2013; pp. 243–252.
29. Yang, Y.; Liu, X.; Deng, R. Expressive query over outsourced encrypted data. *Inf. Sci.* **2018**, *442*, 33–53. [CrossRef]
30. Cash, D.; Grubbs, P.; Perry, J.; Ristenpart, T. Leakage-abuse attacks against searchable encryption. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 668–679.